Automatic Data Structures Visualization in IntelliJ IDEA

Grigorii Emdin*†, Dmitrii Kirkhmeier*, and Nikita Koval*
*JetBrains
†Ecole Polytechnique Fédérale de Lausanne

ABSTRACT Debugging code that involves complex data structures poses significant challenges. It is often difficult to track the flow of data through the data structures while debugging the code, particularly for novice programmers learning algorithms and data structures. A potential approach to simplify this process is to visualize the relevant data structures along with their current states during debugging. However, identifying the specific portions of code that correspond to these data structures presents a unique challenge. To address this, we propose an approach that visualizes the runtime state of key data structures - such as arrays, linked lists, and trees - during debugging by leveraging a large language model (LLM) to facilitate and guide the visualization process. Our current implementation, developed as a plug-in for IntelliJ IDEA, supports the highlighting of array indices and enforces correct element sequencing in linked lists. With this integration of the visualization into the debugging workflow, we aim to improve code comprehension and reduce debugging time and effort. Future work includes expanding support to tree structures and migrating from a remote to an embedded LLM. We plan to evaluate the effectiveness of our approach through user studies with novice programmers, measuring debugging time, bug detection accuracy, and perceived cognitive load.

KEYWORDS Debugging Tools, Program Visualization, IDE Plugins.

1. Introduction

Mastering foundational data structures and algorithms is a critical milestone for novice programmers, yet this process often proves challenging. When learning foundational data structures such as arrays, linked lists, and trees, novice learners often experience difficulty in mentally mapping the operations of these structures by analyzing the code, as demonstrated in (Zingaro et al. 2018).. This difficulty is further compounded when learners have to track how data flows through these structures and transforms during program execution. As a result, the process of studying data structures and algorithms becomes frustrating. Real-time visualization of data structures during debugging offers a powerful solution to alleviate this cognitive burden.

Visual representations that reflect the current state of data structures allow learners to directly observe changes, identify errors, and better understand algorithmic behavior. Prior studies (e.g., (Scott Grissom 2003)) have shown how visual aids can improve comprehension, yet implementing such solutions is not trivial. One of the major challenges lies in identifying the specific portions of code associated with relevant data structures and their corresponding elements. Without effective identification, visualizations can easily become cluttered and overwhelming for learners. Identification of relevant portions of code becomes even more challenging when learners start creating their own implementations of the data structures.

To address these limitations, we propose a novel approach that leverages the capabilities of large language models (LLMs) to automate the process of generating precise, real-time visualizations. Our method uses an LLM to analyze source code and identify the elements and operations associated with popular data structures such as arrays, linked lists, and trees. By integrating these visualizations into the debugging workflow, learners can track indices, element order, and structural changes within these data structures more effectively. Our ultimate objective is to improve code comprehension, streamline debugging workflows, and enhance the learning experience for novice pro-

JOT reference format:
Grigorii Emdin, Dmitrii Kirkhmeier, and Nikita Koval. *Automatic Data Structures Visualization in IntelliJ IDEA*. Journal of Object Technology. Vol. 25, No. 3, 2026. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2026.25.3.a7

grammers.

This approach has been implemented as a plug-in for IntelliJ IDEA, allowing seamless integration into existing development environments and educational infrastructures such as JetBrains Academy(JetBrains 2025). The current implementation targets Java/Kotlin code and supports arrays and linked lists, with future efforts focused on extending functionality to trees and transitioning from a remote LLM to an embedded one for improved efficiency and accessibility.

Through this work, we aim to make learning data structures and algorithms more intuitive and enjoyable for novice programmers, while contributing to the development of intelligent debugging tools that bridge the gap between theoretical understanding and practical coding skills. The following sections detail our methodology, implementation status, and future directions, showcasing the potential of combining LLM capabilities with debugging workflows.

2. Approach

In our study, we used a large language model (LLM) to identify and categorize data structures in source code into predefined or inferred types based on their usage and semantics. This approach provides an alternative to the more conventional approach of parsing the Abstract Syntax Tree (AST). The AST represents the hierarchical structure of the source code, allowing for performing static analysis of programs. While AST-based analysis excels at recognizing standard library data structures - thanks to their well-documented, predictable patterns - the LLM approach allows for a broader and potentially more flexible interpretation of code beyond syntactic structure.

