Name: Jhaveri Varun Nimitt

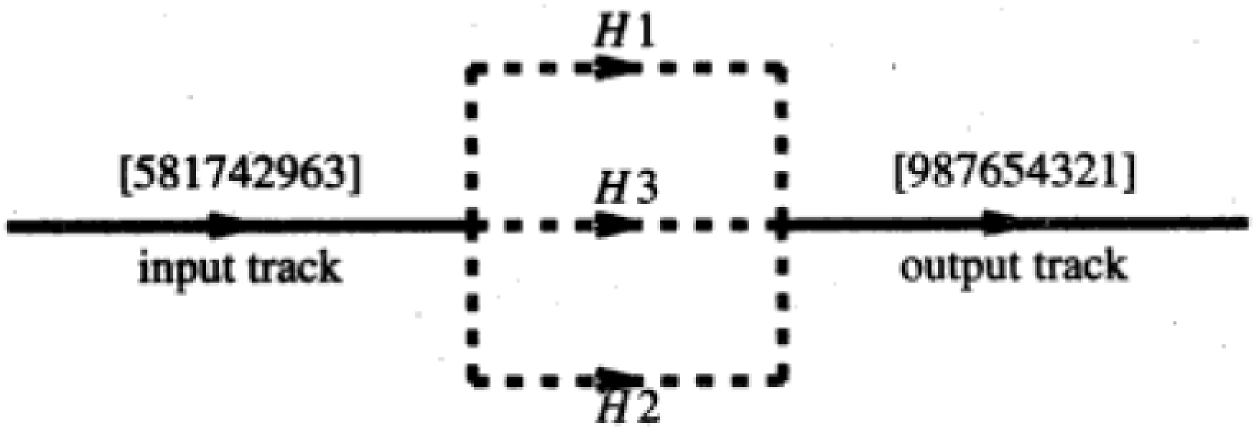
UID: 2023800042

Batch: CSE A Batch C

Experiment No.: 1

Aim: Stack and Queue applicat ion

Problem: **A freight train has n railroad cars. Each is to be left at a different station. Assume that the n stations are numbered 1 through n and that the freight train visits these stations in the order n through 1. The railroad cars are labeled by their destination. To facilitate removal of the railroad cars from the train, we must reorder the cars so that they are in the order 1 through n from front to back. When the cars are in this order, the last car is detached at each station. We rearrange the cars at a shunting yard that has an input track, an output track, and k holding tracks between the input and output tracks. Figure 9.6(a) shows a shunting yard with k = 3 holding tracks H1, H2, and H3. The n cars of the freight train begin in the input track and are to end up in the output track in the order 1 through n from right to left. In Figure 9.6(a), n = 9; the cars are initially in the order 5, 8, 1, 7, 4, 2, 9, 6, 3 from back to front. Figure 9.6(b) shows the cars rearranged in the desired order.**



APPROACH NO 1 : USING QUEUES :

Program:

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

#define N 9
#define K 3

struct Queue {
   int data[N];
   int front, rear, size;
};

struct Queue* createQueue() {
   struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
   queue->front = 0;
   queue->rear = -1;
   queue->size = 0;
   return queue;
}

void enqueue(struct Queue* queue, int value) {
   if (queue->size == N) {
       printf("full\n");
       return;
   }
   queue->rear = (queue->rear + 1) % N;
   queue->data[queue->rear] = value;
   queue->size++;
}

int dequeue(struct Queue* queue) {
   if (queue->size == 0) {
       printf("empty\n");
       return -1;
   }
   int value = queue->data[queue->front];
   queue->front = (queue->front + 1) % N;
   queue->size--;
   return value;
}

int isEmpty(struct Queue* queue) {
   return queue->size == 0;
}

void printQueue(struct Queue* queue) {
   int i = queue->front;
   for (int count = 0; count < queue->size; count++) {
       printf("%d ", queue->data[i]);
       i = (i + 1) % N;
   }
   printf("\n");
}

void printHoldingTracks(struct Queue* holdingTracks[], int k) {
   for (int j = 0; j < k; j++) {
       printf("H%d: ", j + 1);
       printQueue(holdingTracks[j]);
   }
}

int main() {

   int order[N] = {5, 8, 1, 7, 4, 2, 9, 6, 3};

   struct Queue* holdingTracks[K];
   for (int i = 0; i < K; i++) {
       holdingTracks[i] = createQueue();
   }
   int outIndex = 0;
   int output[N] = {0};

   for (int i = 0; i < N; i++) {
       int car = order[i];
       int placed = 0;

       for (int j = 0; j < K; j++) {
           if (isEmpty(holdingTracks[j]) || car <
holdingTracks[j]->data[holdingTracks[j]->rear]) {
               enqueue(holdingTracks[j], car);
               placed = 1;
               break;
           }
       }

       if (!placed) {
           printf("cant place %d anywhere so ig this array isnt possible\n", car);
           return -1;
       }

