2021 International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON) | 978-1-6654-2503-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/SMARTGENCON51891.2021.9645747

Code-Viz: Data Structure Specific Visualization and Animation Tool For User-Provided Code

Kumar N S
Computer Science and Engineering Department
PES University
Bangalore, India
kumaradhara@gmail.com

Revanth Babu P N
Computer Science and Engineering Department
PES University
Bangalore, India
putturevanth@gmail.com

Sai Eashwar K S
Computer Science and Engineering Department
PES University
Bangalore, India
saieashwar.ks@gmail.com

Srinath M P
Computer Science and Engineering Department
PES University
Bangalore, India
srinathmp77@gmail.com

Sreyans Bothra
Computer Science and Engineering Department
PES University
Bangalore, India
sreyansbothra@gmail.com

***Abstract* — Data structures are used by programmers to help in the arrangement of data for a particular purpose. The use of the right data structure improves the performance of algorithms and increases time and space efficiency.**

Data structures involve the usage of pointers and references for representing user's data. It is very difficult for new users/learners to see how these pointers and references change. It is also very difficult to find a correlation between the program being executed and the data structure being used in the code.

Immersive understanding and involved learning are enabled by visualisation of these data structures. As a result, the user can appreciate the way the pointers and references are used in data structures.

The aim of Code-Viz is to provide data structure specific visualisations and animations that are both intuitive and accurate. The tool is language agnostic, supporting major programming languages (currently C and Python). Code-Viz is implemented as a browser-based tool with a code editor where users can visualize code without the need to install any software, thus facilitating ease of access.

Keywords* — *Visualization, Animation, Imagery, Data Structures, Language Agnosticism, Immersive Understanding, Involved Learning

I. Introduction

The idea for this project was inspired by the need of visualisation for user defined algorithms and data structures. Data Structures are an essential aspect of many computer algorithms because of the efficiency that programmers achieve while managing data. A correct selection of data structures can enhance the efficiency of computer programs or algorithms. Therefore, understanding these data structures is pivotal to successful development of efficient code. Visualization of these data structures aids in immersive understanding and involved learning. Visualizing code aids in the identification and correction of bugs, the conceptualization of large-scale projects, the visualisation of dependencies, collaboration with others, the onboarding of new software developers, and the comprehension of user flow.

Code-Viz aims to provide intuitive and accurate visualisations for the code which the user provides. The user can either upload code or edit and submit code on our online text editor.

The tool further is language agnostic and as a proof of concept provides support for C and Python. With the right choice of debuggers for a language of user's choice, the tool is able to create an explanatory visualization .

Additionally, Code-Viz hosts code examples and visualisations, for standard data structures and algorithms which can be used as tutorials.

II. Problem Statement

Building a tool which accepts code as input from the user and generates simple, explanative visualization of line-by-line execution of the code, and provides data structure specific animation for the data structure used.

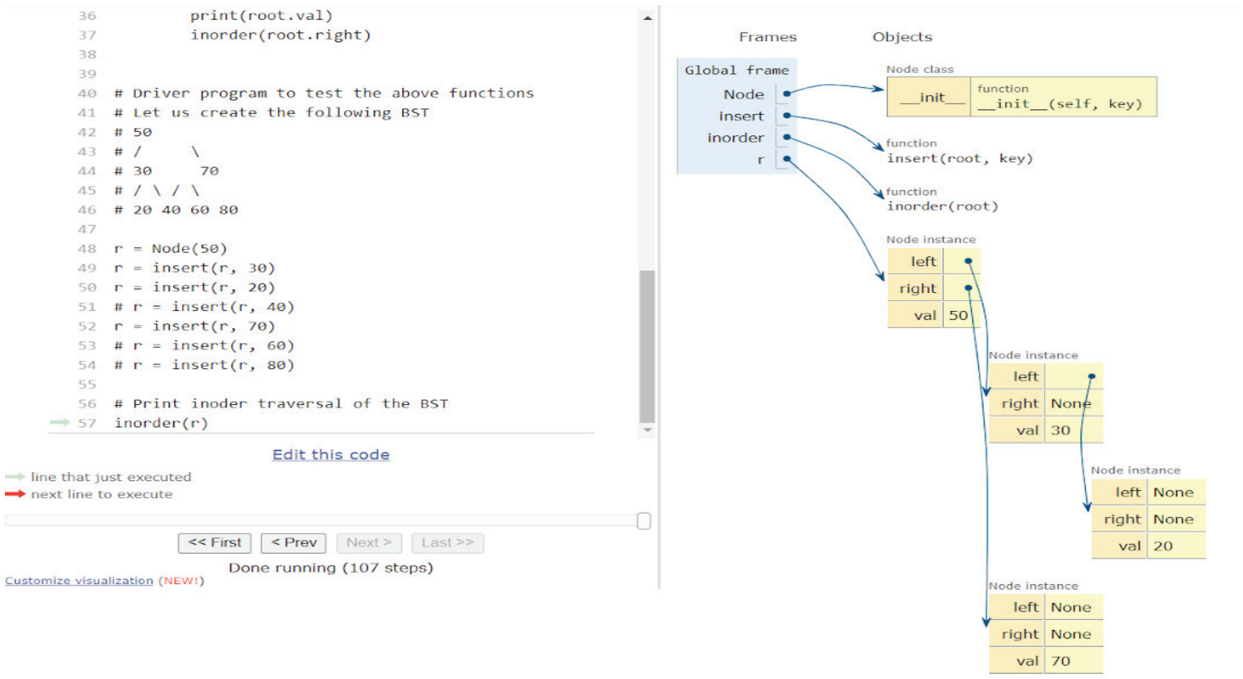
III. Literature Review

Many researchers have focussed on solving the problem of code visualization and animation. As a result, many code visualization tools have been developed.

