CN Code files-

Exp-4b-

```python
import socket

def dns_lookup():
    print("DNS Lookup")
    link = "a"
    while link != "end":
        link = input("Enter the website name: ")
        if link == "end":
            break
        try:
            host = socket.gethostbyname(link)
            print("IP address of", link, "is", host, "\n")
        except socket.gaierror:
            print("Unable to resolve", link, "\n")
    print("Closed.")

if __name__ == '__main__':
    dns_lookup()
```

Explanation-

1. `import socket`: Imports the `socket` module, which provides access to the BSD socket interface.

2. `def dns_lookup():`: Defines a function named `dns_lookup`.

3. `print("DNS Lookup")`: Prints "DNS Lookup" to the console.

4. `link = "a"`: Initializes the variable `link` with the value "a".

5. `while link != "end":`: Starts a while loop that continues until the value of `link` becomes "end".

6. `link = input("Enter the website name: ")`: Prompts the user to enter a website name and assigns the input value to the variable `link`.

7. `if link == "end":`: Checks if the value of `link` is "end".

8. `break`: If the value of `link` is "end", exits the loop.

9. `try:`: Starts a try block to handle possible errors.

10. `host = socket.gethostbyname(link)`: Calls the `gethostbyname()` function from the `socket` module to retrieve the IP address associated with the entered website name and assigns it to the variable `host`.

11. `print("IP address of", link, "is", host, "\n")`: Prints the IP address of the entered website name to the console.

12. `except socket.gaierror:`: Handles the `gaierror` exception, which occurs if the hostname could not be resolved.

13. `print("Unable to resolve", link, "\n")`: Prints a message indicating that the hostname could not be resolved.

14. `print("Closed.")`: Prints "Closed." to the console.

15. `if __name__ == '__main__':`: Checks if the script is being run directly.

16. `dns_lookup()`: Calls the `dns_lookup()` function if the script is being run directly.

Exp-5a- Go back

```
#include <stdio.h>

int main() {
   int window_size, sent, ack;

   printf("Enter window size: ");
   scanf("%d", &window_size);

   while (1) {
      for (sent = 1; sent <= window_size; sent++) {
         printf("Frame %d has been transmitted.\n", sent);
      }

      printf("Please enter the last Acknowledgement received: ");
      scanf("%d", &ack);

      if (ack == window_size) {
         break;
      } else {
         sent = ack + 1;
      }
   }

   return 0;
}
```

Explanation-

1. **Include header:**
   - `#include <stdio.h>`: This line includes the standard input/output library, which provides functions like `printf` for printing to the console and `scanf` for reading input from the user.
2. **Main function:**
   - `int main()`: This is the main function where the program execution begins.
3. **Declare variables:**
   - `int window_size, sent, ack`: These variables are used to store the window size, the sequence number of the next frame to be transmitted, and the acknowledgement number received from the receiver, respectively.
4. **Get window size:**
   - `printf("Enter window size: ");`: This line prompts the user to enter the window size.
   - `scanf("%d", &window_size);`: This line reads the window size entered by the user and stores it in the `window_size` variable.
5. **Transmission loop:**
   - `while (1)`: This is an infinite loop that continues until it's broken out of by a condition inside the loop.
   - `for (sent = 1; sent <= window_size; sent++)`: This loop iterates `window_size` times, transmitting one frame in each iteration.
     - `printf("Frame %d has been transmitted.\n", sent);`: This line prints a message indicating that a frame with sequence number `sent` has been transmitted.
6. **Receive acknowledgement:**
   - `printf("Please enter the last Acknowledgement received: ");`: This line prompts the user to enter the acknowledgement number received from the receiver.
   - `scanf("%d", &ack);`: This line reads the acknowledgement number entered by the user and stores it in the `ack` variable.
7. **Check for complete transmission:**
   - `if (ack == window_size)`: This condition checks if the received acknowledgement number is equal to the window size. If it is, it means all the frames within the window have been acknowledged, and the loop can be broken.
8. **Update sent based on acknowledgement:**
   - `else { sent = ack + 1; }`: If all frames haven't been acknowledged, this part updates the `sent` variable to the next expected acknowledgement number (one more than the received `ack`). This ensures retransmission of the unacknowledged frames in the next iteration of the loop.
9. **Exit loop:**
   - The loop continues iterating until all frames within the window are acknowledged.
10. **End of program:**

- `return 0;`: This line indicates successful program termination.

