```
/**
 * Representation of Graph Data Structure.
 * Adjacency List is being implemented here with C code.
 * (Although I used cpp extension to include library convenience.)
 *
 * ----------------------------------------------------------------
 *  [*] represents arrow head.
 *  [label] defines the label name for each vertex.
 *
 *       ┌-----------------------------------------------┐
 *       *                                               |
 *  |---[Ram]----hasFriend----*[Shyam]                   |
 *  |    |   *                 *                         |
 *  |    |     \               |                     hasBrother
 *  |    |       \          hasSonInLaw                  |
 *  |  hasFriend    \          |                         |
 *  |    |         isSonOf     |                         |
 *  |    |             \       |                         |
 *  |    *              \      |                         *
 *  |   [Hari]          [Dasharath]--------isSonOf----*[Laxman]
 *  |                        *                           |
 *  |------isFatherOf--------|------ isFatherOf----------|
 *
 * ---------------------------------------------------------------
 * Textual Description.
 * ---------------------------------------------------------------
 * 1. [Ram] hasFriend [Shyam].
 * 2. [Ram] hasFriend [Hari].
 * 3. [Ram] hasBrother [Laxman].
 * 4. [Laxman] hasBrother [Ram].
 * 6. [Dasharath] hasSonInLaw [Shyam].
 * 7. [Ram] isSonOf [Dasharath].
 * 8. [Laxman] isSonOf [Dasharath].
 * 9. [Dasharath] isFatherOf [Ram].
 * 10.[Dasharath] isFatherOf [Laxman]
 *----------------------------------------------------------------
 *
 * By: Santa Basnet
 * Everest Engineering College.
 * Date: 2022-01-20
 */

/**
 * Required Header Files.
 */
#include <cstdio>
#include <cstdlib>
#include <cstring>

/**
 * Default initialization for the Cost.
 */
const int DEFAULT_COST = 1;
```

```c
/**
 * Boolean TRUE.
 */
const int TRUE = 1;

/**
 * Boolean FALSE.
 */
const int FALSE = 0;

/**
 * Edge Representation.
 *
 * label: a string label name given the edge.
 * cost: a cost value associated for weighted graph.
 * to: a Vertex pointer to which it is pointing.
 * next: a pointer to the next edge in the adjacency list.
 */
struct Edge {
    char label[32];
    int cost;
    struct Vertex *to;
    struct Edge *next;
};

/**
 * Vertex Representation.
 *
 * name: a label name given the vertex, a character string.
 * adjacencyList: an adjacency list representation, which is associated with
 * the given Vertex.
 */
struct Vertex {
    char name[32];
    struct Edge *adjacencyList;
    struct Vertex *next;
};

/**
 * Final Graph Representation.
 * It has multiple Vertices, and represented by the Dynamic
 * list of vertices, a single variable graph is used to
 * access it.
 *
 * root: Denotes the starting pointer for the Graph, in the adjacency list.
 * vertexCount: Denotes the number of vertices available in the Graph.
 */
struct Graph {
    struct Vertex *root;
    int vertexCount;
};
```

```
/**
 * Initialize a graph variable.
 */
struct Graph *familyGraph = nullptr;

/**
 * Display a Vertex.
 */
void printVertex(struct Vertex *vertex) {
    printf("[%s]", vertex->name);
}

/**
 * Display an Edge.
 */
void printEdge(struct Edge *edge) {
    printf(" ==>|(%s, %d)|==>[%s], ", edge->label, edge->cost, edge->to->name);
}

/**
 * List out all the vertices in the Graph.
 */
void printVertexList() {
    if (familyGraph->root == nullptr) {
        printf("\nNo Vertices added.");
        return;
    }
    struct Vertex *traverse = familyGraph->root;
    int vertexCount = 1;
    while (traverse != nullptr) {
        printf("[(%d, %s)] -> ", vertexCount++, traverse->name);
        traverse = traverse->next;
    }
}

/**
 * Display adjacency list.