The most notable tool is PythonTutor [1] which is an online language agnostic tool to visualize user provided code written in various languages like Python, C, C++ etc. This tool works by taking the input code and produces an execution trace which consists of stack, heap, global variables' values at every line of execution of code. Fig. 1 shows PythonTutor visualization of a binary search tree which shows the global frame and associated objects however the visualization does not reciprocate the usage of binary search tree.

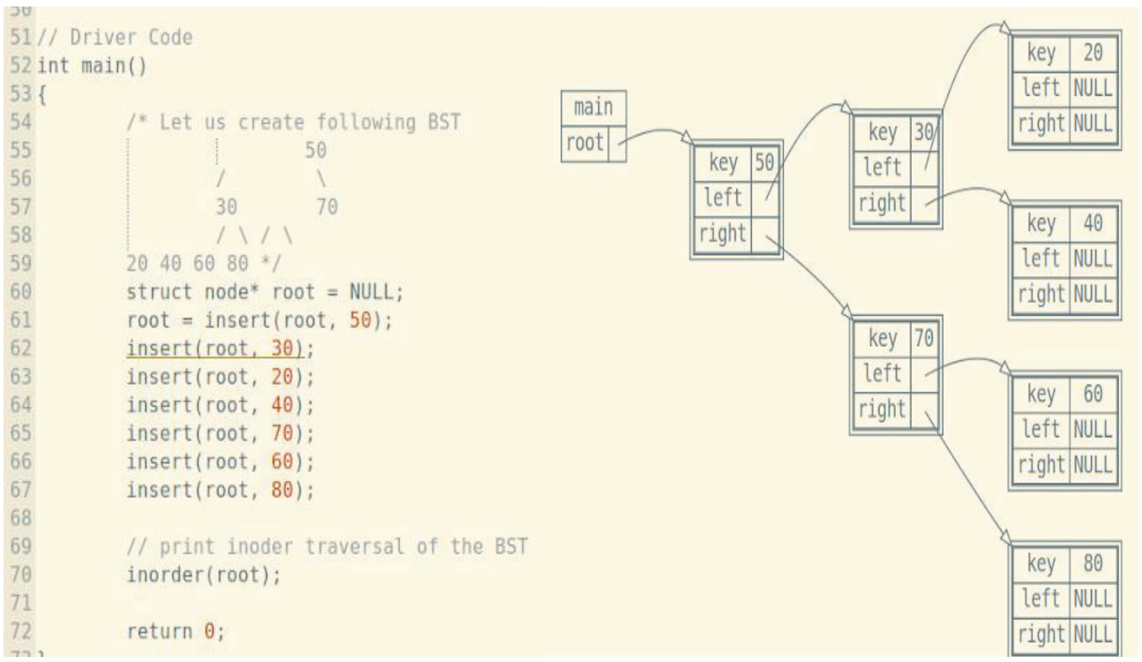
Similarly, SeeC [2] has been developed as an offline visualization tool for user-given code written in C language. It runs natively on the user's machine and hence can be used offline and requires installation of the tool along with other dependencies such as LLVM Clang.

The tool gives a line-by-line description of each step of code and tells where the code might be going wrong (like null pointer dereferencing) by using an execution trace which can be played and replayed. However the installation and usage is very tedious requiring repeated compiling for every change in input code. Fig. 2 shows the visualization of a binary search tree using SeeC tool which is relatively interpretable but has poor usability.



```
        print(root.val)
        inorder(root.right)


# Driver program to test the above functions
# Let us create the following BST
# 50
# /      \
# 30      70
# / \ / \
# 20 40 60 80

r = Node(50)
r = insert(r, 30)
r = insert(r, 20)
# r = insert(r, 40)
r = insert(r, 70)
# r = insert(r, 60)
# r = insert(r, 80)

# Print inoder traversal of the BST
inorder(r)
```

Fig. 1. Visualization of Binary Search Tree in PythonTutor



```
// Driver Code
int main()
{
        /* Let us create following BST
                     50
                  /        \
                 30        70
                 / \ / \
        20 40 60 80 */
        struct node* root = NULL;
        root = insert(root, 50);
        insert(root, 30);
        insert(root, 20);
        insert(root, 40);
        insert(root, 70);
        insert(root, 60);
        insert(root, 80);

        // print inoder traversal of the BST
        inorder(root);

        return 0;
```

Fig. 2. Visualization of Binary Search Tree in SeeC

PVC.js [3] has also been developed to visualize user provided C language code and it runs on the user's browser i.e. it is an online tool.

A common feature that these above mentioned tools miss is the ability to provide data structure specific visualizations. These tools provide generalized visualization as opposed to visualization tailored for the underlying data structure, for example all programs consisting of binary trees are represented in a horizontal manner rather than a tree-like visualization. So there is no way of representing trees, graphs in a form they are thought of / taught in.

Data Structures Visualization [4] tries to solve this problem by providing visualization for many data structures but does not work for user provided code.

Our tool Code-Viz helps to solve the above problems by providing data structure visualization and animation for user provided code.