       //printHoldingTracks(holdingTracks, K);
   }

   while (outIndex < N) {
       int maxCar = 0;
       int maxTrack = -1;

       for (int j = 0; j < K; j++) {
           if (!isEmpty(holdingTracks[j]) &&
               (maxTrack == -1 || holdingTracks[j]->data[holdingTracks[j]->front] >
holdingTracks[maxTrack]->data[holdingTracks[maxTrack]->front])) {
               maxTrack = j;
           }
       }

       if (maxTrack != -1) {
           output[outIndex++] = dequeue(holdingTracks[maxTrack]);
       }
   }

   for (int i = 0; i < N; i++) {
       printf("%d ", output[i]);
   }
   printf("\n");

   for (int i = 0; i < K; i++) {
       free(holdingTracks[i]);
   }

   return 0;
}
```

Observations:

Program Analysis

1. **Main stuff**:
    - holdingTracks is an array of 3 queues. These queues will be used to temporarily hold the cars while sorting.
2. **Logic**:
    - Each car from the order array is placed into one of the holding tracks. It will try to place a car in the track where it can maintain the descending order. If no such track exists (where the car can be placed while keeping the order), the arrangement is not possible.
3. **Final**:
    - It extracts cars from the holding tracks, in descending order of the cars in the front of the tracks, to generate the sorted output.

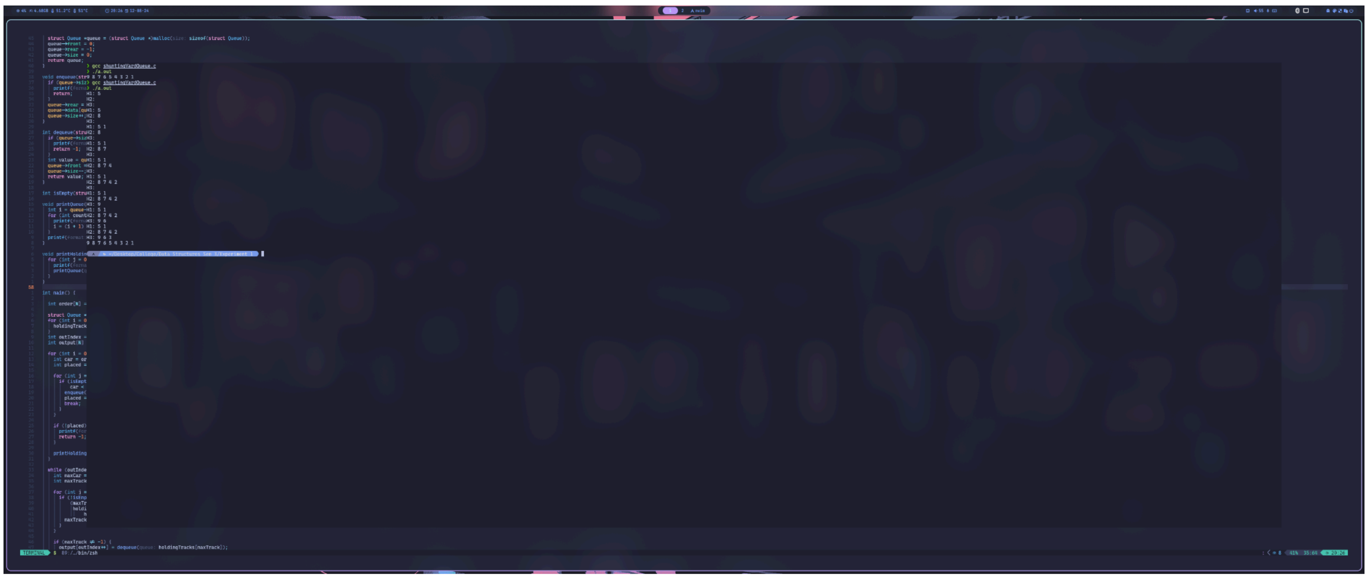
Observations with the Given Test Case

1. **Placing Cars**:
    - The cars are placed into holding tracks such that each track maintains a descending order of cars.
2. **HOw output is genereated**:
    - The cars are extracted from the holding tracks based on the maximum value available at the front of the tracks.

This provides me with the final output which is [9, 8, 7, 6, 5, 4, 3, 2, 1]

(to see step by step movement uncomment line no 89)

OUTPUT: (with both step by step and direct output)



