<u>Project assignment 5</u>
<u>Team 08</u>
Authors: [Prasanta Bhattacharjee, Md. Asif Talukdar, Emam Hossain]

To design a memory-efficient data structure meeting the specified requirements, we can use a compact representation that minimizes the memory footprint. Below is a sample design using arrays to achieve the desired functionality:

(a) Design of the Data Structure:

We are team 8, so our Node name is also 8. In this memory optimization technique, we will use the **uint8_t** data type to store my 1 hop and 2 hop neighbor information. (**uint8_t means 1 byte of unsigned integer.)** As we have a maximum of 24 neighbors 1-hop neighbors, we will have an array named **"N"** of **uint8_t** with the size 24. Here, the 1st bit of each **uint8_t** denotes whether that node is in our 1-hop or not (1 means in and 0 means out). The next 2 bits represent the **duration** of the 1-hop neighbor, which means 2^2 = 4, from 1-4 seconds we can represent with those bits, for the rest of the 4 bits, that means 2^5 = 32, from 1-32 neighbors we can represent with those bits using the index number. We are assuming the nodes are updating every 4 seconds, and the neighbor can have a Node name from 1 to 24, inclusive.

Uint8_t   N[24];
Suppose we take N[0], the first value, and N[0] is '11101111', we represent this value below with details as it is for N[0], that means the information is for the 0+1= 1 named neighbor Node:

Node 1 in 1 Hop (Yes =1, No = 0)	Duration of connection for Node 1 (til 4 seconds)	Number of 2 Hop neighbors from Node 1 (01111 is equal to 15, which means 15+1=16, 2-Hop neighbors are there)
1	11	01111

So, there will be 24 of these from 0 index to 23 index.

Now To store 24 2-Hop neighbors, we create 3 more data structures named **TwoHop1, TwoHop2 and TwoHop3** respectively as below:

Uint8_t TwoHop1[24]; This represents whether 1 to 8 named Nodes are in the 2-Hop list.
Uint8_t TwoHop2[24]; This represents whether 9 to 16 named Nodes are in the 2-Hop list.
Uint8_t TwoHop3[24]; This represents whether 17 to 24 named Nodes are in the 2-Hop list.

We continue to analyze for the 1 named 1-hop Node, suppose we have the value as below:
TwoHop1[0] = {01010111} we go by the index+1 we can see 2, 4, 6, 7, 8 Nodes are connected as 2-Hop neighbors.
TwoHop2[0] = {01110000} we go by the 8+index+1 we can see 10, 11, 12 Nodes are connected as 2-Hop neighbors.
TwoHop3[0] = {00000111} we go by the 16+index+1 we can see 22, 23, 24 Nodes are connected as 2-Hop neighbors.

Collectively these 3 arrays form the 2-Hop neighbor list for the corresponding 1-hop neighbors.

(b) Storage Analysis:

We had a limitation of around 96 bytes for the memory allocation. In our storage algorithm, we used 4 **Uint8_t** arrays of 24 size (**N[24], TwoHop1[24], TwoHop2[24] and TwoHop3[24]**), where **Uint8_t** takes 1 byte per initialization.

Therefore, total space taken is = 24 + 24 + 24 +24 = 96 bytes. As a result, it satisfies the requirement.

(c) Operations supported by our structure:

We have the following data structure:
Uint8_t N[24];
Uint8_t TwoHop1[24];
Uint8_t TwoHop2[24];
Uint8_t TwoHop3[24];

To insert 1st and 2nd Hop neighbors, suppose we receive a packet (char buffer[32]) where the 1st byte is packet type (05), 2nd byte team Number, 3rd is 2-Hop neighbor count and the rest of the bytes are 2-Hop neighbors.

Suppose we get **Team 9** with 5, 2-Hop neighbors such as 2, 3, 5, 16, 21. Therefore,

Int name = buffer[1] which is 9;
Int number = buffer[2] which is 5;

So, we update the N[9-1] = 1-00-00101, the first bit is 1 because team-9 is 1-Hop neighbor, then 00 is the initial duration and the rest of the bits 00101 represent 5 which is 2 hop neighbor count. So, N[9-1] is updated (index starts from 0).

Now, we loop through the 2 hop neighbors if the value is <= 8 it is stored **TwoHop1** here 2, 3, 5 is <=8 so updated **TwoHop1[9-1] = 01101000**, corresponding index is converted to 1.

If the 8<value<=16 it is stored in **TwoHop2** here, 16 satisfies so **TwoHop2[9-1] = 00000001**. We need to subtract 8+1 from 16 to get the corresponding index.

Similarly, for the last value 21, **TwoHop3[9-1] = 00001000** is stored with respect to the index. We need to subtract 16+1 from 21 to get the corresponding index.

When we update or fetch the 2 Hop neighbors, we add 8 and 16 to the index in the for loop depending on the value we are trying to access. We do the same thing when removing any neighbors (making 1 to 0).