IV. Architecture

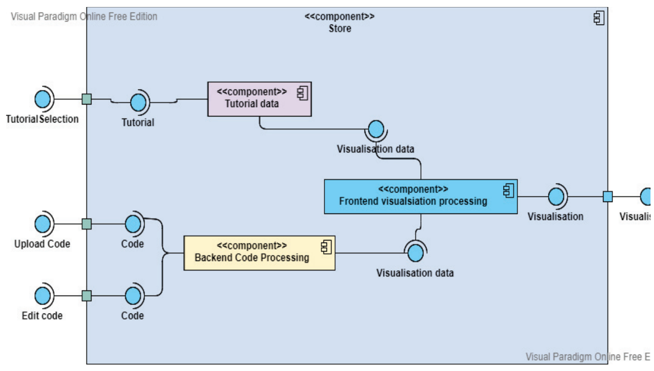

Visual Paradigm Online Free Edition
<<component>> Store
<<component>> Tutorial data
TutorialSelection
Tutorial
Visualisation data
<<component>> Frontend visualisation processing
Visualisation
Visualis
Upload Code
Code
<<component>> Backend Code Processing
Visualisation data
Edit code
Code
Visual Paradigm Online Free E

Fig. 3. UML Component diagram

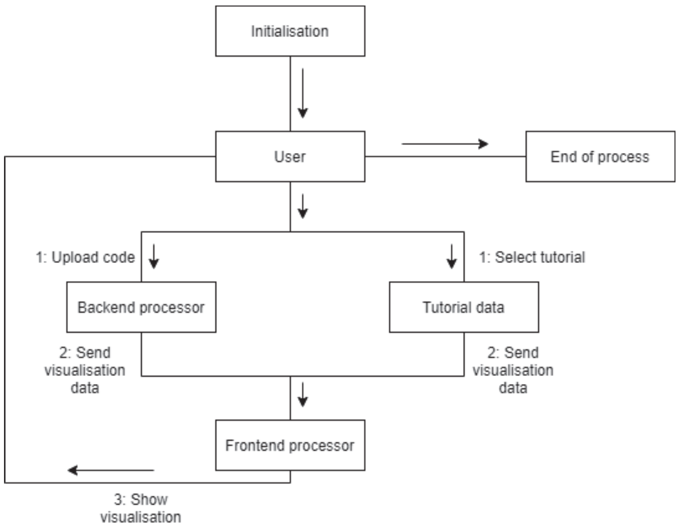


Fig. 4. UML Collaboration diagram

Code-Viz mainly consists of two parts: back-end processing and front-end processing. The back-end processing involves processing the input code and sending data required by the front-end in the format of JSON. The front-end processing uses this data from the back-end and generates data needed for visualisation on screen. As seen in Fig. 3 this data is used to render visualisation with the help of the Data Structure Visualization [4] library.

Fig. 4 shows the two paths a user can opt to either view the visualization for their code or view a preconfigured tutorial for a data structure or an algorithm.

V. Implementation

The back-end module is a Python program that uses language specific debuggers to generate the program trace in JSON format. The front-end processing is done using JavaScript and React JS framework is used for the UI. Data Structure Visualization [4] library has visualization related code, i.e. code to create, connect, move, delete shapes, and more, in JavaScript.

User given syntactically correct code is accepted via the front-end interface. Users can upload their code or write their code into an online editor provided with the front-end's interface. This code is then passed on to the backend.

The backend computation looks at the structure of the given code. It looks at the structs (C language), classes in Python etc. and tries to derive a relationship between the code and the possible data structures that could have been used in the code. All this is done using the language specific debugger like GNU DeBugger (GDB) for C and Python DeBugger (PDB) for Python. The backend process steps into the user given code using the debugger and generates a program trace which is eventually stored in JSON format. This program trace contains information about stack frames, variables, pointers, functions etc. used in the code.

The backend then combines all this into a single JSON file. This file has a predetermined format which helps in the visualization and line by line code animation on the front-end side.

This JSON file is then passed to the front-end where the file is parsed and objects corresponding to the variables in the code are rendered. The rendering is done only when a new object is made or the value of a variable changes or a pre-rendered object is deleted. The frontend uses animation functionality defined in the Data Structure Visualizations Library [4]. The JSON file consists of semantic information structured to aid visualization. Information is extracted from the file and sent to respective custom visualization functions that make use of the animation library.

VI. Example

The user is presented with an interface to submit code as seen in Fig. 5. The frontend sends the user-provided code to the backend, which processes it to generate a JSON file.

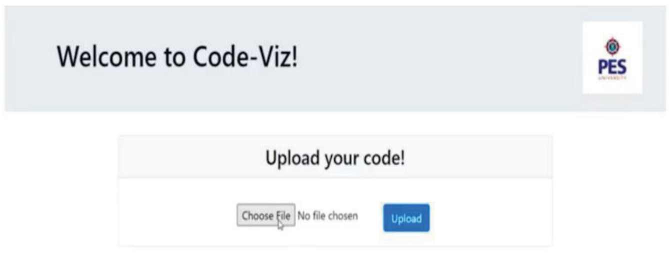
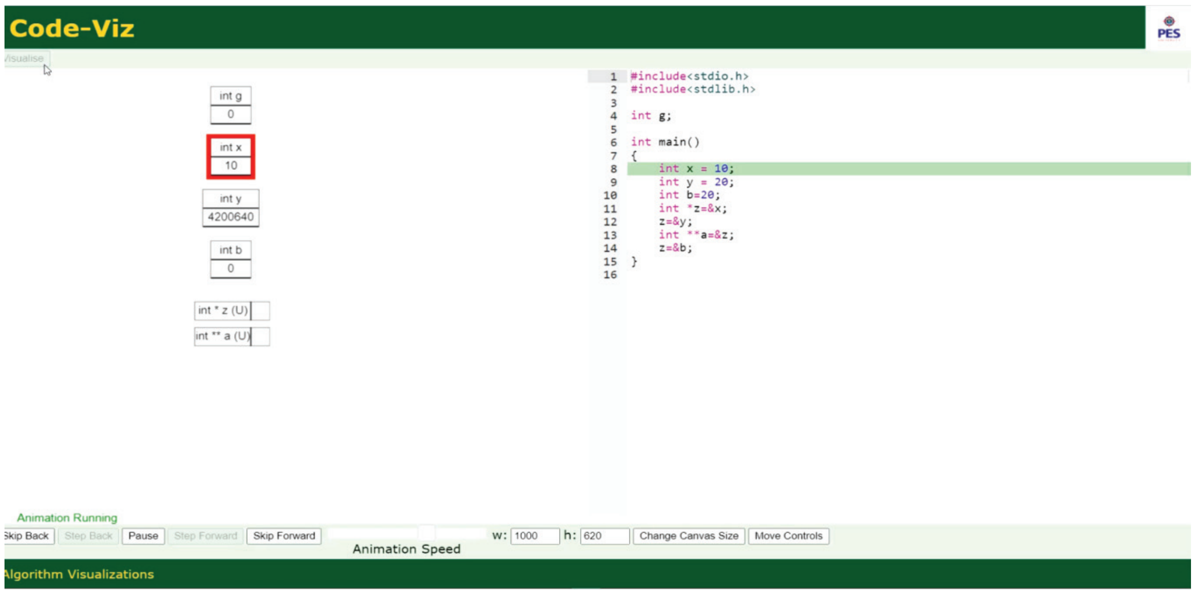