```
struct Queue *queue = (struct Queue *)malloc(size: sizeof(struct Queue));
queue->front = 0;
queue->rear = -1;
queue->size = 0;
return queue;
}
```

```
❯ gcc shuntingYardQueue.c
❯ ./a.out
9 8 7 6 5 4 3 2 1
❯ gcc shuntingYardQueue.c
❯ ./a.out
H1: 5
H2:
H3:
H1: 5
H2: 8
H3:
H1: 5 1
H2: 8
H3:
H1: 5 1
H2: 8 7
H3:
H1: 5 1
H2: 8 7 4
H3:
H1: 5 1
H2: 8 7 4 2
H3:
H1: 5 1
H2: 8 7 4 2
H3: 9
H1: 5 1
H2: 8 7 4 2
H3: 9 6
H1: 5 1
H2: 8 7 4 2
H3: 9 6 3
9 8 7 6 5 4 3 2 1
 ~/Desktop/College/Data Structures Sem 3/Experiment 1
```

```
    if (maxTrack != -1) {
      output[outIndex++] = dequeue(queue: holdingTracks[maxTrack]);
```

```
TERMINAL  $  89:/…/bin/zsh
```

APPROACH NO 2 : USING STACK:

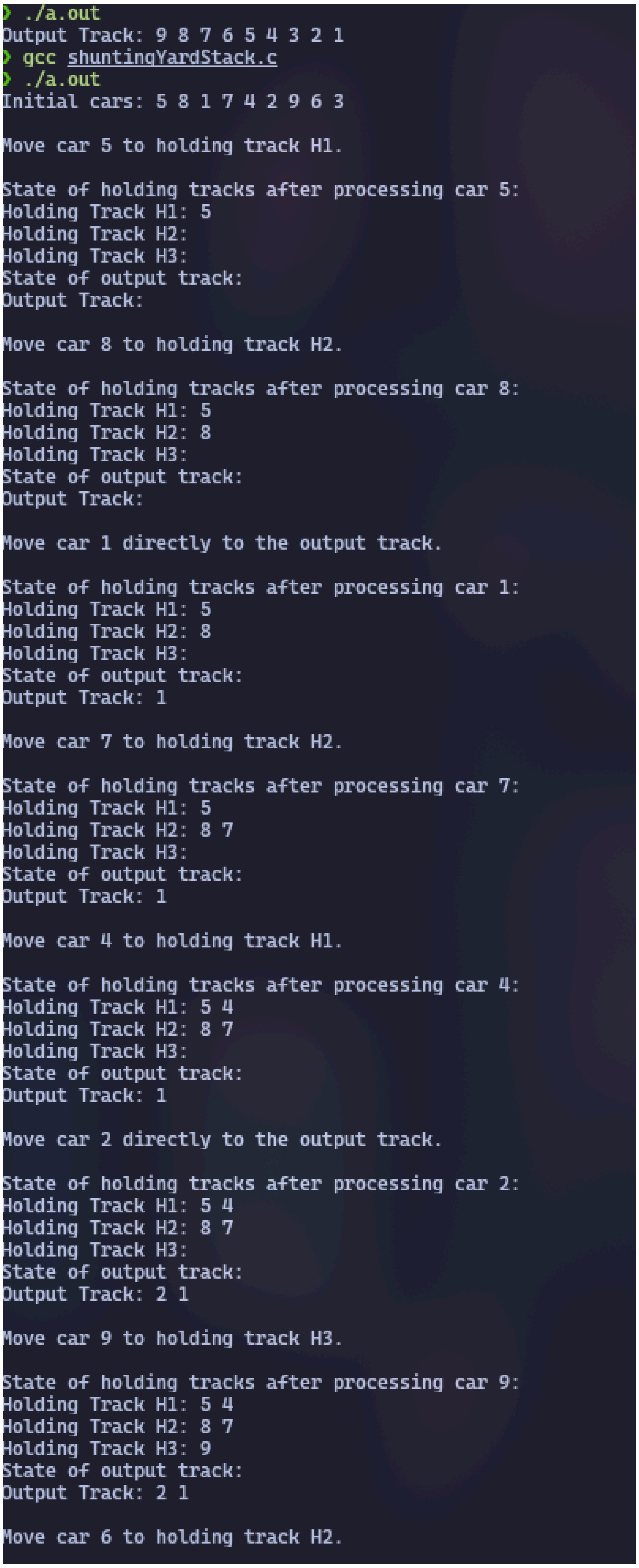
Program:

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

#define MAX_HOLDING_TRACKS 100
#define MAX_CARS 100

typedef struct {
   int top;
   int cars[MAX_CARS];
} Stack;

void initializeStack(Stack* s) {
   s->top = -1;
}

bool isEmpty(Stack* s) {
   return s->top == -1;
}

bool isFull(Stack* s) {
   return s->top == MAX_CARS - 1;
}

void push(Stack* s, int car) {
   if (!isFull(s)) {
       s->cars[++(s->top)] = car;
   }
}

int pop(Stack* s) {
   if (!isEmpty(s)) {
       return s->cars[(s->top)--];
   }
   return -1;
}

int peek(Stack* s) {
   if (!isEmpty(s)) {
       return s->cars[s->top];
   }
   return -1;
}

void printHoldingTracks(Stack* holdingTracks, int k) {
   for (int i = 0; i < k; i++) {
```

```c
        printf("Holding Track H%d: ", i + 1);
        for (int j = 0; j <= holdingTracks[i].top; j++) {
            printf("%d ", holdingTracks[i].cars[j]);
        }
        printf("\n");
    }
}

void printOutputTrack(Stack* outputTrack, int size) {
    printf("Output Track: ");
    for (int i = size - 1; i >= 0; i--) {
        printf("%d ", outputTrack->cars[i]);
    }
    printf("\n");
}

bool rearrangeCars(int n, int k, int* cars, bool debug) {
    if (n <= 0 || k <= 0 || k > MAX_HOLDING_TRACKS) {
        printf("Invalid input.\n");
        return false;
    }

    Stack holdingTracks[MAX_HOLDING_TRACKS];
    Stack outputTrack;
    initializeStack(&outputTrack);

    for (int i = 0; i < k; i++) {
        initializeStack(&holdingTracks[i]);
