PROBLEM DEFINITION:

- Given an input menu item, need to find out all the sets of ancestors to that particular input node and all its children from a JSON file representation of the graph of trees.
- Given a JSON file and the menu item as input, need to find out all paths in a undirected graph.
- Language – JavaScript.

SOLUTION:

IMPLEMENTATION TECHNOLOGY USED:

- ***HTML*** – User Interface
- ***CSS*** – Styling the User Interface
- ***JavaScript*** – Graph Path Traversal Algorithm
- ***JSON*** – Storing menu data

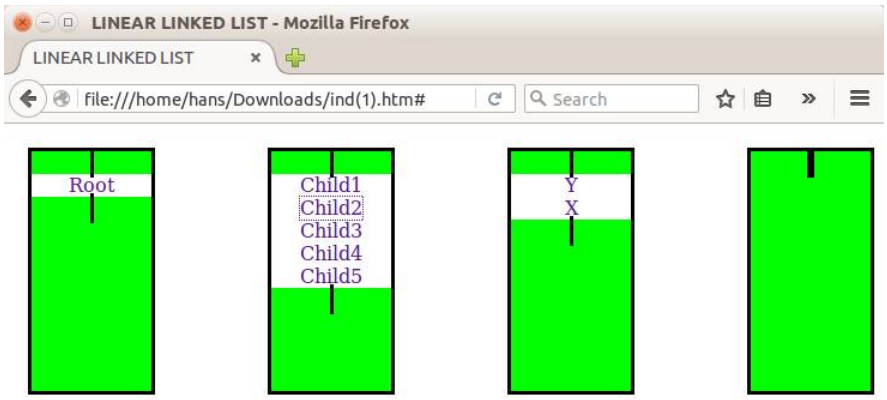
ITERATION – 1: Make a graph data structure using linked lists and edges connected together using self-referential pointers.

INPUT: NONE

OUTPUT: Menu structure iteration of inline sample input provided

ALGORITHM USED:

1. This version did not include the given menu structure inputs and was used as a test algorithm to check to see if using a data structure of self-referential pointers could be used for the JSON input provided.
2. There was a graph data structure called node that was used with insert, delete and edit functionalities.
3. Inputs were given in-line and consisted of numbers that were used to test the structure.
4. This structure had menu items appear when and only when the menu items' parent node was clicked.



REASONS FOR FAILURE:

1. *This model failed because it was found that using a data structure, while making insert, update and delete really easy, made the search algorithm highly inefficient and complex.*

2. *It was very slow in its searching even for inputs of 15-20 nodes.*

3. *Also, this model did not fit all the requirements, as it was found after an update in the requirements. While it satisfied the graph path descendant problem as specified, it failed because it didn't fit the UI structure present in the company's model I.e it didn't display all the menu items all at once and displayed descendants onclick.*

ITERATION – 2: Use a 2D array representation to map the descendants and the ancestors present for each and every node.

INPUT: *NONE*

OUTPUT: *A sequential graph of trees menu structure, onclick*

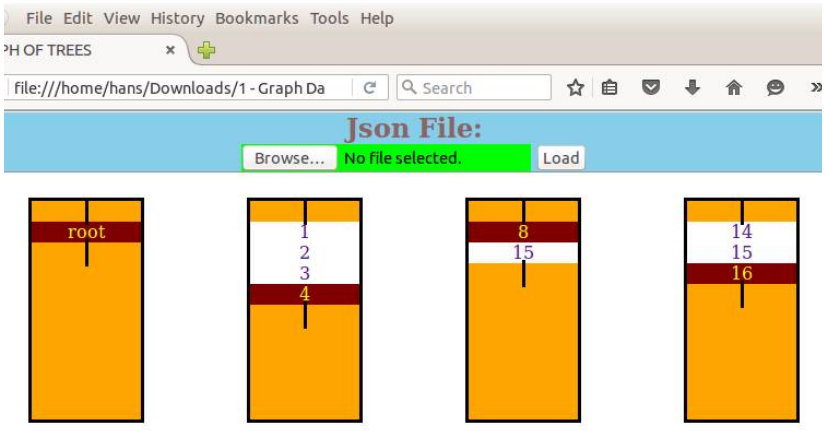
ALGORITHM USED:

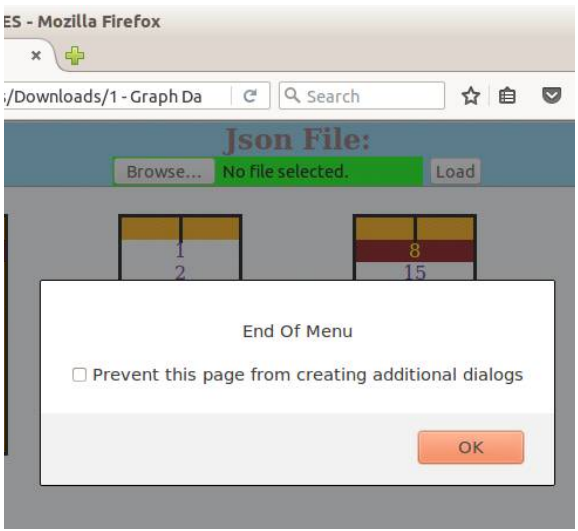
1. The input JSON file hasn't been used in this algorithm and it was only used for testing for speed.

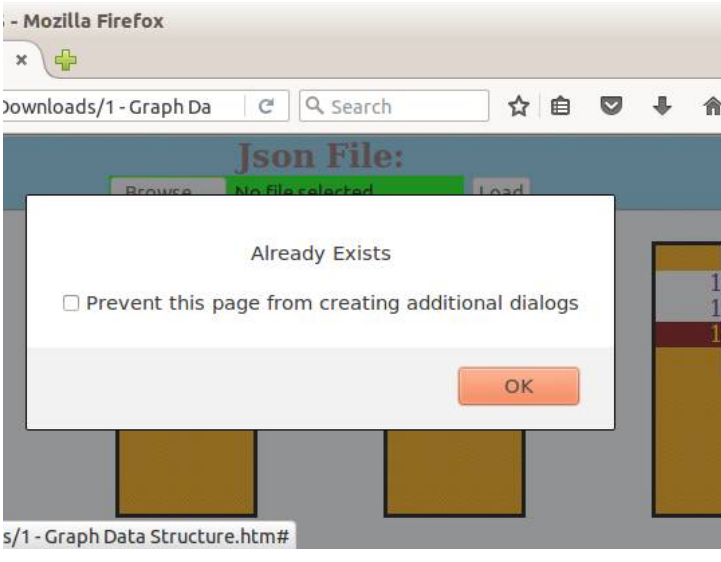
2. There were number inputs given in-line.

3. The menu structure was stored as a 2D array where every node has a unique row referring to the descendants of that particular node.

4. Also, this was a sequential one-step menu as well only revealing the immediate descendants upon click.







REASONS FOR FAILURE:

1. *This version, while much faster than the previous one, had a huge space complexity [$O(N^2)$] and hence was not scalable to the input size that is required to fit the algorithm.*

2. *Every new insert has a time complexity $O(N^2)$ making the system very rigid and thus has no openness.*

3. *Also, the sequential format didn't fit the requirements provided.*

ITERATION – 3: JSON Analysis

1. *Since the second algorithm had no urgently addressable defects, it was decided to concentrate more on the JSON structure itself.*

2. *Since this structure provided was too complex to understand because of the compressed format, an on-line JSON Viewer was used to make it easier.*

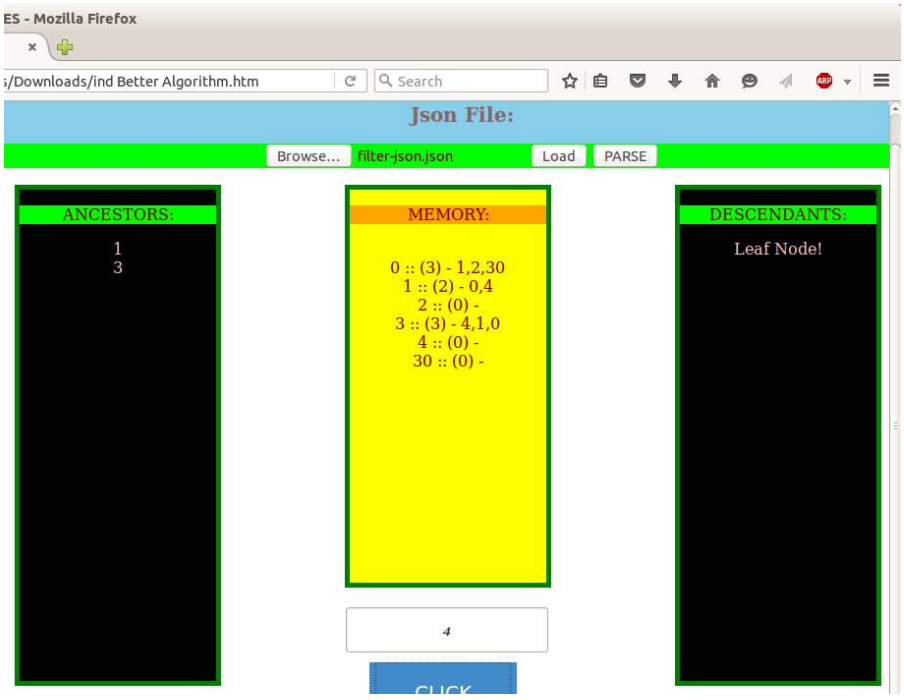
3. *The JSON file was received using multi-part form data input.*

4. *The inbuilt JSON.parse method was used on the file import input. This initially didn't work because the file given was the output from the server and hence didn't have formatting characters like double quote encloses for strings etc.*

5. *Also, even after formatting, the JSON format was quite unclear because of the size of input making readability impossible.*

6. *Queries were raised about the understanding of the JSON when a use case and a sample file was provided making understanding better.*

7. *Also, clear specification of the output required in the use case provided made me change the structure of the output file.*



ITERATION 4: Using the array representation algorithm on the JSON Sample Input.