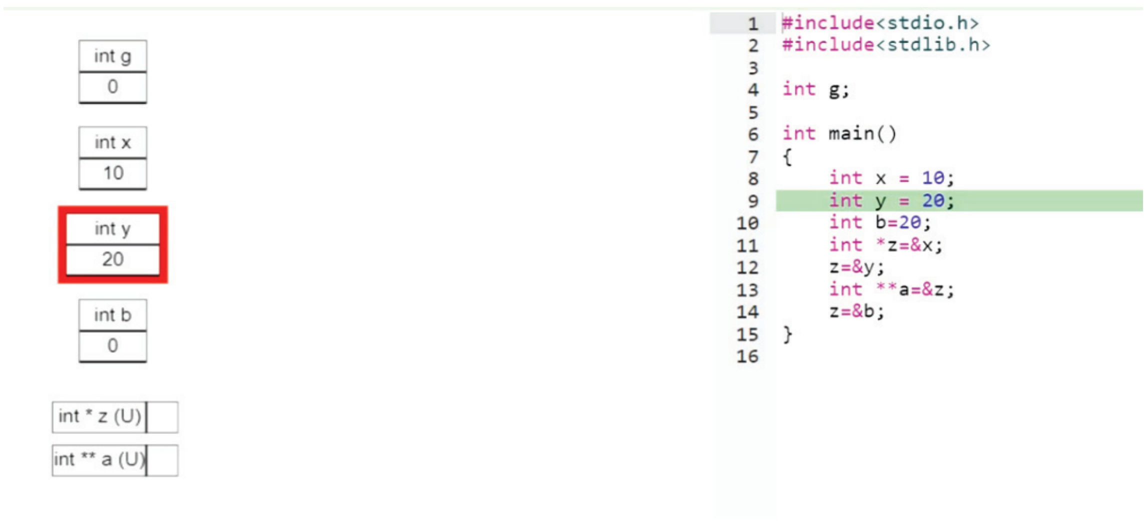
Fig. 5. User presented interface: user selects a C/Python file to upload, the selected file's contents are displayed before the user submits

Example 6.1 Visualization of Pointers and their movements in a C program



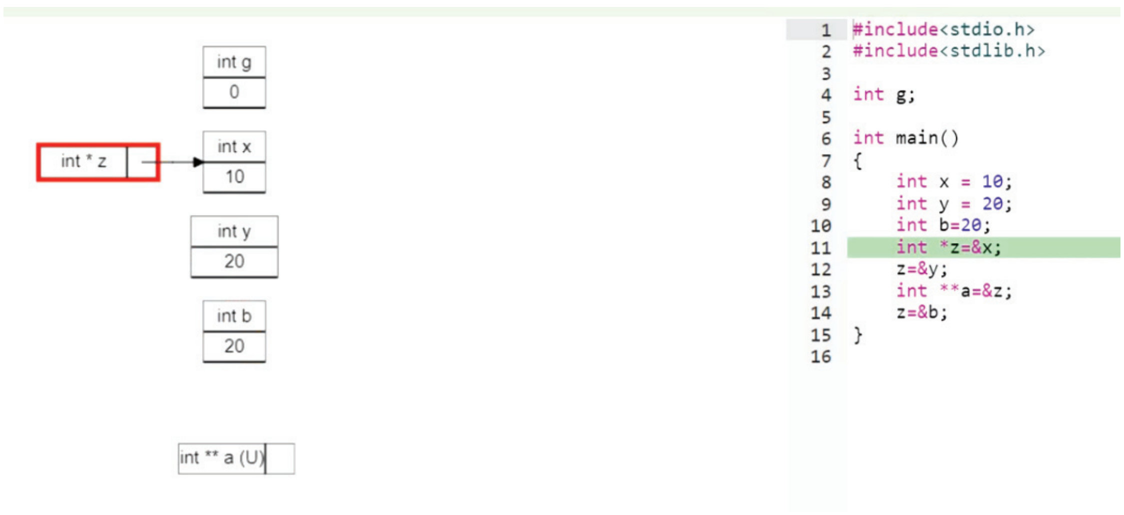
```
#include<stdio.h>
#include<stdlib.h>

int g;

int main()
{
    int x = 10;
    int y = 20;
    int b=20;
    int *z=&x;
    z=&y;
    int **a=&z;
    z=&b;
}
```

Fig. 6. Initial values of all variables/pointers in the program are shown and variable x is assigned a value of 10.