    }

    int nextExpectedCar = 1;

    if (debug) {
        printf("Initial cars: ");
        for (int i = 0; i < n; i++) {
            printf("%d ", cars[i]);
        }
        printf("\n\n");
    }

    for (int i = 0; i < n; i++) {
        int car = cars[i];

        if (car == nextExpectedCar) {
            push(&outputTrack, car);
            if (debug) printf("Move car %d directly to the output track.\n", car);
            nextExpectedCar++;
        } else {
            bool placed = false;
            for (int j = 0; j < k; j++) {
                if (isEmpty(&holdingTracks[j]) || peek(&holdingTracks[j]) > car) {
                    push(&holdingTracks[j], car);
                    if (debug) printf("Move car %d to holding track H%d.\n", car, j +
1);

                    placed = true;
                    break;
                }
            }
            if (!placed) {
                return false;
            }
        }

        for (int j = 0; j < k; j++) {
            while (!isEmpty(&holdingTracks[j]) && peek(&holdingTracks[j]) ==
nextExpectedCar) {
                push(&outputTrack, pop(&holdingTracks[j]));
                if (debug) printf("Move car %d from holding track H%d to the output
track.\n", outputTrack.cars[outputTrack.top], j + 1);
                nextExpectedCar++;
            }
        }

        if (debug) {
            printf("\nState of holding tracks after processing car %d:\n", car);
            printHoldingTracks(holdingTracks, k);
            printf("State of output track:\n");
            printOutputTrack(&outputTrack, outputTrack.top + 1);
            printf("\n");
        }
    }

    if (debug) {
        printf("\nFinal state of holding tracks:\n");
        printHoldingTracks(holdingTracks, k);
        printf("Final state of output track:\n");
        printOutputTrack(&outputTrack, outputTrack.top + 1);
        printf("\n");
    }
    printOutputTrack(&outputTrack, outputTrack.top + 1);
    return true;
}

int main() {
    int cars[] = {5, 8, 1, 7, 4, 2, 9, 6, 3};
    int n = sizeof(cars) / sizeof(cars[0]);
    int k = 3;
    bool debug = false;


    rearrangeCars(n, k, cars, debug);

    return 0;
}
```