Exp-5b- Selective Repeat

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

#define WINDOW_SIZE 4

typedef struct {
    int frame_no;
    bool acked;
} Frame;

void send_frames(Frame frames[], int start, int end) {
    for (int i = start; i <= end; i++) {
        printf("Sending Frame %d\n", frames[i].frame_no);
    }
}

void receive_ack(Frame frames[], int acked_frame) {
    frames[acked_frame].acked = true;
    printf("Received ACK for Frame %d\n", frames[acked_frame].frame_no);
}

int main() {
    Frame frames[WINDOW_SIZE];

    // Initialize frames
    for (int i = 0; i < WINDOW_SIZE; i++) {
        frames[i].frame_no = i + 1;
        frames[i].acked = false;
    }

    int next_to_send = 0;
    int last_acked = -1;

    while (true) {
        // Send frames
        int frames_to_send = WINDOW_SIZE - (next_to_send - last_acked);
        if (frames_to_send > 0) {
            int end = next_to_send + frames_to_send - 1;
            if (end >= WINDOW_SIZE) {
                end = WINDOW_SIZE - 1;
            }
            send_frames(frames, next_to_send, end);
            next_to_send = end + 1;
```

```
        }

        // Check for ACKs
        int ack;
        printf("Enter the ACK received: ");
        scanf("%d", &ack);

        if (ack >= 0 && ack < WINDOW_SIZE) {
            receive_ack(frames, ack);
            last_acked = ack;
        } else {
            printf("Invalid ACK\n");
        }

        // Check if all frames are acknowledged
        bool all_acked = true;
        for (int i = 0; i < WINDOW_SIZE; i++) {
            if (!frames[i].acked) {
                all_acked = false;
                break;
            }
        }

        if (all_acked) {
            break;
        }
    }

    printf("All frames acknowledged successfully.\n");

    return 0;
}
```

Explanation-

1. **Include headers:**
    - `#include <stdio.h>`: This line includes the standard input/output library, which provides functions like `printf` for printing to the console and `scanf` for reading input from the user.
    - `#include <stdbool.h>`: This line includes the boolean library, which provides the `bool` data type for true/false values.
2. **Define window size:**
    - `#define WINDOW_SIZE 4`: This line defines a preprocessor macro named `WINDOW_SIZE` with a value of 4. This value determines the number of frames that can be transmitted in a single window.
3. **Frame structure:**

- `typedef struct { ... } Frame;`: This defines a structure named `Frame` to store information about each frame.
  - `int frame_no`: This member variable stores the sequence number of the frame.
  - `bool acked`: This member variable is a flag that indicates whether the frame has been acknowledged by the receiver.

4. **Send frames function:**
   - `void send_frames(Frame frames[], int start, int end)`: This function takes an array of frames, a starting index, and an ending index as arguments. It iterates over the specified range in the `frames` array and prints a message indicating that each frame is being sent.
5. **Receive acknowledgement function:**
   - `void receive_ack(Frame frames[], int acked_frame)`: This function takes an array of frames and the index of the acknowledged frame as arguments. It marks the corresponding frame in the `frames` array as acknowledged by setting the `acked` flag to `true`. It also prints a message indicating that an acknowledgement has been received for a specific frame.
6. **Main function:**
   - `int main()`: This is the main function where the program execution begins.
7. **Initialize frames:**
   - `Frame frames[WINDOW_SIZE];`: This line declares an array of `Frame` structures

Exp-6a-Leaky Bucket
// Name: Aaryan Thipse

```
#include <stdio.h>

int main() {
   int incoming, outgoing, bucket_size, n, store = 0;

   printf("Enter bucket size, outgoing rate, and number of inputs: ");
   scanf("%d %d %d", &bucket_size, &outgoing, &n);

   while (n > 0) {
      printf("Enter the incoming packet size: ");
      scanf("%d", &incoming);
      printf("Incoming packet size %d\n", incoming);

      if (incoming <= (bucket_size - store)) {
         store += incoming;
         printf("Bucket buffer size %d out of %d\n", store, bucket_size);
      } else {
         printf("Dropped %d packets\n", incoming - (bucket_size - store));
         printf("Bucket buffer size %d out of %d\n", store, bucket_size);
```

