USE CASE -1

An operating system uses the Banker's algorithm for deadlock avoidance when managing the allocation of three resource types X, Y, and Z to three processes P0, P1, and P2. The table given below presents the current system state. Here, the Allocation matrix shows the current number of resources of each type allocated to each process and the Max matrix shows the maximum number of resources of each type required by each process during its execution.

	Alloc			Req		
	X	**Y**	**Z**	**X**	**Y**	**Z**
P0	**0**	**0**	**1**	**8**	**4**	**3**
P1	**3**	**2**	**0**	**6**	**2**	**0**
P2	**2**	**1**	**1**	**3**	**3**	**3**

There are 3 units of type X, 2 units of type Y and 2 units of type Z still available. The system is currently in a safe state. Consider the following independent requests for additional resources in the current state:

REQ1: P0 ->requests -> (0,0,2)

REQ2: P1 requests -> (2,0,0)

Consider the following scenario, where the column alloc denotes the number of units of each resource type allocated to each process, and the column request denotes the number of units of each resource type requested by a process in order to complete execution.

Which of these processes will finish LAST? Develop a program to check whether the system is in safe state or not?

```
#include <stdio.h>

#define NUM_PROCESSES 3

#define NUM_RESOURCES 3

int available[NUM_RESOURCES] = {3, 2, 2};

int max[NUM_PROCESSES][NUM_RESOURCES] = {

   {8, 4, 3},

   {6, 2, 0},

   {3, 3, 3}

};

int allocation[NUM_PROCESSES][NUM_RESOURCES] = {
```

```
    {0, 0, 1},
    {3, 2, 0},
    {2, 1, 1}
};
int request[NUM_PROCESSES][NUM_RESOURCES] = {
    {0, 0, 2},
    {2, 0, 0},
    {0, 0, 0}
};
int finish[NUM_PROCESSES] = {0};
int work[NUM_RESOURCES];
int safeSequence[NUM_PROCESSES];
int isSafeState() {
    int i, j;
    for (i = 0; i < NUM_RESOURCES; i++) {
        work[i] = available[i];
    }
    int count = 0;
    while (count < NUM_PROCESSES) {
        int found = 0;
        for (i = 0; i < NUM_PROCESSES; i++) {
            if (!finish[i]) {
                int canExecute = 1;
                for (j = 0; j < NUM_RESOURCES; j++) {
                    if (request[i][j] > work[j]) {
                        canExecute = 0;
                        break;
                        for (j = 0; j < NUM_RESOURCES; j++)
                    work[j] += allocation[i][j];
```

```
                    }
                    safeSequence[count++] = i;
                    finish[i] = 1;
                    found = 1;
                }
            }
        }
        if (!found) {
            return 0; // System is not in a safe state
        }
    }
    return 1; // System is in a safe state
}

int main() {
    if (isSafeState()) {
        printf("System is in a safe state.\n");
        printf("Safe Sequence: ");
        for (int i = 0; i < NUM_PROCESSES; i++) {
            printf("P%d ", safeSequence[i]);
        }
        printf("\n");
    } else {
        printf("System is not in a safe state.\n");
    }
    return 0;
}
```

USE CASE-2

Interpret a C program to simulate memory management scheme using page replacement algorithms. Consider the reference string 6, 1, 1, 2, 0, 3, 4, 6, 0, 2, 1, 2, 1, 2, 0, 3, 2, 1, 2, 0 for a memory with Four frames and calculate number of page faults by using FIFO, LRU and OPTIMAL Page replacement algorithms.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_FRAMES 4
   int fifo(int reference_string[], int n);
   int lru(int reference_string[], int n);
   int optimal(int reference_string[], int n);
   int main() {
   int reference_string[] = {6, 1, 1, 2, 0, 3, 4, 6, 0, 2, 1, 2, 1, 2, 0, 3, 2, 1, 2, 0};
   int n = sizeof(reference_string) / sizeof(reference_string[0]);
   int fifo_faults = fifo(reference_string, n);
   int lru_faults = lru(reference_string, n);
   int optimal_faults = optimal(reference_string, n);
   printf("FIFO Page Faults: %d\n", fifo_faults);
   printf("LRU Page Faults: %d\n", lru_faults);
   printf("OPTIMAL Page Faults: %d\n", optimal_faults);
   return 0;
}
   int fifo(int reference_string[], int n) {
   int frames[MAX_FRAMES];
   int frame_index = 0;
   int faults = 0;
   for (int i = 0; i < MAX_FRAMES; i++) {
      frames[i] = -1; // Initialize frames with -1 to indicate empty slots
   }
      for (int i = 0; i < n; i++) {
      int page = reference_string[i];
      int page_found = 0
      for (int j = 0; j < MAX_FRAMES; j++) {
         if (frames[j] == page) {
            page_found = 1;
            break;
         }
      }
         if (!page_found) {
         frames[frame_index] = page;
         frame_index = (frame_index + 1) % MAX_FRAMES;
         faults++;
      }
   }
         return faults;
}
   int lru(int reference_string[], int n) {
```