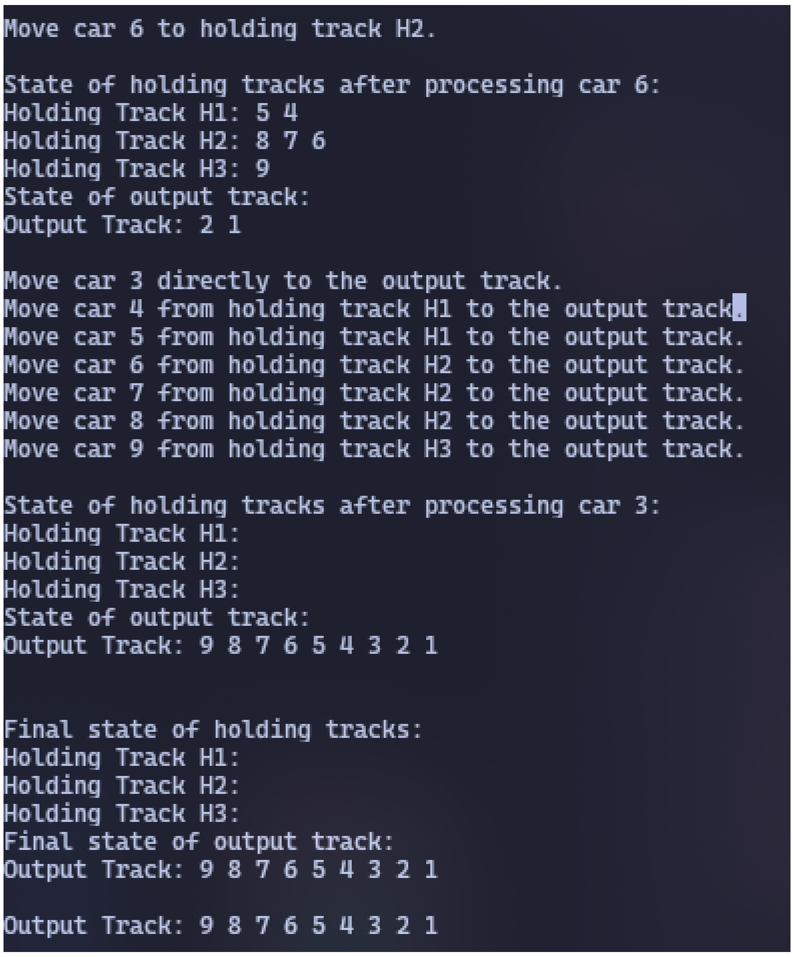
Observations:

1. **Initialization:**
    - The outputTrack is initialized to be empty.
    - Three holding tracks (H1, H2, H3) are initialized to be empty.
2. **Logic:**
    - For each car in the sequence, it places it either directly into the output track (if it's the next expected car) or into one of the holding tracks (if it can be placed there in a non-decreasing order).
    - If a car matches the next expected number, it is moved directly to the output track.
    - If not, the car is placed in one of the holding tracks if the top of the holding track is either empty or larger than the current car.
    - After placing a car, the program checks each holding track to move cars to the output track if they match the next expected number.



```
❯ ./a.out
Output Track: 9 8 7 6 5 4 3 2 1
❯ gcc shuntingYardStack.c
❯ ./a.out
Initial cars: 5 8 1 7 4 2 9 6 3

Move car 5 to holding track H1.

State of holding tracks after processing car 5:
Holding Track H1: 5
Holding Track H2:
Holding Track H3:
State of output track:
Output Track:

Move car 8 to holding track H2.

State of holding tracks after processing car 8:
Holding Track H1: 5
Holding Track H2: 8
Holding Track H3:
State of output track:
Output Track:

Move car 1 directly to the output track.

State of holding tracks after processing car 1:
Holding Track H1: 5
Holding Track H2: 8
Holding Track H3:
State of output track:
Output Track: 1

Move car 7 to holding track H2.

State of holding tracks after processing car 7:
Holding Track H1: 5
Holding Track H2: 8 7
Holding Track H3:
State of output track:
Output Track: 1

Move car 4 to holding track H1.

State of holding tracks after processing car 4:
Holding Track H1: 5 4
Holding Track H2: 8 7
Holding Track H3:
State of output track:
Output Track: 1

Move car 2 directly to the output track.

State of holding tracks after processing car 2:
Holding Track H1: 5 4
Holding Track H2: 8 7
Holding Track H3:
State of output track:
Output Track: 2 1

Move car 9 to holding track H3.

State of holding tracks after processing car 9:
Holding Track H1: 5 4
Holding Track H2: 8 7
Holding Track H3: 9
State of output track:
Output Track: 2 1

Move car 6 to holding track H2.
```

