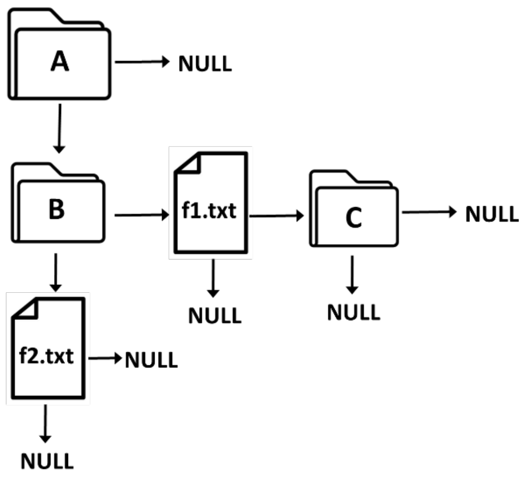
Question 1.

The data structure below represents a file system component (which can be a file or a directory) and contains the following information:

- *name*: A string representing the name of the file/directory.
- *isFile*: A 0/1 value indicating whether the node represents a directory or a file.
- *size*: In the case of files, this field represents the file size in bytes. For directories, this value should be 0 as we assume that the size of a directory is determined by computing the total size of all the files contained in it and its sub-directories.
- *firstChild*: A pointer to the first child node, i.e. the first file or sub-directory within the directory represented by the node. For nodes representing files, this field should be NULL.
- *nextSibling*: A pointer to the next file or sub-directory within the same parent directory.

```
typedef struct file_or_dir_node{
      char name[20];
      int isFile;
      int size;
      struct file_or_dir_node *firstChild;
      struct file_or_dir_node *nextSibling;
}Node;
```

The following diagram represents a file system hierarchy, with downward arrows indicating the *firstChild* pointer and right arrows denoting the *nextSibling* connection. Directory A has two sub-directories (B and C) and a file named f1.txt. In addition, B has one file named f2.txt and C is empty.



a) Complete the implementation of function `createNode` to create a new file/directory node based on the values provided as inputs. Note that the newly created component does not have any children or siblings.

```
Node *createNode(char name[20], int isFile, int size){
   //complete the implementation









}
```

b) Complete the implementation of function `addChild` to add `new_node` as a child of `parent`. Take note of the following:

- The function should add `new_node` as a child only if `parent` represents a directory
- `new_node` can be added anywhere in the list of children of `parent`

```
void addChild(Node *parent, Node *new_node){

     //complete the implementation









}
```

c) Complete the implementation of function `computeSize` to compute the size of the entity represented by `root`. If `root` corresponds to a file, the function should return the actual size (in bytes) of the file. Otherwise, it should return the sum of the sizes of all the files contained under the directory represented by `root`. For example, if the sizes of *f1.txt* and *f2.txt* shown in the diagram above are 10 bytes and 20 bytes respectively and `computeSize` is called with a pointer representing directory A as an argument, it should return 30.

```
int computeSize(Node *root){
      //complete the implementation








}
```

d) Using functions `createNode` and `addChild`, write a code to create the structure shown above. You can assume that the sizes of *f1.txt* and *f2.txt* are 10 and 20 bytes respectively.

```
int main(){
      //write your code here








}
```

Question 2.

Consider an air travel system that uses a graph structure where airports are represented as vertices. Two vertices are connected by an edge if there is a direct route between the corresponding airports.

a) Assuming an adjacency matrix representation, complete the following function to check whether there is a short path between two vertices. A path is considered short if it contains at most two intermediate nodes between the source and the destination.

```
//The function returns 1 if there is a short path between
//src_vertex and dst_vertex, and 0 otherwise
//V is a constant that represents the number of vertices
//Adj is the adjacency matrix
int short_path_exists(int src_vertex, int dst_vertex, int
Adj[V][V]){
   //complete the implementation















}
```

b) Determine the worst-case big O time complexity of the function you defined in the previous part in terms of V and/or E where E represents the number of edges in the graph.

c) Complete the implementation of function `convert` to create an adjacency list representation of the graph provided as an adjacency matrix.

```
typedef struct list_node{
     int vertex;
     struct list_node *next;
} Node;

typedef struct{
     Node **adj_list;
}Graph;

Node *createNode(int vertex){
     Node *new_node = (Node *)calloc(1, sizeof(Node));
     new_node->vertex = vertex;
     new_node->next = NULL;
     return new_node;
}

//returns a pointer to the adjacency list representation
Graph *convert(int Adj[V][V]){
     //complete the implementation









}
```

d) Determine the worst-case big O time complexity of the function you defined in the previous part in terms of V and/or E where E represents the number of edges in the graph.

e) Assuming that the graph contains a large number of edges, which of the two representations (adjacency matrix vs. adjacency list) would be more suitable? Explain your answer.