 */
void displayAdjacencyList(struct Vertex *vTraverse) {
    printVertex(vTraverse);
    struct Edge *eTraverse = vTraverse->adjacencyList;
    while (eTraverse != nullptr) {
        printEdge(eTraverse);
        eTraverse = eTraverse->next;
    }
}

/**
 * Display the entire graph.
 */
void printGraph() {
    if (familyGraph->root == nullptr) {
        printf("\nEmpty Graph.");
        return;
    }
    /**
     * Iterate over all the vertices.
     */
    struct Vertex *vTraverse = familyGraph->root;
    while (vTraverse != nullptr) {
        displayAdjacencyList(vTraverse);
        vTraverse = vTraverse->next;
        printf("\n");
    }
}

/**
 * Get vertex by name.
 */
struct Vertex *vertexByName(char *vertexName) {
    if (familyGraph == nullptr) return nullptr;
    struct Vertex *traverse = familyGraph->root;
    while (traverse != nullptr) {
        if (strcmp(traverse->name, vertexName) == 0) {
            return traverse;
        }
        traverse = traverse->next;
    }
    return nullptr;
}

/**
 * Returns the index of vertex.
 */
int vertexIndex(char *name) {
    int index = 0;
    struct Vertex *traverse = familyGraph->root;
    while (traverse != nullptr) {
        if (strcmp(traverse->name, name) == 0) {
            return index;
        }
        index++;
        traverse = traverse->next;
    }
    return -1;
}

/**
 * Check if any Vertex of the name is present or not.
 */
int isVertexPresent(char *vertexName) {
    return vertexByName(vertexName) == nullptr ? FALSE : TRUE;
}


/**
```

```
 * Check if any Vertex of the given name is already created or not.
 */
int isVertexPresent(struct Vertex *givenVertex) {
    return isVertexPresent(givenVertex->name);
}


/**
 * Create a Vertex Operation.
 * It also verifies if the vertex is already created or not.
 * If so, it returns the vertex pointer of already created vertex.
 *
 * @param name, a label name given the Vertex. It uniquely identifies
 * the given vertex.
 *
 * @return vertexPointer, it returns the vertex of the newly created or
 * already exist in the Graph.
 */
struct Vertex *createVertex(char *name) {
    struct Vertex *newVertex = (struct Vertex *) malloc(sizeof(struct Vertex));
    strcpy(newVertex->name, name);
    newVertex->adjacencyList = nullptr;
    newVertex->next = nullptr;
    return newVertex;
}

/**
 * Create Edge Operation.
 * @param label, label assigned to the given edge.
 * @param cost, cost of the edge, default 1 for all cost.
 * @param to, connected vertex through the current edge.
 *
 * @return newlyCreatedEdge
 */
struct Edge *createEdge(char *label, int cost, struct Vertex *to) {
    struct Edge *newEdge = (struct Edge *) malloc(sizeof(struct Edge));
    strcpy(newEdge->label, label);
    newEdge->cost = cost;
    newEdge->to = to;
    return newEdge;
}

/**
 * Perform add vertex operation.
 */
void addVertex(struct Vertex *vertex) {
    if (familyGraph == nullptr) {
        familyGraph = (struct Graph *) (malloc(sizeof(struct Graph)));
        familyGraph->root = vertex;
        familyGraph->vertexCount = 1;
    } else if (!isVertexPresent(vertex)) {
        struct Vertex *traverse = familyGraph->root;
        while (traverse->next != nullptr) traverse = traverse->next;
        traverse->next = vertex;
        familyGraph->vertexCount++;
    }
}

/**
 * Add an edge to the given vertex.
 */
void addEdge(struct Vertex *source, struct Edge *edge) {
    if (source->adjacencyList == nullptr) source->adjacencyList = edge;
    else {
        struct Edge *traverse = source->adjacencyList;
        while (traverse->next != nullptr) traverse = traverse->next;
        traverse->next = edge;
    }
}


/**
 * Initialize a root vertex and the whole graph.