Working with custom class hierarchies - often created by students learning data structures - presents more substantial challenges for AST-based methods. Students frequently design bespoke implementations of fundamental structures such as linked lists, binary trees, or hash tables, often employing unique naming conventions or structural variations that deviate from standard implementations. As a result, simple syntactic pattern-matching may fail to accurately detect these custom structures. Overcoming this requires more advanced strategies, such as analyzing inter-method relationships, inferring functionality from usage patterns, or developing heuristics to capture structural intent.

While parsing the AST offers certain advantages, such as precise control over syntax analysis, it lacks the ability to capture high-level contextual understanding that an LLM can provide. Although effective for static analysis of standardized codebases, AST-based approaches struggle with the variability found in student-authored code and often demand ongoing effort to develop and maintain analysis rules.

LLMs, on the other hand, are trained to identify and extrapolate patterns, reducing the need for manually crafted and maintained heuristics. Their ability to recognize contextual patterns and relationships gives them a unique advantage in extracting data structure information from code. These models are trained on a large amount of source code data and able to capture different aspects of programming languages such as syntax, conventions, coding style, and recognize the many ways in which programmers express similar programming constructs.

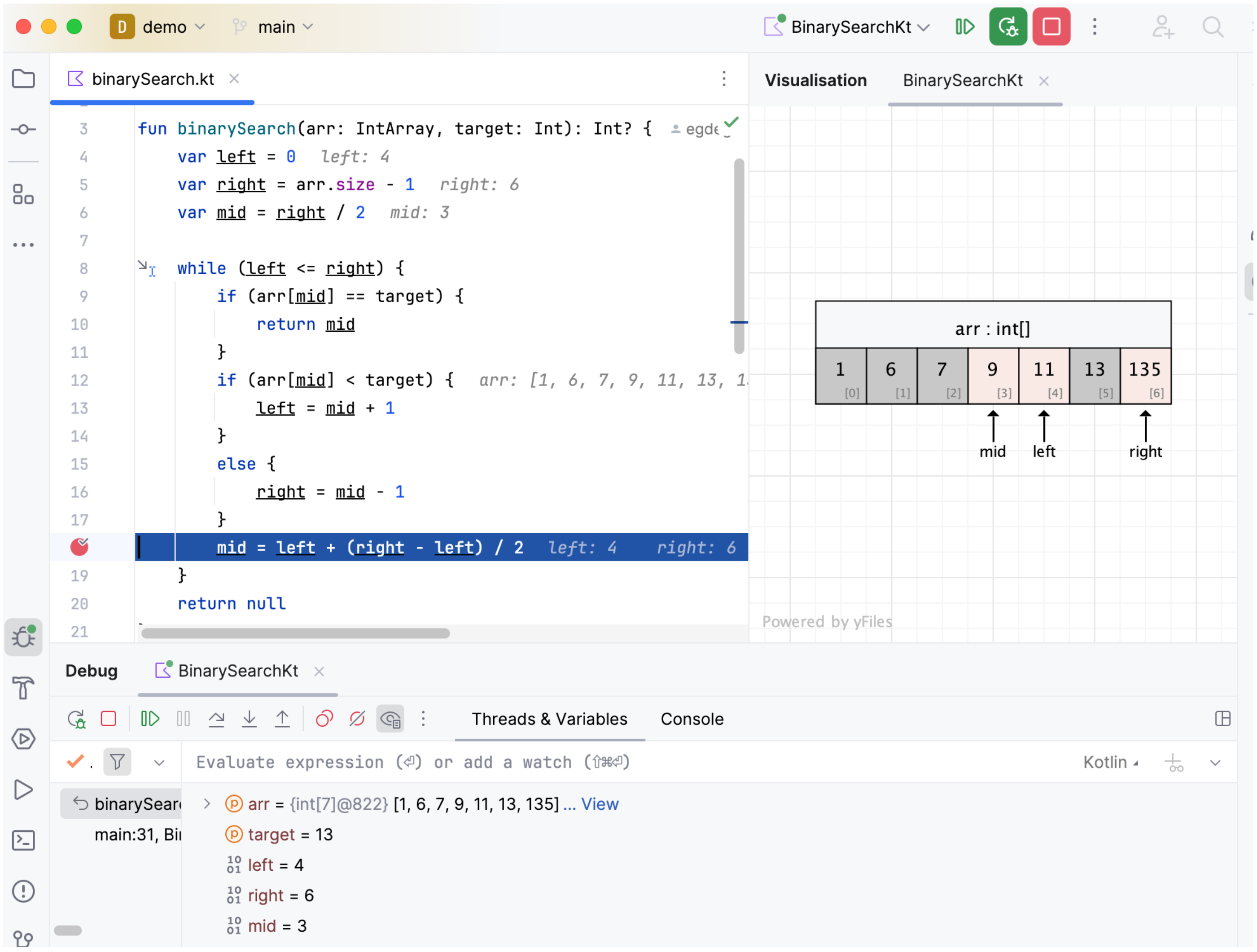
Building on the strengths of LLMs in interpreting code semantics and structure, we next describe how these capabilities are harnessed within our system. Rather than using the LLM as a generic tool for code comprehension, our program integrates it into a focused workflow designed to extract, analyze, and visualize data structures in real-time. The LLM is tailored to operate alongside a debugging environment, leveraging both the static code context and runtime state information to identify relevant data structures and their associated properties. This information is obtained using the IntelliJ IDEA plugin API, which provides seamless integration with the debugger for context extraction. The collected data, along with the relevant source code, is sent to the LLM with a prompt designed to identify data structures and supplementary data.