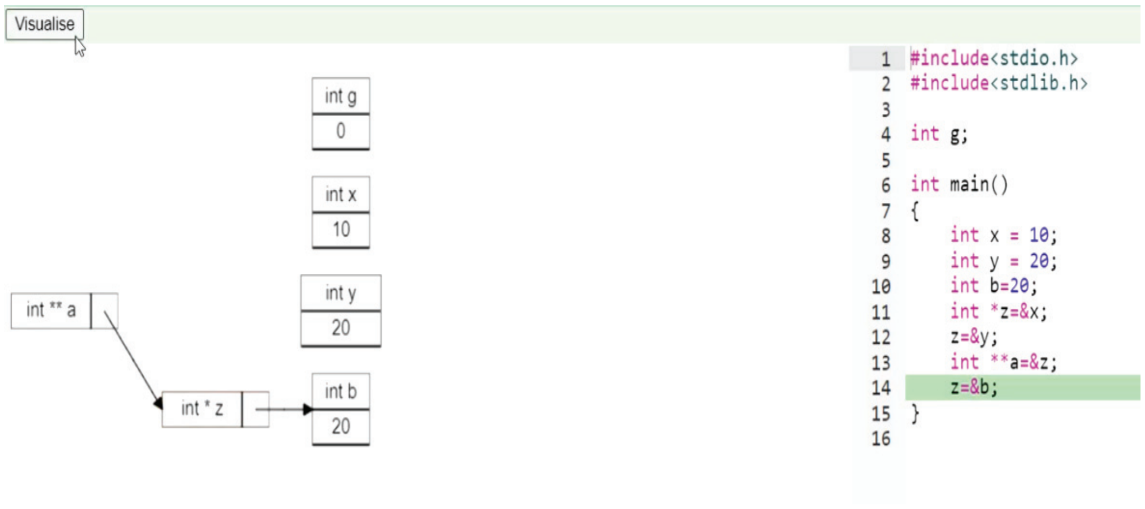
```
#include<stdio.h>
#include<stdlib.h>

int g;

int main()
{
    int x = 10;
    int y = 20;
    int b=20;
    int *z=&x;
    z=&y;
    int **a=&z;
    z=&b;
}
```

Fig. 7. Variable y is assigned a value of 20



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

int g;

int main()
{
    int x = 10;
    int y = 20;
    int b=20;
    int *z=&x;
    z=&y;
    int **a=&z;
    z=&b;
}
```

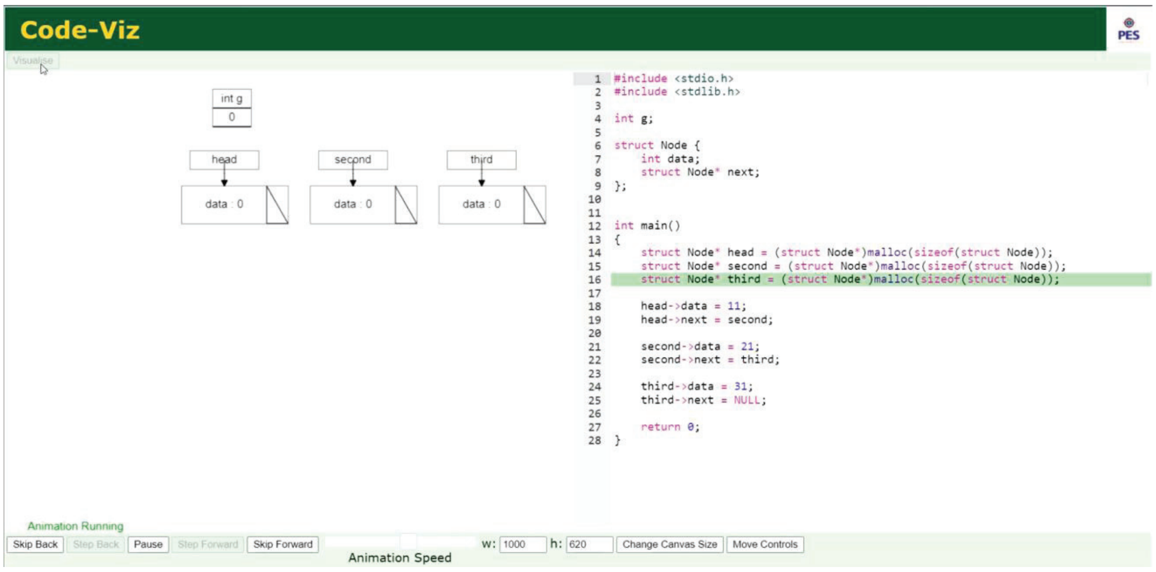
Fig. 8. Pointer z points to variable x



```c
#include<stdio.h>
#include<stdlib.h>

int g;

int main()
{
    int x = 10;
    int y = 20;
    int b=20;
    int *z=&x;
    z=&y;
    int **a=&z;
    z=&b;
}
```

Fig. 9. Pointer a points to z which is pointing to b

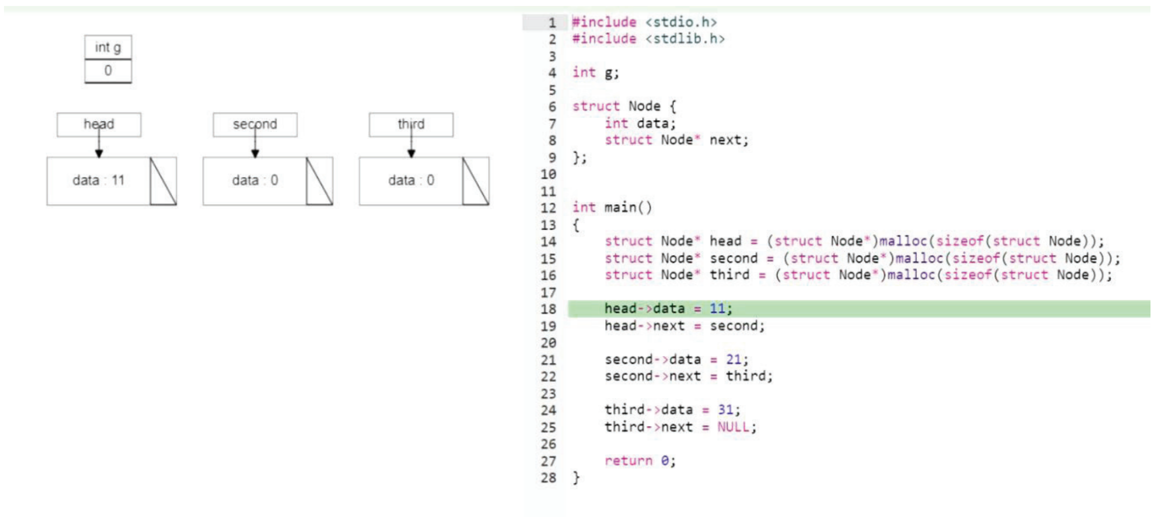
Example 6.2 Visualization for Linked Lists (insertion at end) in C