Output:



```
Move car 6 to holding track H2.

State of holding tracks after processing car 6:
Holding Track H1: 5 4
Holding Track H2: 8 7 6
Holding Track H3: 9
State of output track:
Output Track: 2 1

Move car 3 directly to the output track.
Move car 4 from holding track H1 to the output track.
Move car 5 from holding track H1 to the output track.
Move car 6 from holding track H2 to the output track.
Move car 7 from holding track H2 to the output track.
Move car 8 from holding track H2 to the output track.
Move car 9 from holding track H3 to the output track.

State of holding tracks after processing car 3:
Holding Track H1:
Holding Track H2:
Holding Track H3:
State of output track:
Output Track: 9 8 7 6 5 4 3 2 1


Final state of holding tracks:
Holding Track H1:
Holding Track H2:
Holding Track H3:
Final state of output track:
Output Track: 9 8 7 6 5 4 3 2 1

Output Track: 9 8 7 6 5 4 3 2 1
```

APPROACH NO 3: STACKS with an engine simulation:

Program:

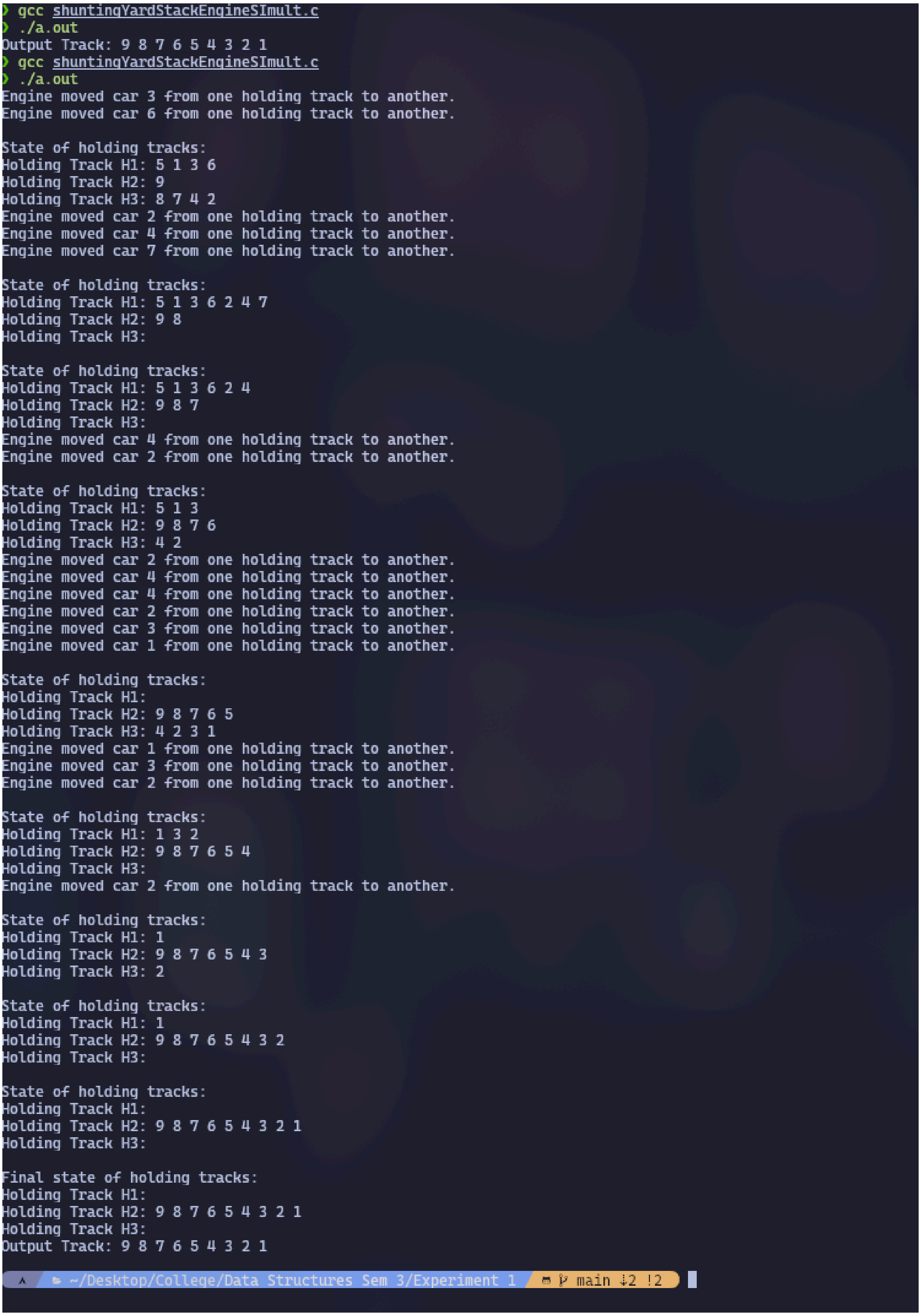
```
#include <stdio.h>
#include <stdbool.h>

#define MAX_HOLDING_TRACKS 100
#define MAX_CARS 100

typedef struct {
   int top;
   int cars[MAX_CARS];
} Stack;

void initializeStack(Stack* s) {
   s->top = -1;
}

bool isEmpty(Stack* s) {
   return s->top == -1;
}

bool isFull(Stack* s) {
   return s->top == MAX_CARS - 1;
}

void push(Stack* s, int car) {
   if (!isFull(s)) {
       s->cars[++(s->top)] = car;
   }
}

int pop(Stack* s) {
   if (!isEmpty(s)) {
       return s->cars[(s->top)--];
   }
   return -1;
}

int peek(Stack* s) {
   if (!isEmpty(s)) {
       return s->cars[s->top];
   }
   return -1;
}

void moveCar(Stack* source, Stack* destination, bool debug) {
   int car = pop(source);
```

```c
    push(destination, car);
    if (debug) {
        printf("Engine moved car %d from one holding track to another.\n", car);
    }
}

void printHoldingTracks(Stack* holdingTracks, int k) {
    for (int i = 0; i < k; i++) {
        printf("Holding Track H%d: ", i + 1);
        for (int j = 0; j <= holdingTracks[i].top; j++) {
            printf("%d ", holdingTracks[i].cars[j]);
        }
        printf("\n");
    }
}

void printOutputTrack(Stack* outputTrack, int size) {
    printf("Output Track: ");