```
            store = bucket_size; // The bucket is full, no more packets can be
added.
        }

        store = store - outgoing;
        if (store < 0) {
            store = 0; // Ensure the bucket doesn't go negative.
        }
        printf("After outgoing %d packets left out of %d in buffer\n", store,
bucket_size);
        n--;
    }

    return 0;
}
```

Explanation-

This code simulates a leaky bucket model for network traffic management. Here's a breakdown of how it works:

1. Variables:

- `bucket_size`: This variable stores the maximum capacity of the bucket, representing the buffer size for incoming packets.
- `outgoing`: This variable stores the outgoing rate, representing the number of packets leaving the bucket per unit time.
- `n`: This variable stores the total number of incoming packets to be processed.
- `store`: This variable keeps track of the current number of packets stored in the bucket.
- `incoming`: This variable is used temporarily to store the size of each incoming packet during the loop.

2. Input:

- The code prompts the user to enter three values:
    - The bucket size (buffer capacity)
    - The outgoing rate (packets leaving per unit time)
    - The total number of incoming packets to be processed (`n`)

3. Looping through incoming packets:

- The `while` loop continues as long as there are still packets to process (`n` is greater than 0).

- Inside the loop, the user is prompted to enter the size of the incoming packet, which is stored in the `incoming` variable.
- The code checks if the incoming packet size is less than or equal to the available space in the bucket (`bucket_size - store`).
    - If there's enough space:
        - The `incoming` packet size is added to the `store`, effectively placing the packet in the bucket.
        - The updated `store` value (current number of packets in the bucket) is printed along with the bucket size.
    - If there's not enough space:
        - The number of dropped packets is calculated as the difference between the incoming packet size and the available space.
        - A message is printed indicating the number of dropped packets.
        - The `store` value is set to the maximum capacity (`bucket_size`) to simulate the bucket being full and no more packets being added.
        - The current bucket state (full with `bucket_size` packets) is printed.

4. Simulating outgoing packets:

- After processing the incoming packet, the code simulates outgoing packets by subtracting the `outgoing` rate from the `store` value.
- If the resulting value is negative (more packets are removed than present), the `store` is set to 0 to ensure the bucket doesn't go into a negative state.
- The updated `store` value (number of packets remaining after outgoing packets) is printed along with the bucket size.

5. Decrementing counter and exiting:

- The `n` counter is decremented by 1 to indicate one less packet needs to be processed.
- The loop continues until all incoming packets (`n`) have been processed.

6. Exit:

- After processing all packets, the code exits by returning 0.

In summary, this code simulates a network scenario where incoming packets arrive at a buffer (bucket) with a limited capacity. Packets are dropped if they arrive when the buffer is full. The code also considers an outgoing rate, simulating packets leaving the buffer over time.

Exp-7a-Distance vector Routing

```
#include <stdio.h>

struct node {
   unsigned dist[20];
   unsigned from[20];
} rt[10];

int main() {
   int dmat[20][20];
   int n, i, j, k, count = 0;

   printf("\nEnter the number of nodes : ");
   scanf("%d", &n);

   printf("\nEnter the cost matrix :\n");
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++) {
         scanf("%d", &dmat[i][j]);
         dmat[i][i] = 0;
         rt[i].dist[j] = dmat[i][j];
         rt[i].from[j] = j;
      }

   do {
      count = 0;
      for (i = 0; i < n; i++)
         for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
               if (rt[i].dist[j] > dmat[i][k] + rt[k].dist[j]) {
                  rt[i].dist[j] = rt[i].dist[k] + rt[k].dist[j];
                  rt[i].from[j] = k;
                  count++;
               }
   } while (count != 0);

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

```
        printf("\n\nState value for router %d is \n", i + 1);
        for (j = 0; j < n; j++) {
            printf("\t\nnode %d via %d Distance%d", j + 1, rt[i].from[j] + 1,
rt[i].dist[j]);
        }
    }
    printf("\n\n");
}
```

Explanation-
Sure, let's break down the code:

1. `#include <stdio.h>`: This line includes the standard input/output library header file, which provides functions like `printf()` and `scanf()`.

2. `struct node { unsigned dist[20]; unsigned from[20]; } rt[10];`: This defines a structure named `node`, which contains two arrays: `dist` and `from`, each with a size of 20. It also declares an array `rt` of 10 elements, each of type `node`.

3. `int main() { ... }`: This is the main function where the execution of the program begins.

4. `int dmat[20][20];`: Declares a 2D array `dmat` of size 20x20 to store the cost matrix.