```c
#include <stdio.h>
#include <stdlib.h>

int g;

struct Node {
    int data;
    struct Node* next;
};


int main()
{
    struct Node* head = (struct Node*)malloc(sizeof(struct Node));
    struct Node* second = (struct Node*)malloc(sizeof(struct Node));
    struct Node* third = (struct Node*)malloc(sizeof(struct Node));

    head->data = 11;
    head->next = second;

    second->data = 21;
    second->next = third;

    third->data = 31;
    third->next = NULL;

    return 0;
}
```

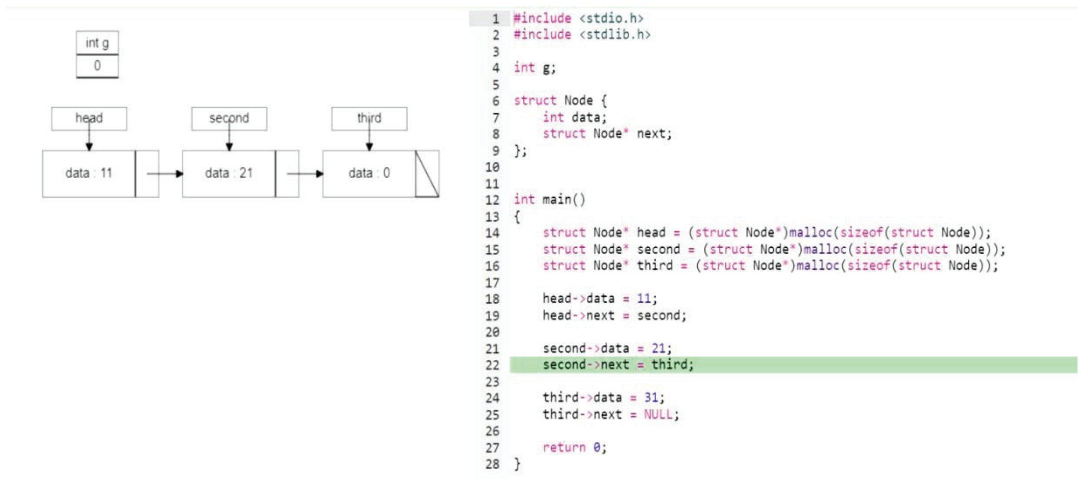
w: 1000 h: 620

Fig. 10. Initial allocation of memory for the C program with all the 3 nodes being initialized dynamically



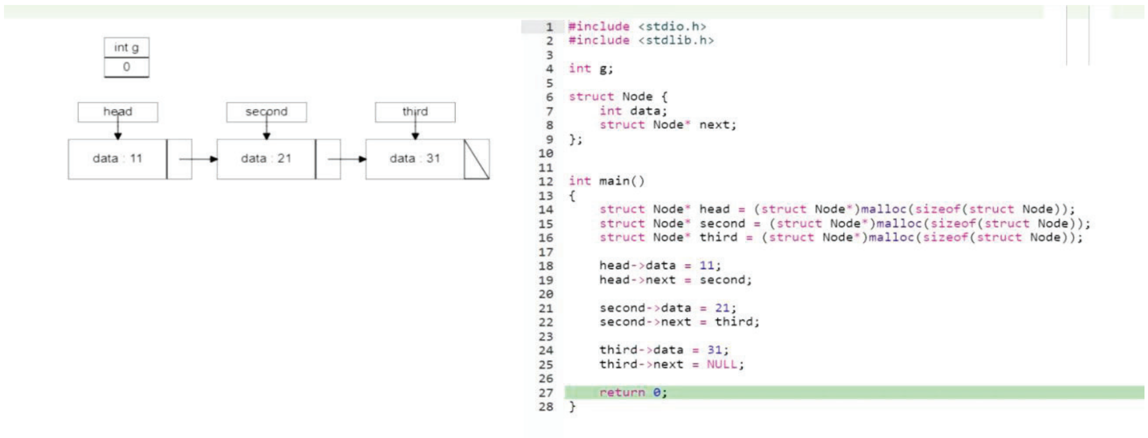
```c
#include <stdio.h>
#include <stdlib.h>

int g;

struct Node {
    int data;
    struct Node* next;
};


int main()
{
    struct Node* head = (struct Node*)malloc(sizeof(struct Node));
    struct Node* second = (struct Node*)malloc(sizeof(struct Node));
    struct Node* third = (struct Node*)malloc(sizeof(struct Node));

    head->data = 11;
    head->next = second;

    second->data = 21;
    second->next = third;

    third->data = 31;
    third->next = NULL;

    return 0;
}
```

Fig. 11. The data field of node ‘head’ is assigned 11.