INPUT: *Sample JSON File*

OUTPUT: *One-level ancestors and descendants.*

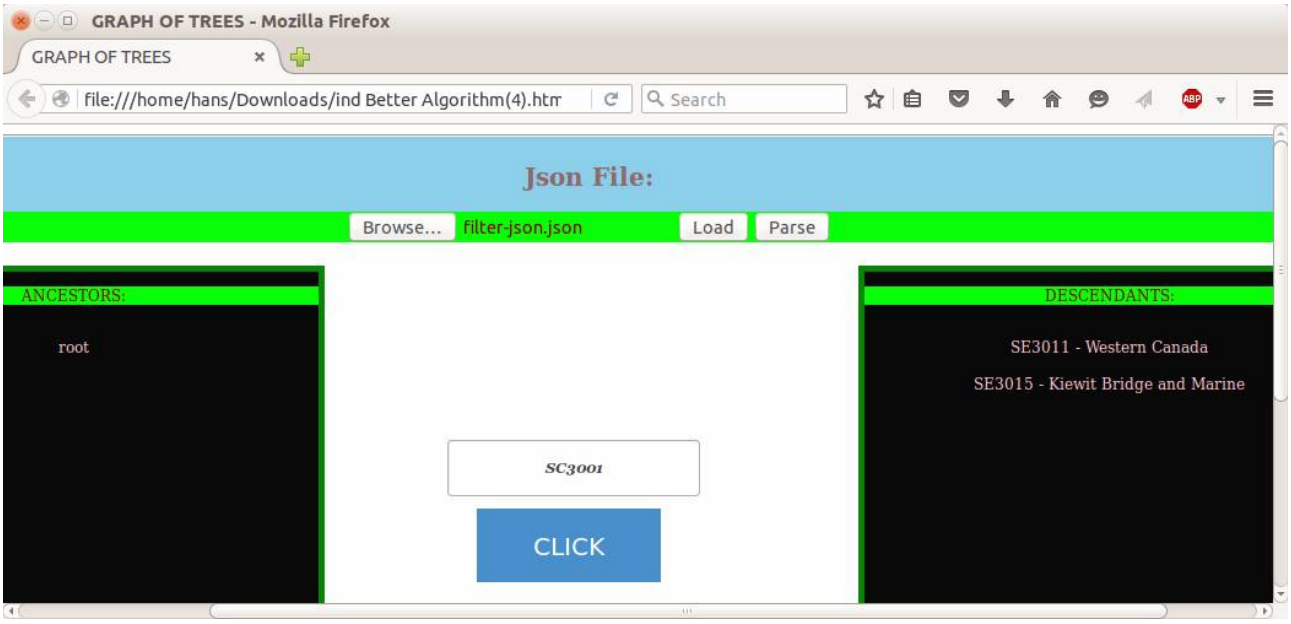
ALGORITHM USED:

1. *Because of the String data-type of the menu items present in the JSON file, it couldn't be directly fit into the 2D array algorithm which used numbers to index the nodes.*

2. *To ensure that the input fit the algorithm, a String array was created with all the nodes from the root node, thereby giving unique integer key → value pairs.*

3. *The array containing the key → value pairs was used as a reference to fit the input nodes from the JSON file into the algorithm earlier devised.*

4. *The JSON file was parsed inside the html file by means of a manual parsing function that accepted the input JSON file one character at a time, inserting each and every node into an array.*



REASONS FOR FAILURE:

1. *The manual parser used had complexity of O(N) where N is the number of characters in the word file including the newlines and the tab spaces. This made character by character parsing impossible to scale.*

2. *Because of the sequential nature of the menu structure, only one level up ancestors and one level down descendants are displayed.*

3. *The space complexity is way higher than in the 2D array method, because along with the $O(N^2)$ complexity of the algorithm, the key → value array takes up another O(N) space, making the space complexity $O(N^3)$.*

4. *The size of the key → value array was dynamic and was assumed to be a huge constant, making the model not scalable after a fixed value.*

ITERATION – 5: Recursion to traverse JS object of the JSON file. After corrections back and forth, this model worked very efficiently alleviating all of the problems addressed so far.

Complete set of requirements successfully acquired:

Given input is an interleaved menu structure, with menu nodes connected to multiple parents in a graph of trees structure in JSON format, a menu node is given.

Output expected:

A list of all paths to the input from the root to the input node – ancestors.

All the menu nodes from the input node provided to all the leaf nodes that can possibly be reached from there.

Problems faced:

1. Tree data structure made search have too much time complexity.

2. 2D array structure made the algorithm very messy and too space consuming.

3. The algorithm needed to generalize to the whole input and should be able to handle further additions, updates and deletions.

4. Since it's a menu structure, the output needed to be instantaneous and should be very quick.

5. Reading input and including into a data structure for parsing is pointless.

PROBLEM 1: TIME COMPLEXITY

When a system with a lesser time complexity is required and there is an abstract input structure that can change with the use of JSON format or any other format, an algorithm should be able to handle to load and successfully arrive at the result. This is done by means of **Recursion on the JS file variable of the JSON format.**

PROBLEM 2: SPACE COMPLEXITY

Since the other algorithm with the 2D arrays had very high space complexity, it needed to be fine-tuned such that it doesn't occupy much space. For this purpose, there were no extra structures (arrays,classes etc.) for the data collected from the recursive traversal throughout the menu. While recursion eliminates the need for different arrays for implementation, thereby increasing space complexity, it still adds to the space complexity by having a huge stack for the recursion states that get saved when the system goes into a second recursion. These **structures overlap and form a very complex collection of saved states.**

It was thought that this would be a problem and hence recursion wasn't implemented. But it was found by experimentation that for their particular menu structure and parsing to find ancestors and descendants required was fastest and occupied optimal amount of space.

PROBLEM 3: ALGORITHM

It was too tedious to get each and every unique name as a token and use it with a referential array structure to access the tokens, since it increased the complexity N-times for array access. Therefore, **string manipulation** was decided to be used. When the recursive traversal of the JSON document occurs, there is a string variable called path. **This variable was adaptive such that when a second descendant came into the picture, it replaced the first one.**

PROBLEM 4: JSON INPUT

There was some debate on how to insert the JSON file into the algorithmic display screen. We had to decide between Server-Client architecture where the server was sending the file to the Client, which was the front-end to the algorithm, using AJAX. But when server sends information in JSON format, it increases the complexity of the project undertaking because of the need for a hosting service. This was undesirable as long as other methods could be used. Therefore, it was decided to **insert the file as multi-part form data that needed to be uploaded into the document and parsed** before the algorithm could be implemented.

PROBLEM 5: SPACES

Spaces weren't accepted by JS because JS was programmed to **ignore them**. Hence, the special format non breakable space character (** **) was used instead.

PROBLEM 6: REPRESENTATIONAL INPUT

The algorithm worked perfectly for the sample test file they gave us. But Caterpillar wanted a system in which certain nodes didn't need to be displayed and were only present because they were representational. Therefore, these nodes were required to be taken out of the menu structure output. This was done by **maintaining an array representations that had a list of all the nodes that were not required.**

PROBLEM 7: LIST OF ALL ANCESTORS

The algorithm initially only gave the first occurrence of a successful traversal path from the root node. This was undesirable because a list of all ancestors were required. Therefore, the algorithm was changed to check for all possible inputs inside a loop. But the loop redundantly checked existing paths every time because it traversed from left extreme to the right (DFS) for every unique path. This made the algorithm extremely slow. Therefore, **a new idea was implemented where a variable was used to track the depth of the tree as traversal** was done and it stopped when it found the input node. Therefore, **this variable was used as a checkpoint** to ensure that after the first successful path, the traversal never goes below the level that the node was found in (Modified BFS). Also, the search was made linear where there is one complete traversal from the left to the right noting down all the ancestors at once.

PROBLEM 8: PRESENCE OF ANCESTOR RELATIONSHIP DECIDING DESCENDANTS:

There was one level of nodes on the structure, the children of the representational input, that required that the descendants varied based on the project under which they were placed. This made it difficult because the algorithm hinged on a delicate recursion mechanism and any break could give unexpected results. Therefore, conditions were imposed inside of the recursion function to address this particular scenario such that when the input is one of the descendants of the representational input, that the set of descendants were displayed alongwith the project under which they are grouped in.

PROBLEM 9: BREAK WHEN LIST OF ANCESTORS ARE FOUND

This was a very big problem because at the point where all the ancestors were found to the particular input node, the function was sitting on top of an arbitrary size of recursion layers that all needed to be returned to without changing the actual implementation of the algorithm. One of the first methods thought of was to **use a flag variable** indicating that indicates that all paths have been successfully traversed. But the problem with this methodology is that there is no way of knowing whether all paths have been found. It was only possible to find out whenever the input node has been reached which would be **useless to flag**. Therefore, the whole function call was **placed under a try block and the system was exception handled out of the recursion for every time a successful path is found.** This function was implemented in a loop until all paths were found.

ACTUAL ALGORITHM:

INPUT:

(1) **filter-json.json –** File containing menu in JSON format
(2) **Name of node –** Entered in text box for output

OUTPUT:

(1) **List of Ancestors** – List of nodes from parent to entered input node
(2) **List of Descendants** – List of all nodes from current to leaf nodes

ALGORITHM:

1) Input the JSON file into the page using the controls provided. {1}{2}
2) Receive the file and create a JavaScript object file "obj" to manipulate the imported file. {1}
3) Receive the text box input as name of node whose ancestors and descendants are displayed.
4) Traverse obj recursively to find out which branch from the root the given input node belongs to. {3}
5) When the input node is found, throw the current version of obj using a user defined exception to handle the display of all descendants separately. {2}
6) Using the object thrown, all its descendants are recursively added to the path string. {3}
7) Provide proper indentation for ancestor and descendant display.
8) The path string is used to fill the ancestor and descendant lists separately.