    for (int i = 0; i <= outputTrack->top; i++) {
        printf("%d ", outputTrack->cars[i]);
    }
    printf("\n");
}

bool rearrangeCars(int n, int k, int* cars, bool debug) {
    if (n <= 0 || k < 3 || k > MAX_HOLDING_TRACKS) {
        printf("Invalid input.\n");
        return false;
    }

    Stack holdingTracks[3];
    initializeStack(&holdingTracks[0]);
    initializeStack(&holdingTracks[1]);
    initializeStack(&holdingTracks[2]);

    int nextExpectedCar = n;

    for (int i = 0; i < n; i++) {
        int car = cars[i];

        if (isEmpty(&holdingTracks[0]) || car < peek(&holdingTracks[0])) {
            push(&holdingTracks[0], car);
        } else {
            push(&holdingTracks[2], car);
        }
    }

    while (nextExpectedCar > 0) {
        bool carFound = false;

        while (!isEmpty(&holdingTracks[2]) && !carFound) {
            if (peek(&holdingTracks[2]) == nextExpectedCar) {
                push(&holdingTracks[1], pop(&holdingTracks[2]));
                carFound = true;
            } else {
                moveCar(&holdingTracks[2], &holdingTracks[0], debug);
            }
        }

        while (!isEmpty(&holdingTracks[0]) && !carFound) {
            if (peek(&holdingTracks[0]) == nextExpectedCar) {
                push(&holdingTracks[1], pop(&holdingTracks[0]));
                carFound = true;
            } else {
                moveCar(&holdingTracks[0], &holdingTracks[2], debug);
            }
        }

        if (carFound) {
            nextExpectedCar--;
        } else {
            printf("Failed to find the next expected car: %d\n", nextExpectedCar);
            return false;
        }

        if (debug) {
            printf("\nState of holding tracks:\n");
            printHoldingTracks(holdingTracks, 3);
        }
    }

    if (debug) {
        printf("\nFinal state of holding tracks:\n");
        printHoldingTracks(holdingTracks, 3);
    }

    printOutputTrack(&holdingTracks[1], holdingTracks[1].top + 1);
    return true;
}

int main() {
    int cars[] = {5, 8, 1, 7, 4, 2, 9, 6, 3};
    int n = sizeof(cars) / sizeof(cars[0]);
    int k = 3;
    bool debug = false;
    rearrangeCars(n, k, cars, debug);

    return 0;
}
```