 */
void initFamilyGraph() {
    struct Vertex *ram = createVertex("Ram");
    struct Vertex *shyam = createVertex("Shyam");
    struct Vertex *hari = createVertex("Hari");
    struct Vertex *laxman = createVertex("Laxman");
    struct Vertex *dasharath = createVertex("Dasharath");

    struct Edge *hasFriendShyam = createEdge("hasFriend", DEFAULT_COST, shyam);
    struct Edge *hasFriendHari = createEdge("hasFriend", DEFAULT_COST, hari);
    struct Edge *hasBrotherLaxman = createEdge("hasBrother", DEFAULT_COST,
laxman);
    struct Edge *hasBrotherRam = createEdge("hasBrother", DEFAULT_COST, ram);
    struct Edge *hasSonInLawShyam = createEdge("hasSonInLaw", DEFAULT_COST,
shyam);
    struct Edge *isSonOfDasharath = createEdge("isSonOf", DEFAULT_COST,
dasharath);
    struct Edge *isFatherOfRam = createEdge("isFatherOf", DEFAULT_COST, ram);
    struct Edge *isFatherOfLaxman = createEdge("isFatherOf", DEFAULT_COST,
laxman);

    addVertex(ram);
    addVertex(laxman);
    addVertex(shyam);
    addVertex(hari);
    addVertex(dasharath);

    addEdge(ram, hasFriendShyam);
    addEdge(ram, hasFriendHari);
    addEdge(ram, hasBrotherLaxman);
    addEdge(laxman, hasBrotherRam);
    addEdge(dasharath, hasSonInLawShyam);
    addEdge(ram, isSonOfDasharath);
    addEdge(laxman, isSonOfDasharath);
    addEdge(dasharath, isFatherOfRam);
    addEdge(dasharath, isFatherOfLaxman);
}

/**
 * Find if the edge exists between two vertices or not.
 */
int isRelationExists(char *sourceName, char *edgeLabel, char *destinationName) {
    struct Vertex *source = vertexByName(sourceName);
    if (source == nullptr) return FALSE;
    struct Edge *eTraverse = source->adjacencyList;
    while (eTraverse != nullptr) {
        if (strcmp(eTraverse->label, edgeLabel) == 0 && strcmp(eTraverse->to-
>name, destinationName) == 0) {
            return TRUE;
        }
        eTraverse = eTraverse->next;
    }
    return FALSE;
}

/**
 * Perform Graph Cleanup to make make memory free.
 */
void cleanupGraph() {
    if (familyGraph->root == nullptr) return;
    /**
     * Iterate over all the vertices and clean up them.
     */
    struct Vertex *vTraverse = familyGraph->root;
    while (vTraverse != nullptr) {
        struct Vertex *tVertex = vTraverse;
        vTraverse = vTraverse->next;
        free(tVertex);
    }
}

/**
 * Display vertex existence of given name.
 */
void displayVertexExistence() {
    char vName[16] = "Dasharath";
    int vResult = isVertexPresent(vName);
    if (vResult) printf("\n2. Yes, the vertex \"%s\" is available.", vName);
    else printf("\n2. No, the vertex \"%s\" is not available.", vName);
}

/**
 * Display the relation existence between two vertices and the relation by name.
 */
void displayRelation() {
    char sVertex[16] = "Ram";
    char dVertex[16] = "Dasharath";
    char relation[16] = "isSonOf";
    int rResult = isRelationExists(sVertex, relation, dVertex);
    if (rResult)
        printf("\n3. Yes, the relation \"%s\" is available between \"%s\" and
\"%s\".", relation, sVertex, dVertex);
    else printf("\n3. No, the relation \"%s\" is not available between \"%s\"
and \"%s\".", relation, sVertex, dVertex);
}

/**
 * Display the all the available relations of the given vertex.