5. `int n, i, j, k, count = 0;`: Declares integer variables `n`, `i`, `j`, `k`, and `count`, where `n` represents the number of nodes, and the others are loop counters and a counter for the number of updates in the distance vectors.

6. `printf("\nEnter the number of nodes : "); scanf("%d", &n);`: Prompts the user to enter the number of nodes and reads the input value into the variable `n`.

7. Nested loops to input the cost matrix:
   - `for (i = 0; i < n; i++)`: Iterates over each row of the matrix.
   - `for (j = 0; j < n; j++)`: Iterates over each column of the matrix.
   - `scanf("%d", &dmat[i][j]);`: Reads the cost of the edge from node `i` to node `j`.
   - `dmat[i][i] = 0;`: Sets the diagonal elements of the matrix to 0.
   - `rt[i].dist[j] = dmat[i][j];`: Initializes the distance vector of router `i`.
   - `rt[i].from[j] = j;`: Initializes the next-hop vector of router `i`.

8. `do { ... } while (count != 0);`: A do-while loop to perform the distance vector update until no more changes occur.

- Nested loops iterate over each node and each destination node to update the distance vector based on the Bellman-Ford algorithm.
- If a shorter path to a destination node is found through another node, the distance and next-hop vectors are updated, and `count` is incremented.

9. Output the routing table:
- `for (i = 0; i < n; i++)`: Iterates over each router.
- `printf("\n\nState value for router %d is \n", i + 1);`: Prints the router number.
- Nested loop to print the distance and next-hop vectors for each destination node.
- `printf("\t\nnode %d via %d Distance%d", j + 1, rt[i].from[j] + 1, rt[i].dist[j]);`: Prints the destination node number, next-hop node number, and distance.

10. `printf("\n\n");`: Prints two newlines for better formatting.

Exp-7b- Implementation of link state routing algorithm.

```
#include <stdio.h>

#define INFINITY 9999
#define MAX 10

void dijkstra(int G[MAX][MAX], int n, int startnode);

int main() {
    int G[MAX][MAX], i, j, n, u;

    printf("Enter no. of vertices:");
    scanf("%d", &n);

    printf("\nEnter the adjacency matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &G[i][j]);

    printf("\nEnter the starting node:");
    scanf("%d", &u);

    dijkstra(G, n, u);

    return 0;
}
```

```
void dijkstra(int G[MAX][MAX], int n, int startnode) {
    int cost[MAX][MAX], distance[MAX], pred[MAX];
    int visited[MAX], count, mindistance, nextnode, i, j;

    // pred[] stores the predecessor of each node
    // count gives the number of nodes seen so far
    // create the cost matrix
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            cost[i][j] = (G[i][j] == 0) ? INFINITY : G[i][j];

    // initialize pred[], distance[], and visited[]
    for (i = 0; i < n; i++) {
        distance[i] = cost[startnode][i];
        pred[i] = startnode;
        visited[i] = 0;
    }

    distance[startnode] = 0;
    visited[startnode] = 1;
    count = 1;

    while (count < n - 1) {
        mindistance = INFINITY;
        // nextnode gives the node at minimum distance
        for (i = 0; i < n; i++)
            if (distance[i] < mindistance && !visited[i]) {
                mindistance = distance[i];
                nextnode = i;
            }

        // check if a better path exists through nextnode
        visited[nextnode] = 1;
        for (i = 0; i < n; i++)
            if (!visited[i] && mindistance + cost[nextnode][i] < distance[i]) {
                distance[i] = mindistance + cost[nextnode][i];
                pred[i] = nextnode;
            }
        count++;
    }

    // print the path and distance of each node
    for (i = 0; i < n; i++)
        if (i != startnode) {
            printf("\nDistance of node%d=%d", i, distance[i]);
            printf("\nPath=%d", i);
```

```
            j = i;
            do {
                j = pred[j];
                printf("<-%d", j);
            } while (j != startnode);
        }
}
```

Explanation-

The code implements Dijkstra's algorithm to find the shortest paths from a given starting node to all other nodes in a graph.

- dijkstra function takes the adjacency matrix G, the number of vertices n, and the starting node startnode.
- It initializes arrays cost, distance, pred, and visited for computing shortest paths.
- It creates the cost matrix based on the adjacency matrix, initializing unreachable nodes with INFINITY.
- Inside the while loop, it iteratively selects the node with the shortest distance and updates the distances to its neighbors.
- After computing the shortest paths, it prints the distance and path from the starting node to each other node.