Observations:

1. Initial Setup:
    - The engine initializes three holding tracks (H1, H2, and H3) and an output track (H2).
    - Cars are initially distributed between H1 and H3 based on their values. The smaller cars are placed in H1, and larger cars are pushed into H3.
2. Logic:
    - Finding the Car:
        - The engine aims to find the highest expected car (starting from n and going downwards) and move it to the output track (H2).
        - It first checks H3 for the car. If not found, it moves cars between H3 and H1 to ensure that the car can be found in H3.
    - Moving Between Tracks:
        - From H3 to H2: If the car is found on H3, it is moved to the output track H2.
        - From H3 to H1: If the car is not found on H3, the engine moves cars from H3 to H1 (temporarily) to free up space in H3.
        - From H1 to H3: The cars in H1 are moved to H3 if needed, which helps in searching for the desired car more effectively.
3. Search:
    - The engine continues this process iteratively for each car in descending order. It adjusts the tracks by moving cars between H1, H2, and H3 to ensure that the correct car can be moved to H2.
4. FAilure points:
    - If at any point the engine cannot find the expected car, it reports a failure. This ensures that the rearrangement process halts and reports an issue if the goal is not achievable.

OUTPUT:



```
❯ gcc shuntingYardStackEngineSImult.c
❯ ./a.out
Output Track: 9 8 7 6 5 4 3 2 1
❯ gcc shuntingYardStackEngineSImult.c
❯ ./a.out
Engine moved car 3 from one holding track to another.
Engine moved car 6 from one holding track to another.

State of holding tracks:
Holding Track H1: 5 1 3 6
Holding Track H2: 9
Holding Track H3: 8 7 4 2
Engine moved car 2 from one holding track to another.
Engine moved car 4 from one holding track to another.
Engine moved car 7 from one holding track to another.

State of holding tracks:
Holding Track H1: 5 1 3 6 2 4 7
Holding Track H2: 9 8
Holding Track H3:

State of holding tracks:
Holding Track H1: 5 1 3 6 2 4
Holding Track H2: 9 8 7
Holding Track H3:
Engine moved car 4 from one holding track to another.
Engine moved car 2 from one holding track to another.

State of holding tracks:
Holding Track H1: 5 1 3
Holding Track H2: 9 8 7 6
Holding Track H3: 4 2
Engine moved car 2 from one holding track to another.
Engine moved car 4 from one holding track to another.
Engine moved car 4 from one holding track to another.
Engine moved car 2 from one holding track to another.
Engine moved car 3 from one holding track to another.
Engine moved car 1 from one holding track to another.

State of holding tracks:
Holding Track H1:
Holding Track H2: 9 8 7 6 5
Holding Track H3: 4 2 3 1
Engine moved car 1 from one holding track to another.
Engine moved car 3 from one holding track to another.
Engine moved car 2 from one holding track to another.

State of holding tracks:
Holding Track H1: 1 3 2
Holding Track H2: 9 8 7 6 5 4
Holding Track H3:
Engine moved car 2 from one holding track to another.

State of holding tracks:
Holding Track H1: 1
Holding Track H2: 9 8 7 6 5 4 3
Holding Track H3: 2

State of holding tracks:
Holding Track H1: 1
Holding Track H2: 9 8 7 6 5 4 3 2
Holding Track H3:

State of holding tracks:
Holding Track H1:
Holding Track H2: 9 8 7 6 5 4 3 2 1
Holding Track H3:

Final state of holding tracks:
Holding Track H1:
Holding Track H2: 9 8 7 6 5 4 3 2 1
Holding Track H3:
Output Track: 9 8 7 6 5 4 3 2 1

 ~/Desktop/College/Data Structures Sem 3/Experiment 1   main ⇣2 !2
```