```c
#include <stdio.h>
#include <stdlib.h>

int g;

struct Node {
    int data;
    struct Node* next;
};


int main()
{
    struct Node* head = (struct Node*)malloc(sizeof(struct Node));
    struct Node* second = (struct Node*)malloc(sizeof(struct Node));
    struct Node* third = (struct Node*)malloc(sizeof(struct Node));

    head->data = 11;
    head->next = second;

    second->data = 21;
    second->next = third;

    third->data = 31;
    third->next = NULL;

    return 0;
}
```

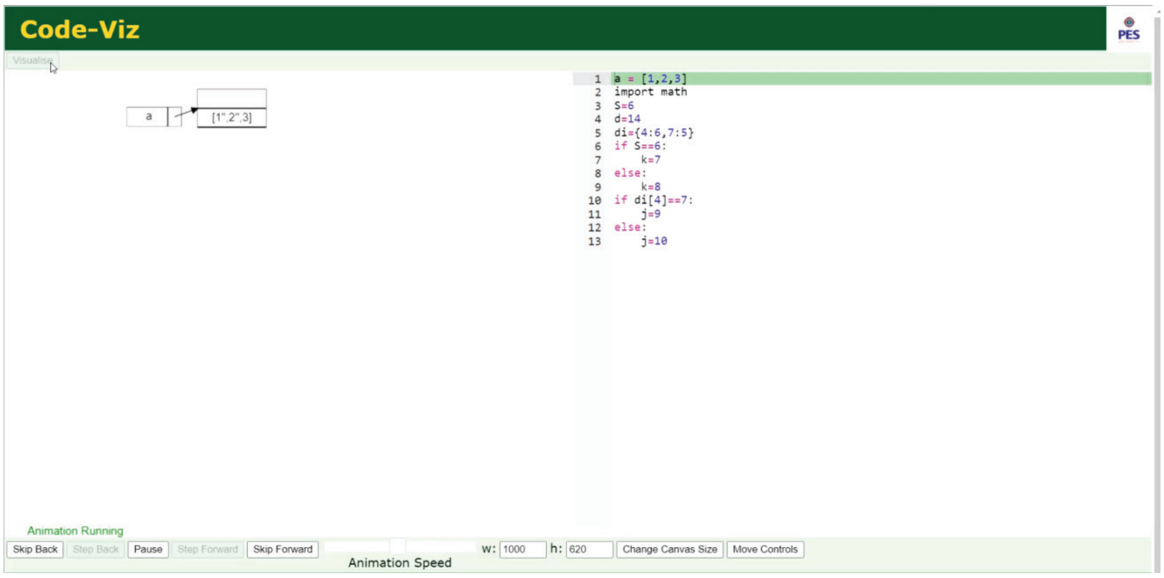
Fig. 12. The data field of node ‘second’ is assigned 21 and points to node ‘third’.



```c
#include <stdio.h>
#include <stdlib.h>

int g;

struct Node {
    int data;
    struct Node* next;
};


int main()
{
    struct Node* head = (struct Node*)malloc(sizeof(struct Node));
    struct Node* second = (struct Node*)malloc(sizeof(struct Node));
    struct Node* third = (struct Node*)malloc(sizeof(struct Node));

    head->data = 11;
    head->next = second;

    second->data = 21;
    second->next = third;

    third->data = 31;
    third->next = NULL;