 */
void displayVertexRelations() {
    char vertexName[16] = "Dasharath";
    struct Vertex *sVertex = vertexByName(vertexName);
    printf("\n4. All the available relations of \"%s\": [", vertexName);
    struct Edge *eTraverse = sVertex->adjacencyList;
    while (eTraverse != nullptr) {
        printf("%s, ", eTraverse->label);
        eTraverse = eTraverse->next;
    }
    printf("]\n");
}

void displayPath(struct Vertex *root, char *sourceName, char *targetName, char
*path, int *visitedVertex) {
    char newPath[256];
    strcpy(newPath, path);
    if (strlen(path) > 0) strcat(newPath, "=>");
    strcat(newPath, sourceName);
    struct Edge *relation = root->adjacencyList;
    while (relation != nullptr) {
        if (strcmp(relation->to->name, targetName) == 0) {
            printf("\n\tFound: [%s==>%s]", newPath, targetName);
        } else {
            int index = vertexIndex(relation->to->name);
            if (visitedVertex[index] != TRUE) {
                visitedVertex[index] = TRUE;
                displayPath(relation->to, relation->to->name, targetName,
newPath, visitedVertex);
            }
        }
        relation = relation->next;
    }
}

/**
 * Display all the available paths from the given source to the destination
vertex.
 */
void displayAllPaths() {
    int visitedVertex[familyGraph->vertexCount];
```

```c
    for (int i = 0; i < familyGraph->vertexCount; i++) visitedVertex[i] = FALSE;
    char sourceName[16] = "Laxman";
    char targetName[16] = "Shyam";
    printf("\n5. All the available paths: ");
    if (vertexIndex(sourceName) == -1) {
        printf("\n\tSource vertex is not available.");
        return;
    }
    if (vertexIndex(targetName) == -1) {
        printf("\n\tTarget vertex is not available.");
        return;
    }
    visitedVertex[vertexIndex(sourceName)] = TRUE;
    displayPath(vertexByName(sourceName), sourceName, targetName, "",
visitedVertex);
}

/**
 * Graph Representation Main function starts here.
 * @return 0 for successful operations.
 */
int main() {
    printf("\nHello from EEC Graph: ");
    printf("\n--------------------------- ");

    /**
     * Initialization.
     */
    initFamilyGraph();

    /**
     * Display Initialized Data.
     */
    printf("\n1.Display the Graph:");
    printf("\n--------------------------- \n");
    printGraph();

    /**
     * Question 2. Check if the vertex is present or not.
     */
    displayVertexExistence();

    /**
     * Question 3. Is Ram son of Dasharath ?
     */
    displayRelation();

    /**
     * Question 4. How many relations are exists for a given vertex ?
     */
    displayVertexRelations();

    /**
```

```
     * Question 5. What are the available relations between two vertices?
     */
    displayAllPaths();

    cleanupGraph();
    printf("\n");

    return 0;
}

Output
-------------------------------------------------------------------------------

Hello from EEC Graph:
---------------------------
1.Display the Graph:
---------------------------
[Ram] ==>|(hasFriend, 1)|==>[Shyam],  ==>|(hasFriend, 1)|==>[Hari],  ==>|
(hasBrother, 1)|==>[Laxman],  ==>|(isSonOf, 1)|==>[Dasharath],
[Laxman] ==>|(hasBrother, 1)|==>[Ram],  ==>|(isSonOf, 1)|==>[Dasharath],
[Shyam]
[Hari]
[Dasharath] ==>|(hasSonInLaw, 1)|==>[Shyam],  ==>|(isFatherOf, 1)|==>[Ram],
==>|(isFatherOf, 1)|==>[Laxman],

2. Yes, the vertex "Dasharath" is available.
3. Yes, the relation "isSonOf" is available between "Ram" and "Dasharath".
4. All the available relations of "Dasharath": [hasSonInLaw, isFatherOf,
isFatherOf, ]

5. All the available paths:
      Found: [Laxman=>Ram==>Shyam]
      Found: [Laxman=>Ram=>Dasharath==>Shyam]
```