PSEUDO CODE:

```
START
CALL handleFileSelect and hence receivedText to get JSON file input        {1}{2}
Parse the JSON file to a JavaScript object "obj"                           {1}
Receive name of node input and proceed if it is a genuine node name
CALL find with obj and the name of node
IF input supplied is present in the path string
        Display ancestors and descendants from the path string
ELSE
        Display error message about bad input
ENDIF
END

FUNCTION find
        PASS IN: JavaScript Object for JSON file and the Input name of node
        BEGIN
                CALL recursive with obj and input node
        EXCEPTION                                                          {2}
                WHEN obj type is thrown
                        CALL desc with obj caught by exception
```

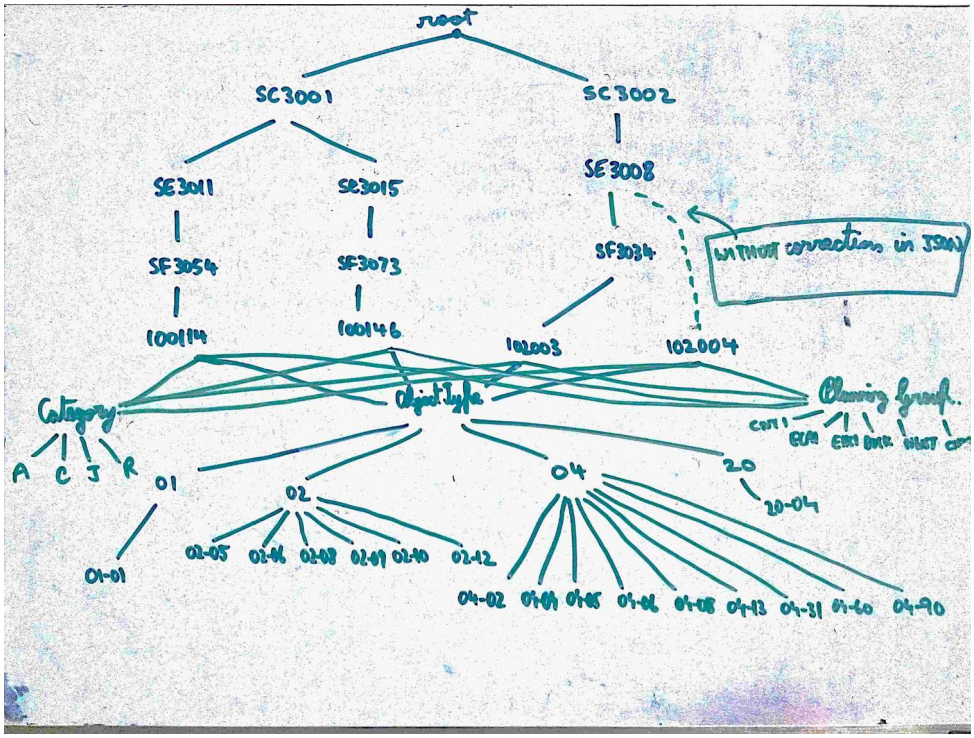
```
        END
        PASS OUT: The path string
ENDFUNCTION

FUNCTION recursive
        PASS IN: The JavaScript object “obj” and input name of node                {3}
        FOR all keys present in obj
                IF type of obj[key] is object
                        Add key to path string
                        CALL recursive with obj[key] and input name of node
                        Remove the key attached to the path string
                        IF key is the input name of node
                                THROW obj
                        ENDIF
                ELSE
                        Add new leaf node by removing already existing leaf node if any
                ENDIF
        ENDFOR
        PASS OUT: The path string
ENDFUNCTION

FUNCTION desc
        PASS IN: The JavaScript object for JSON file at input node depth           {3}
        FOR all keys present in obj
                IF obj[key] is a leaf node
                        Add all possible keys of obj to path
                ELSE
                        Add the immediate descendant to the path string
        ENDFOR
        PASS OUT: The path string
ENDFUNCTION

FUNCTION ParseJSON
START
Accept input
TRY
        CALL recursive to find ancestors string
CATCH
        CALL desc to find all descendants
Display results using appropriate variables
ENDFUNCTION
```

DESCRIPTION:

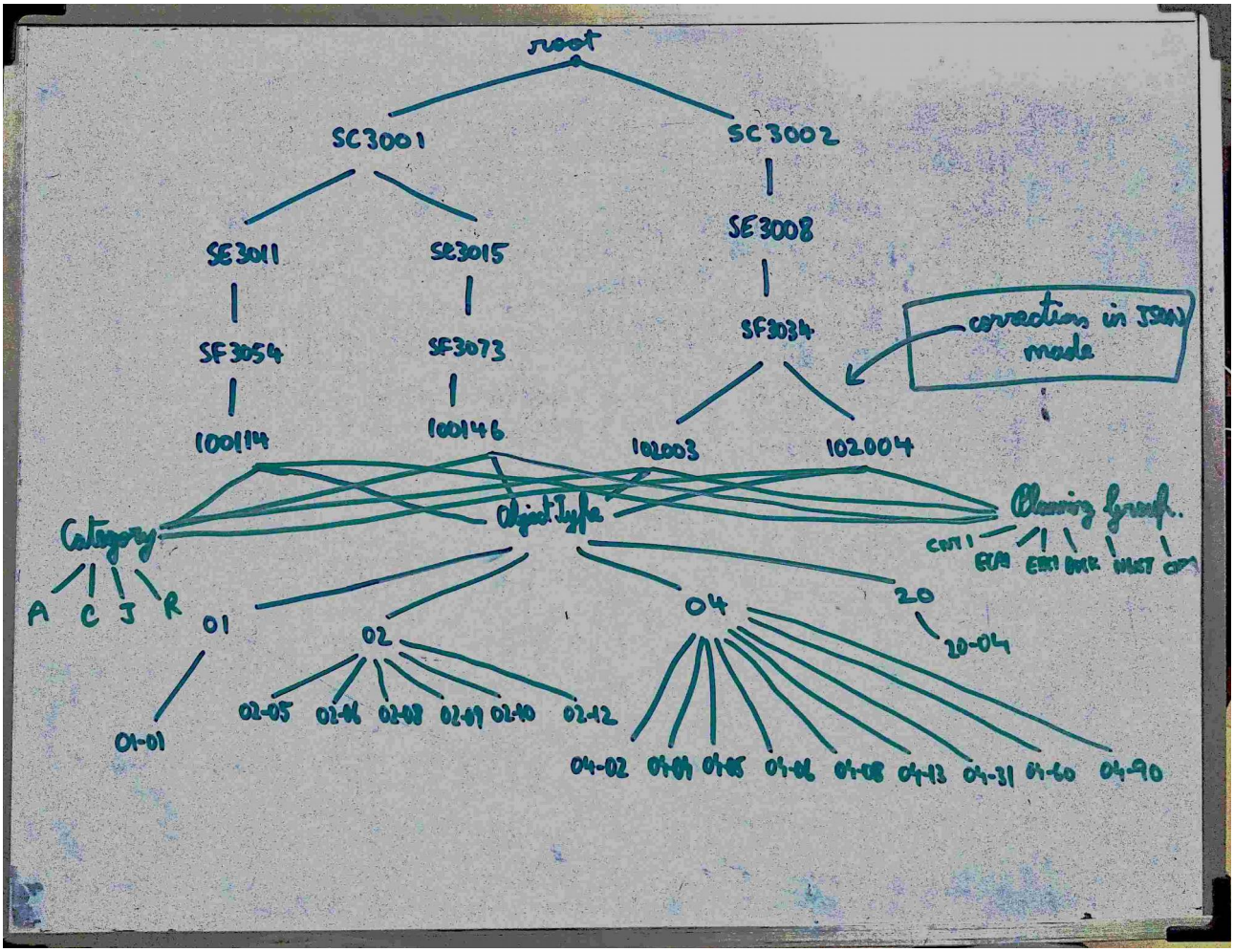
1. The JSON file is accepted as input and parsed into a JavaScript object.
2. The object is then recursively traversed by a recursive DEPTH FIRST SEARCH strategy, while storing the path traversed into a string, until the input node is found.
3. Once input node is found, an exception is thrown using the object at its current depth.
4. That object is then used for a similar DFS traversal to add all child nodes to the same path string.
5. The path string is then properly indented according to the use case specified.

CORRECTIONS MADE:

WITHOUT CORRECTION IN SAMPLE JSON FILE:



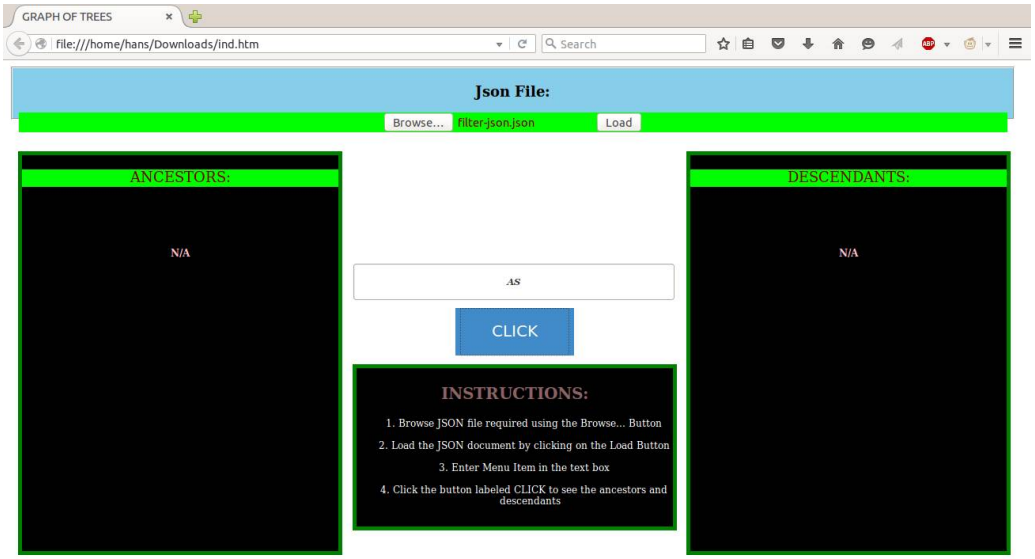
AFTER CORRECTION IN SAMPLE JSON FILE:



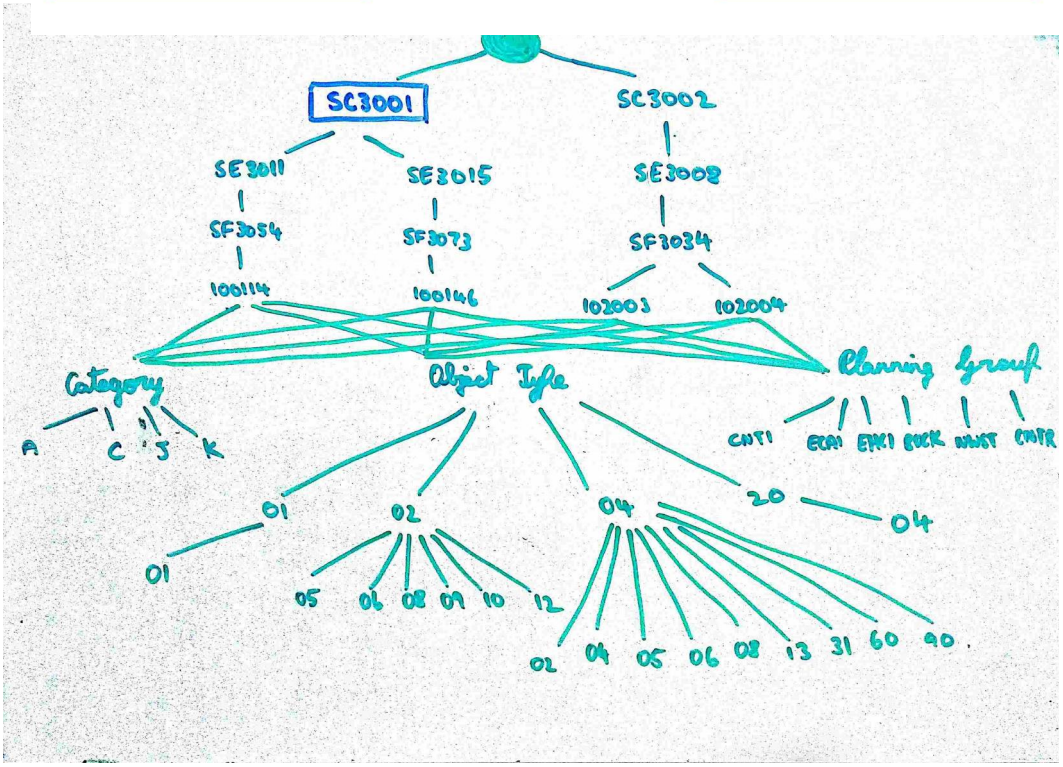
TESTING:

TEST CASE 1: Random Input not present in node list.

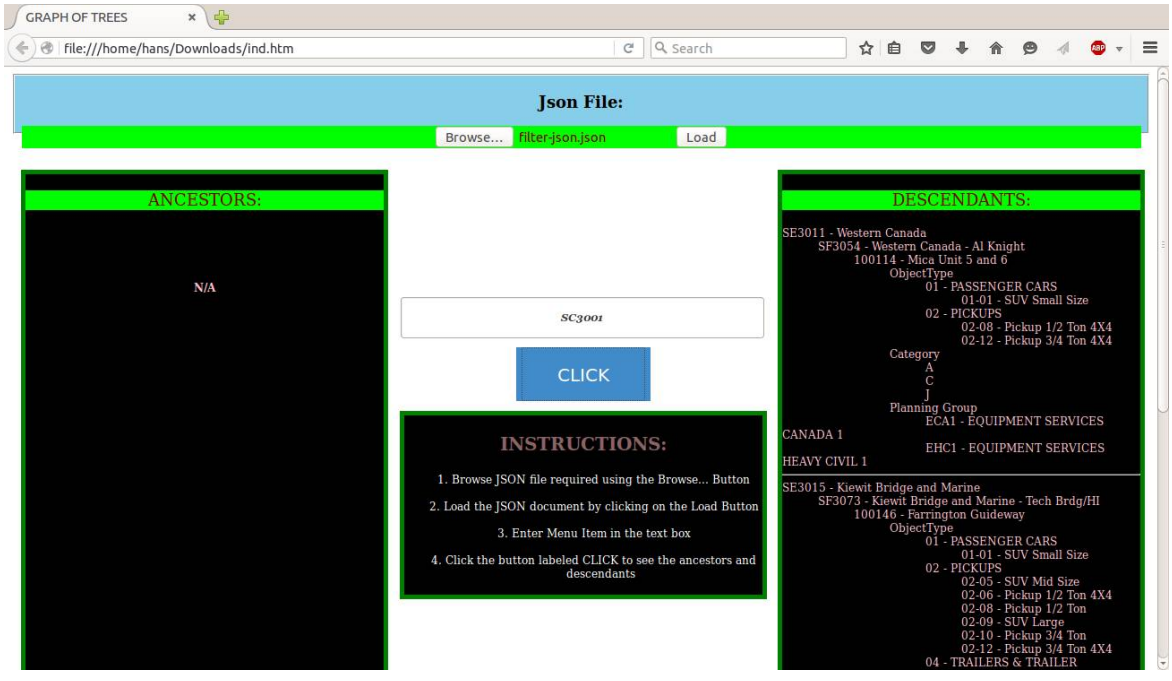
Eg: "XYZABC"
OUTPUT 1: Error Message

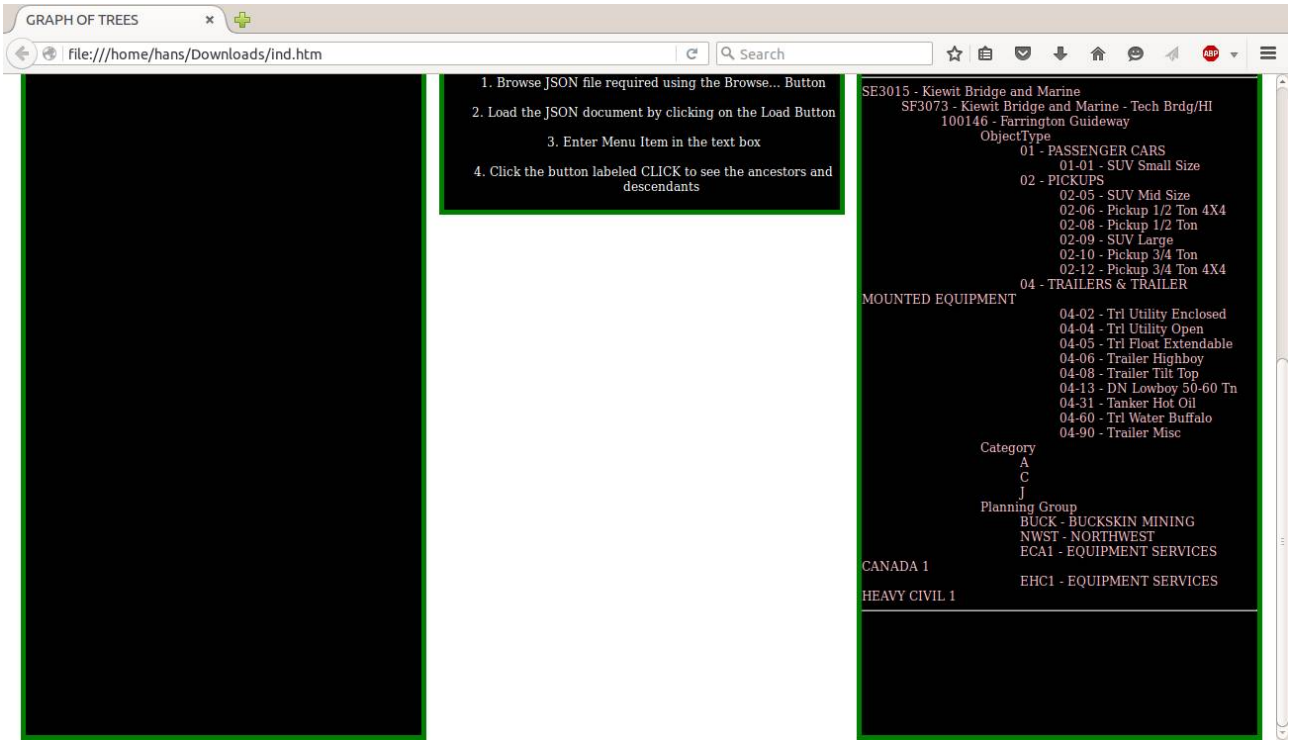


TEST CASE 2:

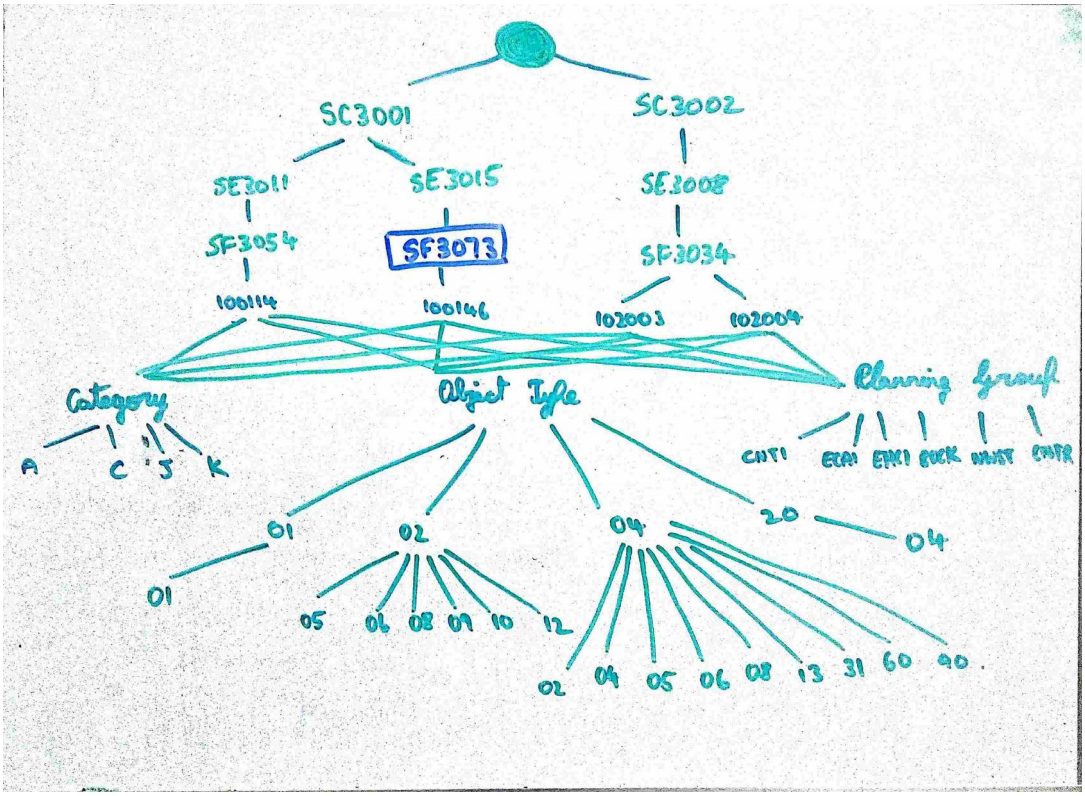


OUTPUT 2:

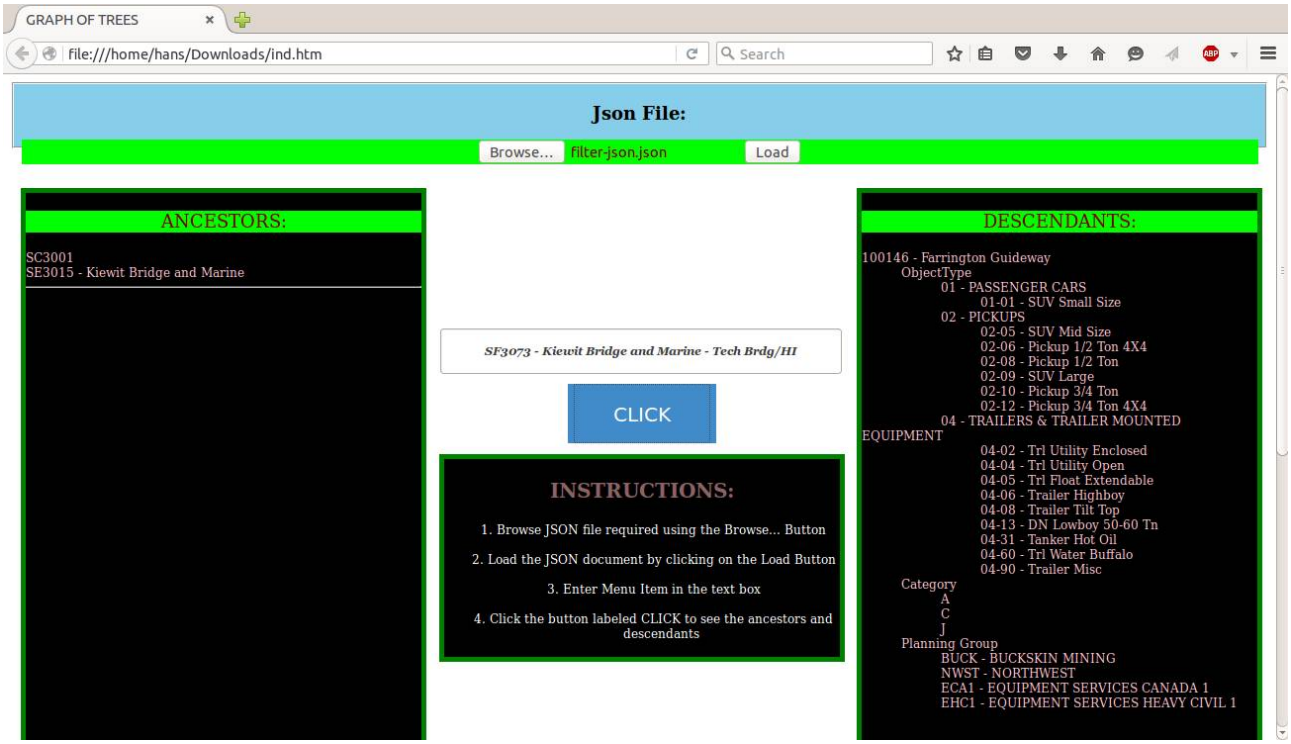




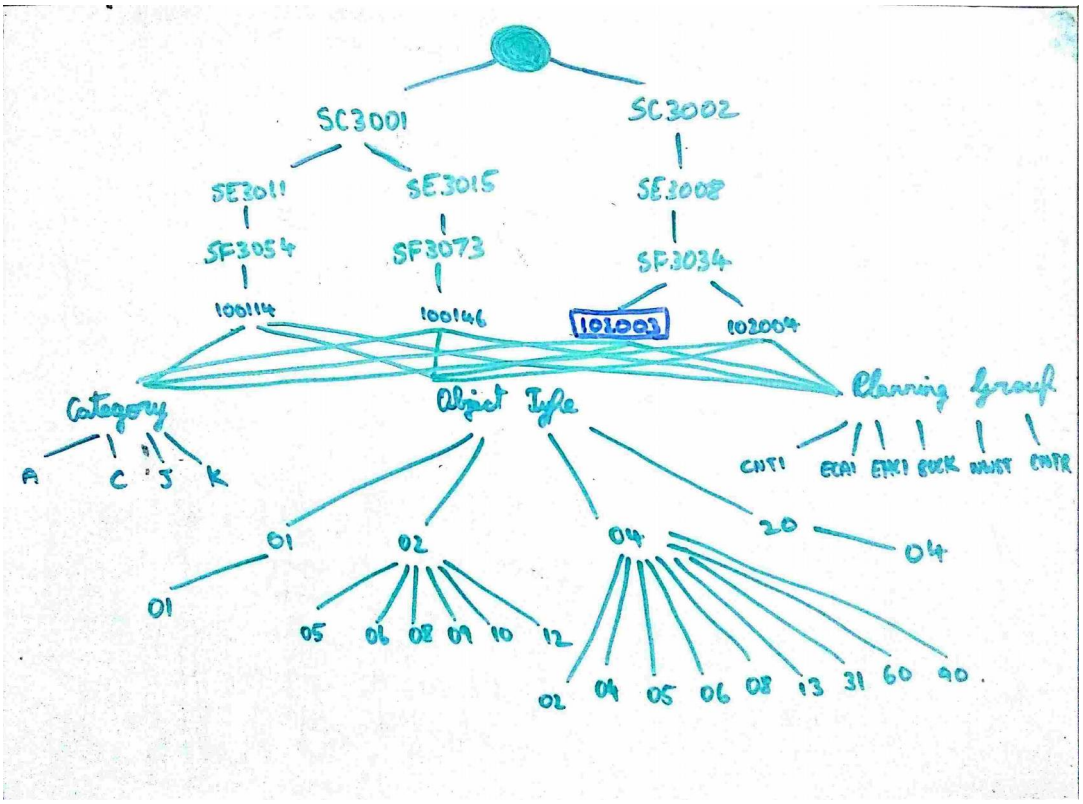
TEST CASE 3:



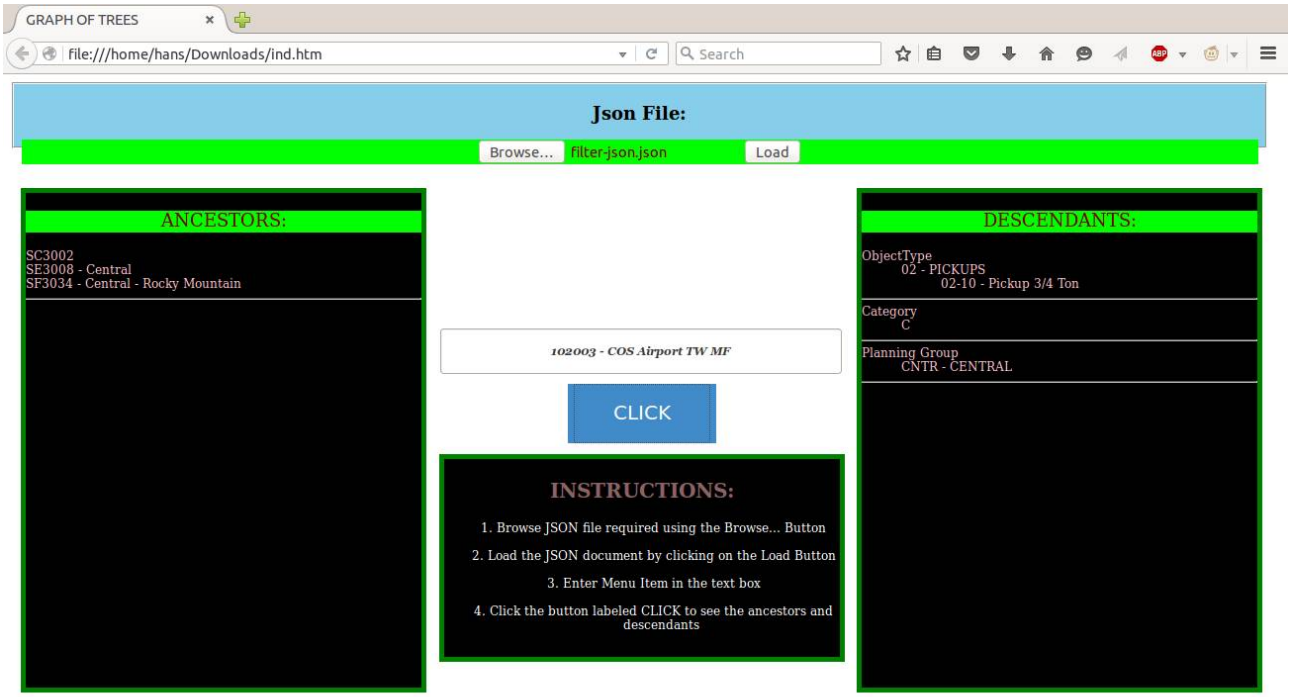
OUTPUT 3:



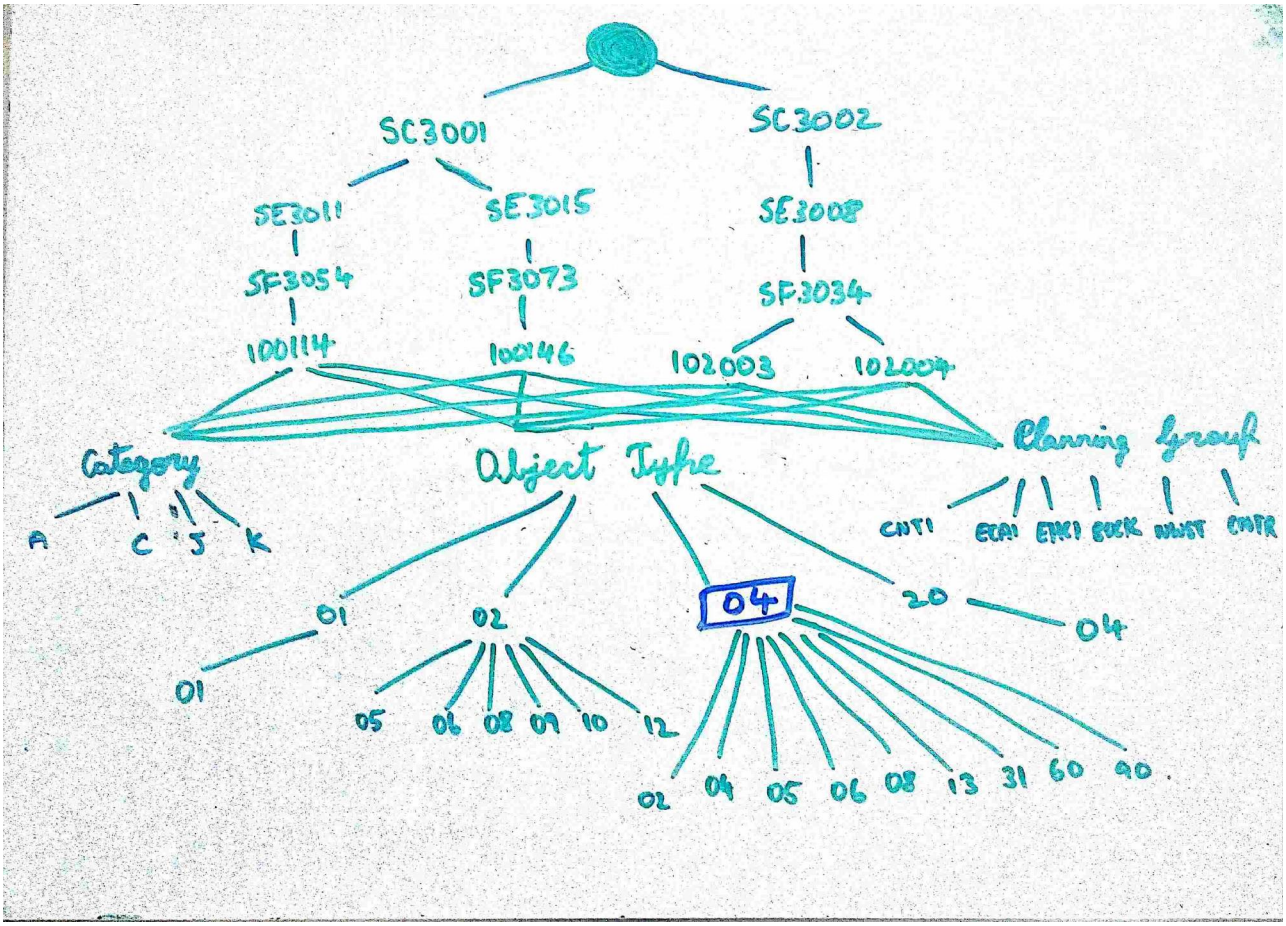
TEST CASE 4:



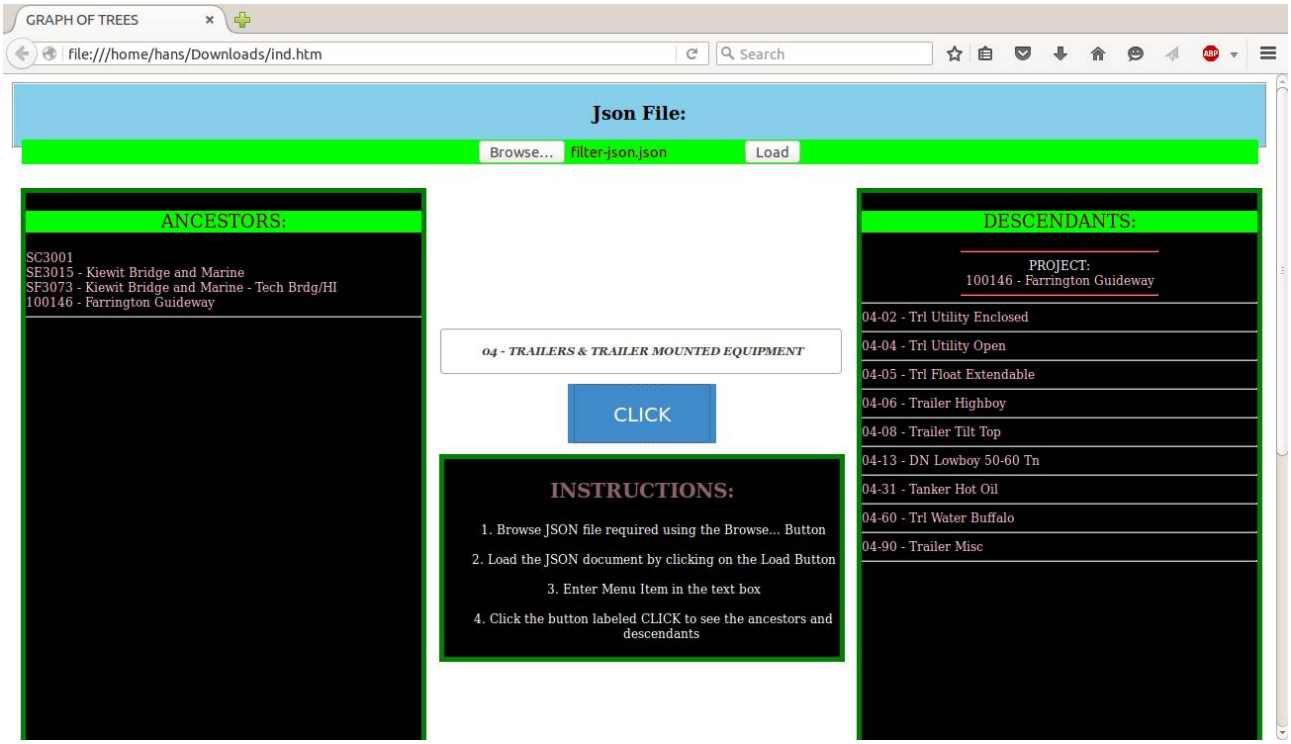
OUTPUT 4:



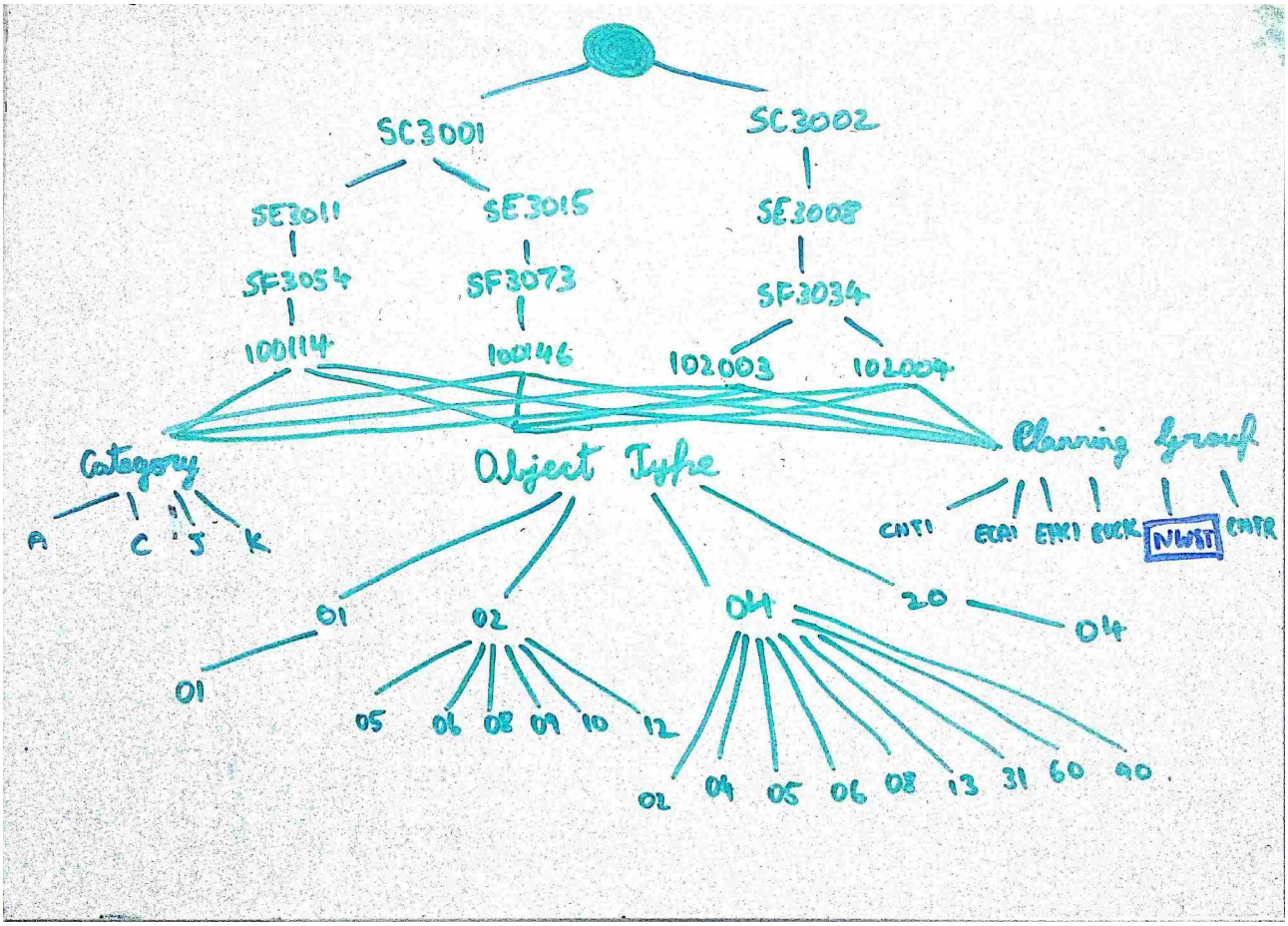
TEST CASE 5:



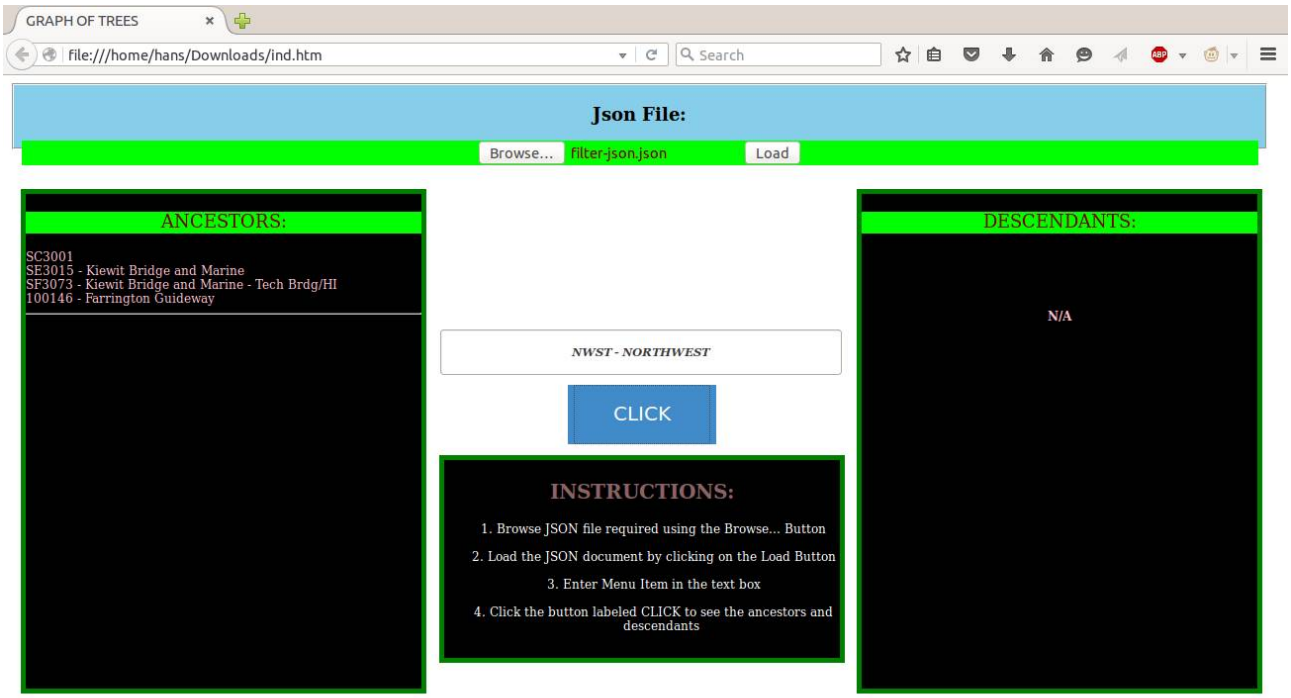
OUTPUT 5:



TEST CASE 6:



OUTPUT 6:

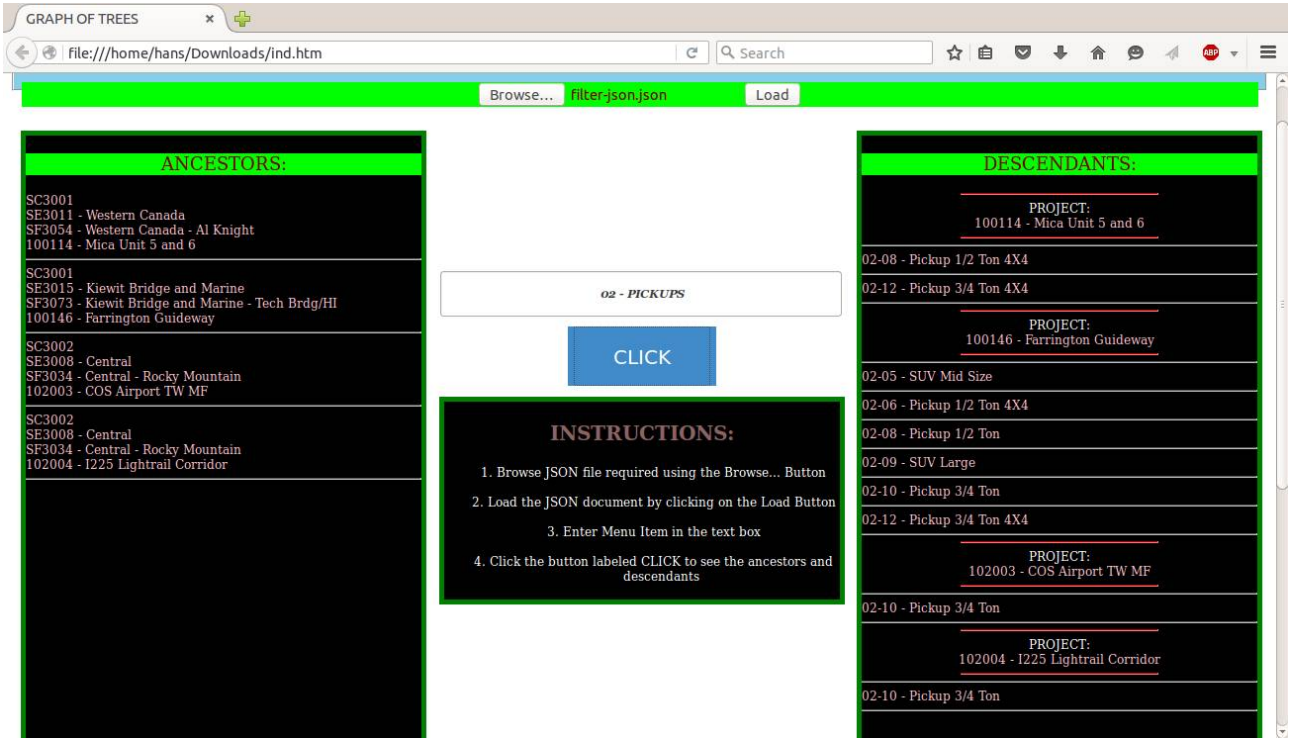


NOTE:

For different objectType elements,

01 - PASSENGER CARS,
02 – PICKUPS,
04 - TRAILERS & TRAILER MOUNTED EQUIPMENT,

the leaf nodes are presented grouped under projects.



Even though this algorithm worked for the sample JSON provided, it didn't for the big file. This was because there was a change in structure between the sample provided and the main JSON structure. Therefore, careful analytical changes were made to the big JSON file to fit it into this algorithm.

ITERATION – 6: All paths for an entirely new JSON structure.

PROBLEM FACED: Identification and differentiation of arrays from JSON objects

This new JSON structure had all new objects present inside arrays, even if the size of the array was only one. While it is easy in C++/Java to detect if it's an array or an object, it isn't that easy in a generic variable language like JavaScript. This made identification a difficult task. Since typeof arr returned “object” for both arrays and objects, a new method had to be thought of. This was done by means of checking that not only the typeof arr is “object”, but also seeing if arr[object.length-1] I.e the last element of the array was defined. If both conditions were satisfied, it was an array. Otherwise, it was an object.

INPUT: A new JSON structure needed to be parsed with the same requirements

OUTPUT: List of ancestors, all descendants.

ALGORITHM:

1. The JS object for the JSON file was created from the imported file.
2. The input node was acquired from the text field as input.
3. The recursive function takes the object, sees if current entry is array or object.
4. If the current entry is an array, for all elements inside the array, the function is recursively invoked.
5. If the current entry is an object, it's id field (arr[“id”]) is checked to see if it matches the input node.
6. If it matches the input node provided, the function is exception handled out of the recursion with the object thrown for the descendants function.
7. The thrown object is traversed, storing information as the current node information.
8. When an object comes inside the input node's object, it is sent to the descendant function to find all possible descendants using DFS.
9. The path string is modified such that when a second descendant comes in, it replaces the first descendant from the desc file used for output

PESUDO CODE:

```
START
CALL handleFileSelect and hence receivedText to get JSON file input        {1}{2}
Parse the JSON file to a JavaScript object “obj”                           {1}
Receive name of node input and proceed if it is a genuine node name
CALL find with obj and the name of node
IF input supplied is present in the path string
        Display ancestors and descendants from the path string
ELSE
        Display error message about bad input
ENDIF
END
```

FUNCTION rec
START
IF arr is an object
    FOR all keys in arr
        IF arr[key] is an entry
            Add “key : arr[key]” pair to string
            IF r is 1
                Remove current descendant from string
                r:=1
            IF arr[“id”] is the input
                THROW the object
        ELSE
            CALL rec with arr[key] and input as parameters
ELSE
    FOR all array elements
        CALL rec with arr[index] and input as parameters
ENDFUNCTION

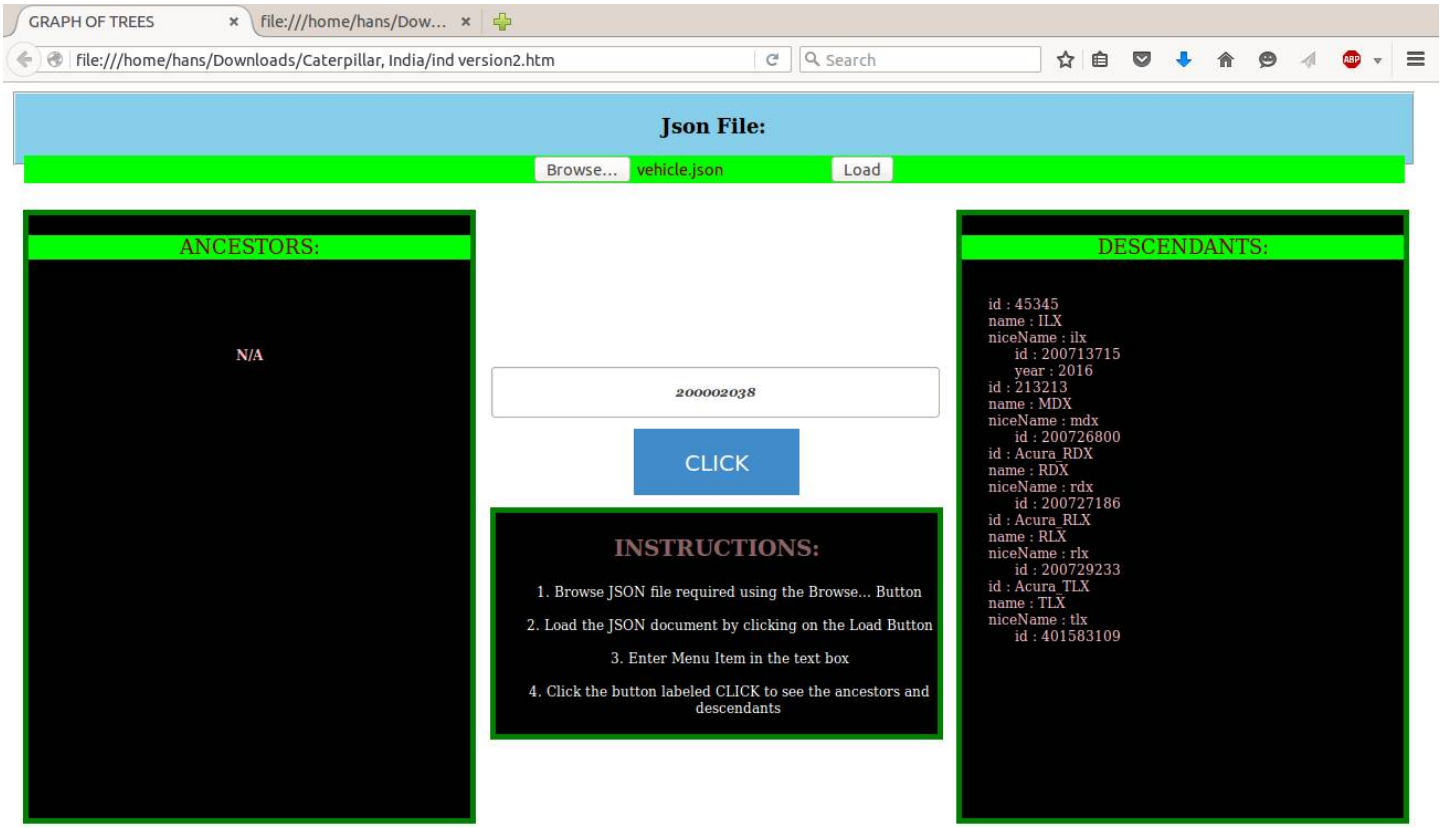
FUNCTION des
START
FOR all keys in arr
    IF arr[key] is not an object
        IF “key : arr[key]” pair is not present in string
            Add “key : arr[key]” pair to string
    ELSE
        CALL des with arr[key] and input as parameters
ENDFUNCTION