```
int frames[MAX_FRAMES];
int lru_count[MAX_FRAMES];
int faults = 0;
for (int i = 0; i < MAX_FRAMES; i++) {
    frames[i] = -1; // Initialize frames with -1 to indicate empty slots
    lru_count[i] = 0;
}
    for (int i = 0; i < n; i++) {
    int page = reference_string[i];
    int page_found = 0;
    for (int j = 0; j < MAX_FRAMES; j++) {
        if (frames[j] == page) {
            page_found = 1;
            lru_count[j] = i; // Update the LRU count for the page
            break;
        }
    }
        if (!page_found) {
        int min_lru_index = 0;
        for (int j = 1; j < MAX_FRAMES; j++) {
            if (lru_count[j] < lru_count[min_lru_index]) {
                min_lru_index = j;
            }
        }
        frames[min_lru_index] = page;
        lru_count[min_lru_index] = i;
        faults++;
    }
}
  return faults;
}
int optimal(int reference_string[], int n) {
int frames[MAX_FRAMES];
int faults = 0;
for (int i = 0; i < MAX_FRAMES; i++) {
    frames[i] = -1; // Initialize frames with -1 to indicate empty slots
}
    for (int i = 0; i < n; i++) {
    int page = reference_string[i];
    int page_found = 0;
        for (int j = 0; j < MAX_FRAMES; j++) {
        if (frames[j] == page) {
            page_found = 1;
            break;
        }
    }
        if (!page_found) {
        int farthest_use = -1;
        int replace_index = -1;
       for (int j = 0; j < MAX_FRAMES; j++) {
```

```
                int page_in_future = 0;
                for (int k = i + 1; k < n; k++) {
                    if (reference_string[k] == frames[j]) {
                        page_in_future = 1;
                        if (k > farthest_use) {
                            farthest_use = k;
                            replace_index = j;
                        }
                        break;
                    }
                }
                if (!page_in_future) {
                    replace_index = j;
                    break;
                }
            }
            frames[replace_index] = page;
            faults++;
        }
    }
    return faults;
}
```

USE CASE-3

Demonstrate a C program to simulate the following contiguous memory allocation techniques. Consider six memory partitions of size 100 KB, 500 KB, 200 KB, 300 KB, 600 KB . These partitions need to be allocated to four processes of sizes 212 KB, 417 KB, 112 KB and 426 KB in that order.
Perform the allocation of processes using-

1. First Fit Algorithm
2. Best Fit Algorithm
3. Worst Fit Algorithm

```
#include <stdio.h>

// Define the number of memory partitions and processes

#define NUM_PARTITIONS 5

#define NUM_PROCESSES 4

// Function to allocate memory using First Fit algorithm

void firstFit(int partitions[], int m, int processSize[], int n) {

 int allocation[n];

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

      allocation[i] = -1; // Initialize allocation as -1 (unallocated)


 for (int j = 0; j < m; j++) {

            if (partitions[j] >= processSize[i]) {

               allocation[i] = j; // Allocate process to partition j

               partitions[j] -= processSize[i]; // Update available memory in partition j

               break;

            }

      }

   }

   // Display the allocation results

   printf("\nFirst Fit Allocation:\n");

   printf("Process Size   Partition Index\n");

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

      printf("%d KB           ", processSize[i]);

      if (allocation[i] != -1) {
```

```
            printf("%d\n", allocation[i]);
        } else {
            printf("Not Allocated\n");
        }
    }
}
// Function to allocate memory using Best Fit algorithm
void bestFit(int partitions[], int m, int processSize[], int n) {
    int allocation[n];
        for (int i = 0; i < n; i++) {
        allocation[i] = -1; // Initialize allocation as -1 (unallocated)
        int bestFitIdx = -1;
            for (int j = 0; j < m; j++) {
            if (partitions[j] >= processSize[i]) {
                if (bestFitIdx == -1 || partitions[j] < partitions[bestFitIdx])
                bestFitIdx = j; // Update bestFitIdx if a better fit is found
                }
            }
        }
            if (bestFitIdx != -1) {
            allocation[i] = bestFitIdx; // Allocate process to best-fit partition
            partitions[bestFitIdx] -= processSize[i]; // Update available memory in the best-fit
partition
        }
    }
    // Display the allocation results
    printf("\nBest Fit Allocation:\n");
    printf("Process Size   Partition Index\n");
    for (int i = 0; i < n; i++) {
        printf("%d KB           ", processSize[i]);
        if (allocation[i] != -1) {
            printf("%d\n", allocation[i]);
```