    return 0;
}
```

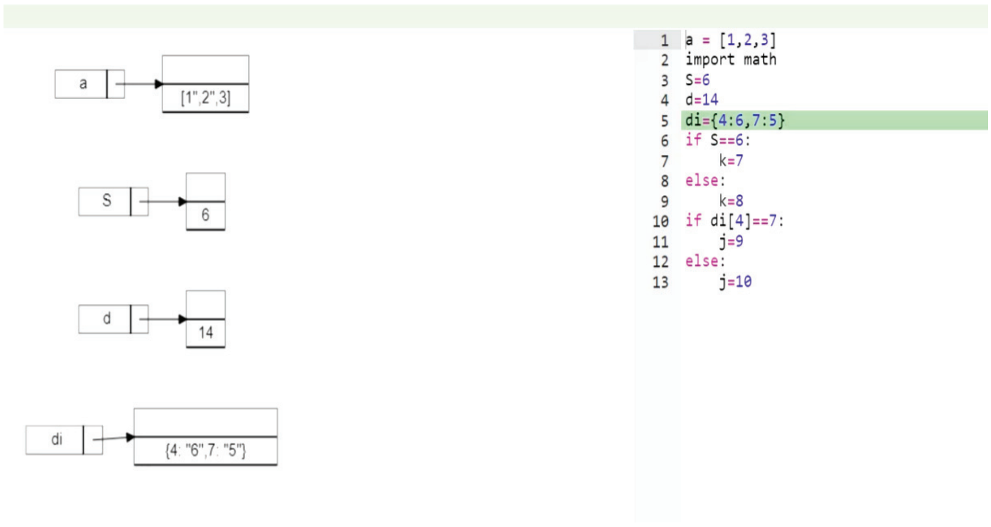
Fig. 13. End of execution.

Example 6.3 Visualization for a Python program



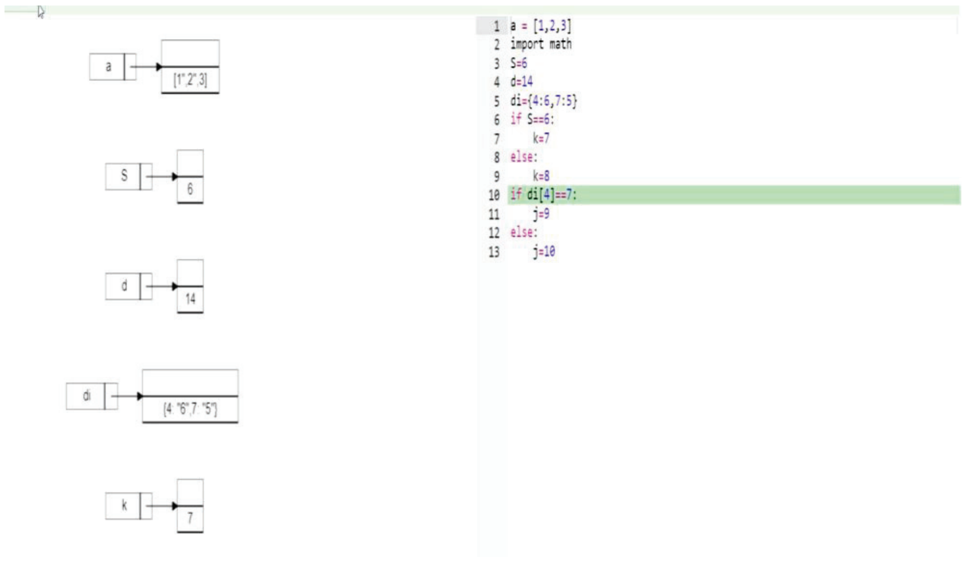
```python
a = [1,2,3]
import math
S=6
d=14
di={4:6,7:5}
if S==6:
    k=7
else:
    k=8
if di[4]==7:
    j=9
else:
    j=10
```

Fig. 14. Variable a is assigned a list.



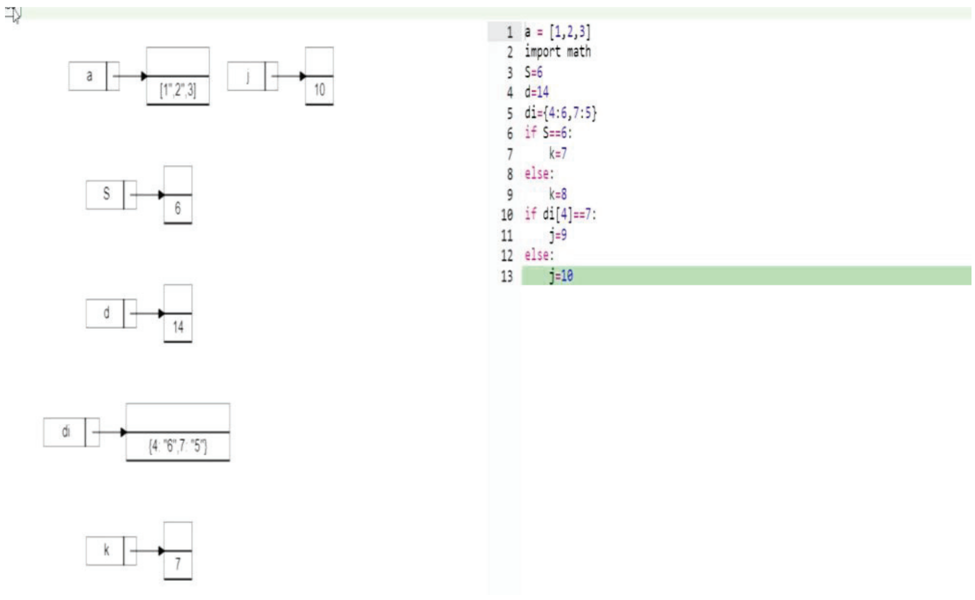
```
a = [1,2,3]
import math
S=6
d=14
di={4:6,7:5}
if S==6:
    k=7
else:
    k=8
if di[4]==7:
    j=9
else:
    j=10
```

Fig. 15. Python specific visualization where everything is considered as an object



```
a = [1,2,3]
import math
S=6
d=14
di={4:6,7:5}
if S==6:
    k=7
else:
    k=8
if di[4]==7:
    j=9
else:
    j=10
```

Fig. 16. Evaluation of if condition.



```
a = [1,2,3]
import math
S=6
d=14
di={4:6,7:5}
if S==6:
    k=7
else:
    k=8
if di[4]==7:
    j=9
else:
    j=10
```

Fig. 17. Evaluation of the else block.

VII. Detection Of Underlying Data Structures

The code that the back-end gets is analysed. The detection of data structures is language specific in the sense that for 'C' we need to look at structs used in the program, whereas for Python we need to look at classes or inbuilt data structures like heaps (heapq), lists etc. So based on the language of the code we receive, we try to detect the data structures. To identify linked lists in C, we see the structures used. We have started with a naive approach and checked if the struct contains a pointer to itself i.e whether it is a self referencing structure. We realized that a linked list node is the most commonly used self referencing structure with a single pointer in the structure definition. Similarly in Python, we look at classes or dictionaries used in the program and try to identify whether the underlying data structure is a linked list. Using this identified linked list, we can visualize various functions related to it.

VIII. Future Work

The front-end processing must be refactored to C++ or Python to reduce computation on the client's machine and for faster processing on the server. The numerous edge cases must be handled in the visualisation and a fool-proof visualisation logic must be developed to prevent overlapping of the drawn objects. Detection of data structures is a vast concept. As future work, we would like to support the detection of many more data structures like trees, graphs, etc. Moreover, we would also like to use an appropriate heuristic to detect the data structures accurately.

IX. Conclusion

Code-Viz aims to provide data-structure specific visualisations which will help the users in better understanding the data structures and algorithms. It has been developed as a learning tool and can be used by: teachers to help in teaching, by learners to better understand concepts and by any programmer to visualise any code. Debugging can also be made easier by seeing the flow of code and the corresponding visualizations. For example, consider a wrongful insertion in a linked list. The line-by-line visualization generated by the tool for the code could help the programmer identify his/her code mistakes and rectify them.

References

[1] Guo, Philip J , "Online python tutor: embeddable web-based program visualization for CS education.". ACM Technical Symposium on Computer science education, 2013.

[2] Heinsen Egan, M., & McDonald, C , An evaluation of SeeC: a tool designed to assist novice C programmers with program understanding and debugging. Computer Science Education, 1-34, 2020.

[3] Ryosuke Ishizue, et al. , PVC.js: visualizing C programs on web browsers for novices. Heliyon, Volume 6 Issue 4, 2020.

[4] David Galles, Data Structures Visualization https://www.cs.usfca.edu/~galles/visualization/source.html. 2011