FUNCTION curr
START
FOR all keys in arr
    IF arr[key] isn't an object
        continue
    ELSE
        return arr[key]
ENDFUNCTION

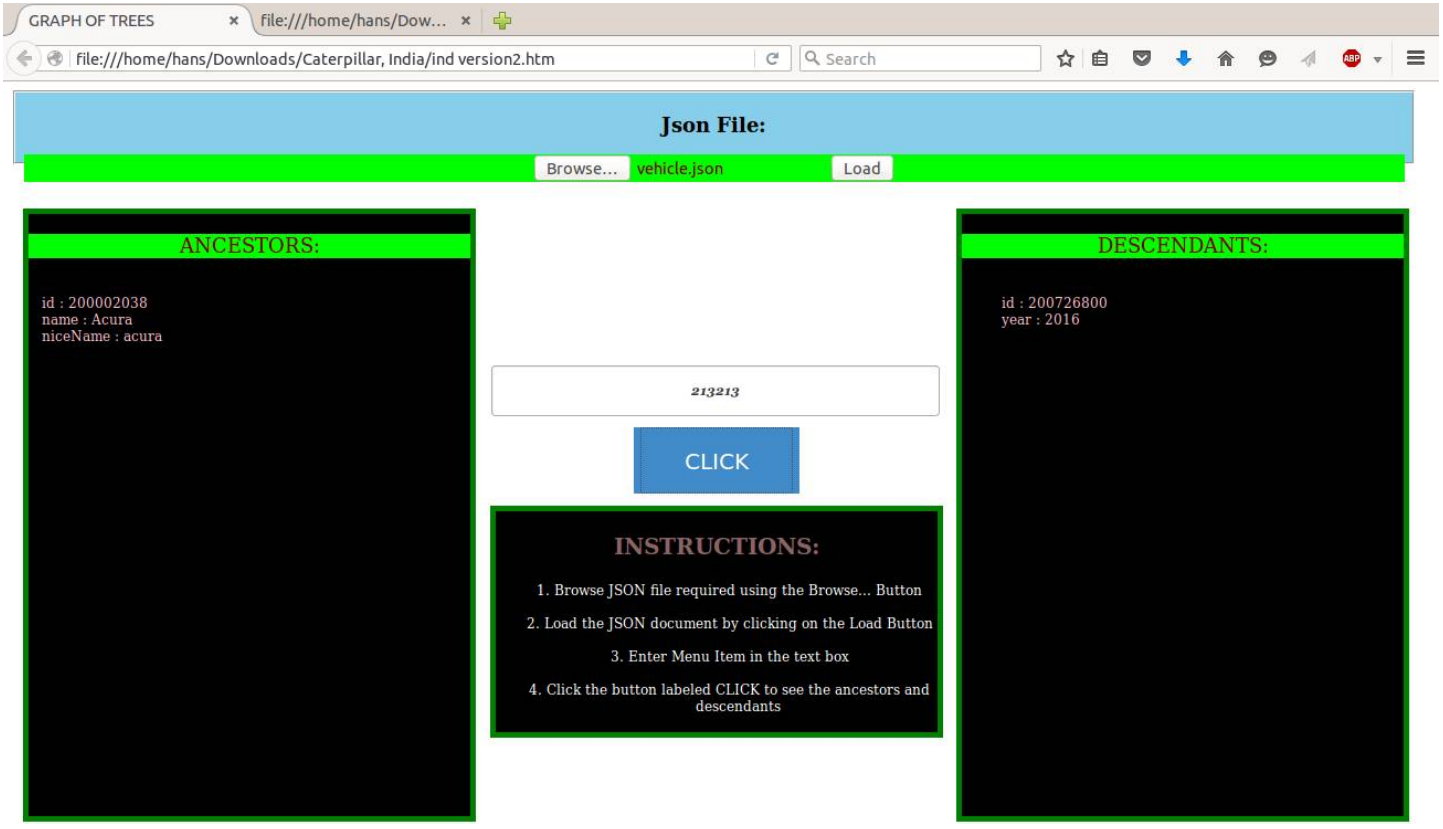
FUNCTION ParseJSON
START
Accept input
TRY
    CALL rec with input and JS object for JSON structure
CATCH
    CALL curr with thrown object and pass that as parameter to the CALL of des
Display results using appropriate variables
ENDFUNCTION

INPUT/OUTPUT:

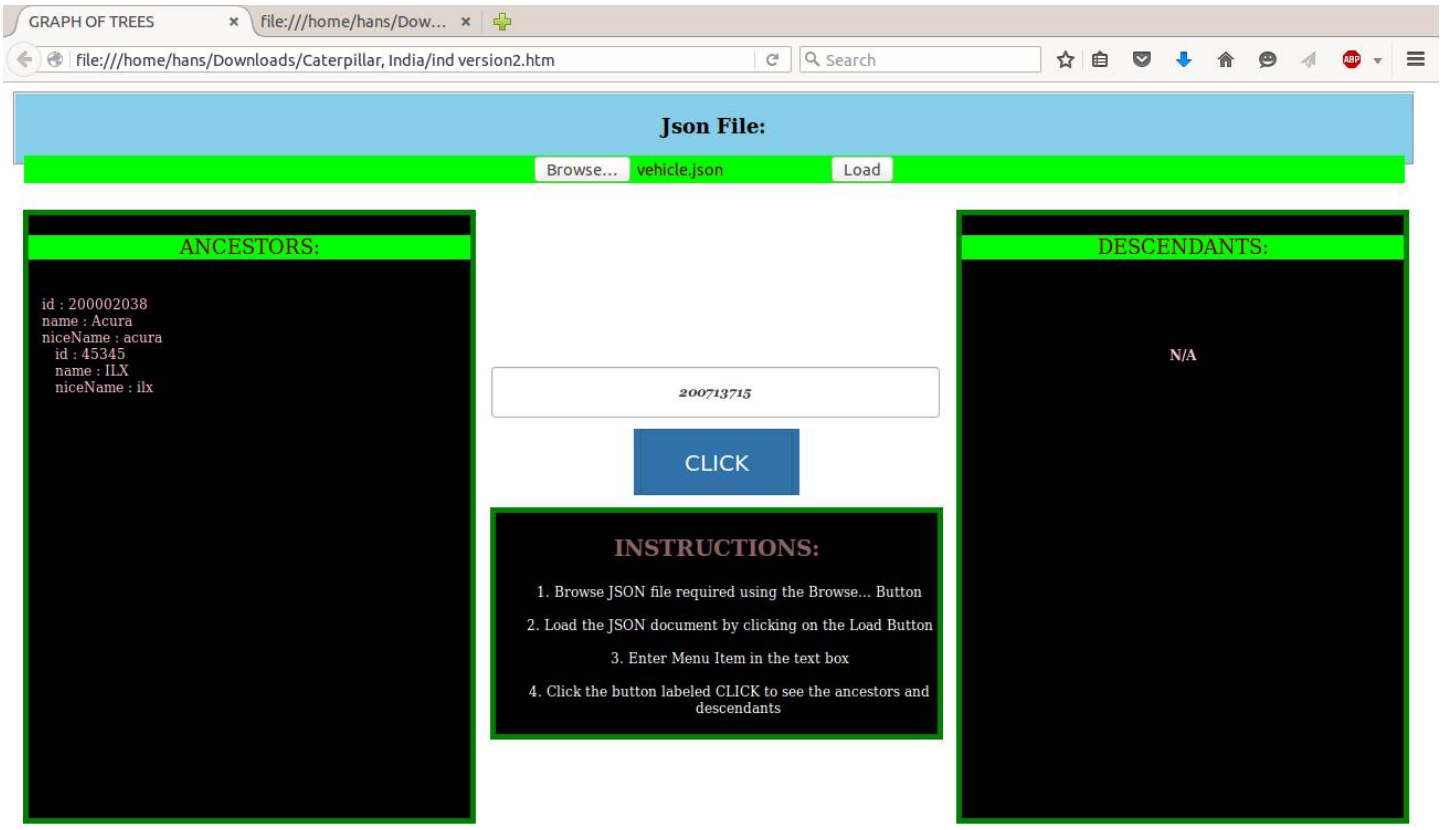
CASE 1: Top most node



CASE 2: Intermediate node



CASE 3: Leaf Node



REFERENCES:

{1} ***http://www.w3schools.com/***

{2} ***http://stackoverflow.com/***

{3} ***http://www.youtube.com/***