```
        } else {
            printf("Not Allocated\n");
        }
    }
}
// Function to allocate memory using Worst Fit algorithm
void worstFit(int partitions[], int m, int processSize[], int n) {
    int allocation[n];
for (int i = 0; i < n; i++) {
        allocation[i] = -1; // Initialize allocation as -1 (unallocated)
        int worstFitIdx = -1;
 for (int j = 0; j < m; j++) {
            if (partitions[j] >= processSize[i]) {
                if (worstFitIdx == -1 || partitions[j] > partitions[worstFitIdx]) {
                    worstFitIdx = j; // Update worstFitIdx if a worse fit is found
                }
            }
        }
            if (worstFitIdx != -1) {
            allocation[i] = worstFitIdx; // Allocate process to worst-fit partition
            partitions[worstFitIdx] -= processSize[i]; // Update available memory in the worst-fit
partition
        }
    }
    // Display the allocation results
    printf("\nWorst Fit Allocation:\n");
    printf("Process Size   Partition Index\n");
    for (int i = 0; i < n; i++) {
        printf("%d KB           ", processSize[i]);
        if (allocation[i] != -1) {
            printf("%d\n", allocation[i]);
        } else {
```

```
            printf("Not Allocated\n");
        }
    }
}
    int main() {
    int partitions[NUM_PARTITIONS] = {100, 500, 200, 300, 600};
    int processSize[NUM_PROCESSES] = {212, 417, 112, 426};
    firstFit(partitions, NUM_PARTITIONS, processSize, NUM_PROCESSES);
    bestFit(partitions, NUM_PARTITIONS, processSize, NUM_PROCESSES);
    worstFit(partitions, NUM_PARTITIONS, processSize, NUM_PROCESSES);
    return 0;
}
```

USE CASE-4

Implement a C program to simulate the following Disk Scheduling techniques
a)FCFS b)SSTF c)SCAN
Suppose that disk drive has 200 cylinders numbered from 0 to 200.The drive is currently serving a request at cylinder 50 .The queue of the pending requests, in FIFO order,is 176, 79, 34, 60, 92, 11, 41, 114 starting from the current head , what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for the above mentioned disk scheduling algorithms?

```
#include<stdio.h>

#include <stdlib.h>

#define CYLINDERS 200

// Function to simulate FCFS scheduling

int fcfs(int queue[], int n, int start) {

    int totalDistance = 0;

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


        totalDistance += abs(start - queue[i]);

        start = queue[i];

    }

    return totalDistance;

}

// Function to simulate SSTF scheduling

int sstf(int queue[], int n, int start) {

    int totalDistance = 0;

    int visited[n];

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

        visited[i] = 0;

    }

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

        int minDistance = CYLINDERS + 1;

        int minIndex = -1;

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

            if (!visited[j]) {
```

```
            int distance = abs(start - queue[j]);
            if (distance < minDistance) {
                minDistance = distance;
                minIndex = j;
            }
        }
    }
    visited[minIndex] = 1;
    totalDistance += minDistance;
    start = queue[minIndex];
  }
  return totalDistance;
}
// Function to simulate SCAN scheduling
int scan(int queue[], int n, int start) {
  int totalDistance = 0;
  int direction = 1; // 1 for right, -1 for left
  int rightLimit = CYLINDERS;
  int leftLimit = 0;
  while (1) {
    if (direction == 1) {
        int minDistance = CYLINDERS + 1;
        int minIndex = -1;
        for (int i = 0; i < n; i++) {
            if (queue[i] >= start && queue[i] < minDistance) {
                minDistance = queue[i];
                minIndex = i;
            }
        }
        if (minIndex != -1) {
            totalDistance += minDistance - start;
            start = minDistance;
```

```
} else {
        totalDistance += rightLimit - start;
        start = rightLimit;
        direction = -1;
    }
} else {
    int minDistance = CYLINDERS + 1;
    int minIndex = -1;
    for (int i = 0; i < n; i++) {
        if (queue[i] <= start && queue[i] > minDistance) {
            minDistance = queue[i];
            minIndex = i;
        }
    }
    if (minIndex != -1) {
        totalDistance += start - minDistance;
        start = minDistance;
    } else {
        totalDistance += start - leftLimit;
        start = leftLimit;
        direction = 1;
    }
}
int allVisited = 1;
for (int i = 0; i < n; i++) {
    if (queue[i] != -1) {
        allVisited = 0;
        break;
    }
}
    if (allVisited) {
    break;
```

```
    }
    }
    return totalDistance;
}
    int main() {
    int diskQueue[] = {176, 79, 34, 60, 92, 11, 41, 114};
    int n = sizeof(diskQueue) / sizeof(diskQueue[0]);
    int currentHead = 50;
    int fcfsDistance = fcfs(diskQueue, n, currentHead);
    int sstfDistance = sstf(diskQueue, n, currentHead);
    int scanDistance = scan(diskQueue, n, currentHead);
    printf("Total distance for FCFS: %d cylinders\n", fcfsDistance);
    printf("Total distance for SSTF: %d cylinders\n", sstfDistance);
    printf("Total distance for SCAN: %d cylinders\n", scanDistance);
    return 0;
}
```