Prompt engineering plays a critical role in ensuring the accuracy and usability of LLM responses. Among the available prompting techniques, we selected *Few-Shot Prompting*, which has been shown to improve response accuracy without requiring task-specific model fine-tuning, as demonstrated in (Brown et al. 2020). This approach also allows us to standardize LLM output by encouraging structured, machine-readable responses. Other prompting techniques, such as *Chain-of-Thought* or *ReAct*, while useful in certain scenarios, are less suitable for this task due to difficulties in generating consistently structured responses, which complicates automated parsing and processing.

Our prompt typically includes a structured task description, followed by K-shot examples, where K is set between 3 and 5 to balance context window limitations and information sufficiency. The prompt specifies the task requirements, selection rules, and output format. Separate prompts are created for each data structure type (e.g., arrays, linked lists, and trees). This modular approach confers several advantages: (1) each prompt can be fine-tuned and optimized individually for better task-specific performance, (2) the overall complexity of each prompt is reduced, simplifying model interpretation and response parsing, and (3) additional data structures can be incorporated into the system with minimal modifications to existing prompts. Once the LLM returns its response, the output is parsed and used to update an internal representation of the data structures in the program. These representations are visualized through the IntelliJ IDEA plugin's built-in diagram API, presenting users with clear and interactive diagrams that assist in understanding the program's data flow and structure.

3. Current Status

At its current stage, our plugin supports the visualization of arrays, matrices, and linked lists (both singly and doubly linked). The plugin integrates seamlessly with the Integrated Development Environment (IDE)'s debugging system, automatically activating upon a debugging breakpoint. Once the breakpoint is triggered, the plugin initiates the visualization process, enabling users to gain insights into variable states and data structures at runtime. Below, we discuss the visual representation provided by the plugin for each supported data structure, along



```
fun binarySearch(arr: IntArray, target: Int): Int? {
    var left = 0   left: 4
    var right = arr.size - 1   right: 6
    var mid = right / 2   mid: 3

    while (left <= right) {
        if (arr[mid] == target) {
            return mid
        }
        if (arr[mid] < target) {   arr: [1, 6, 7, 9, 11, 13, 1
            left = mid + 1
        }
        else {
            right = mid - 1
        }
        mid = left + (right - left) / 2   left: 4     right: 6
    }
    return null
```

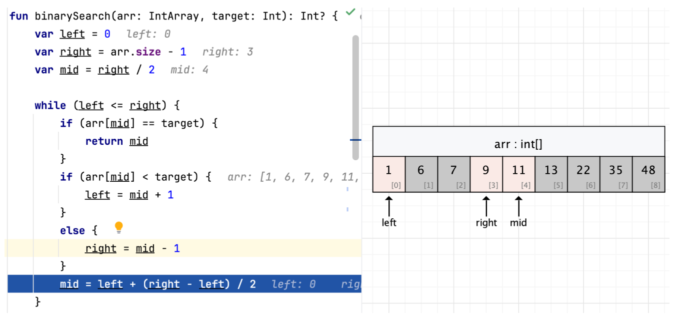
```
binarySear
main:31, Bi

arr = {int[7]@822} [1, 6, 7, 9, 11, 13, 135] ... View
target = 13
left = 4
right = 6
mid = 3
```

Figure 1 Visualization of a binary search state in the debug mode with our Visual Debugger plugin in IntelliJ IDEA.

with accompanying examples.

3.1. Arrays

For arrays, the visualization displays the array itself along with all variables pointing to specific indices of the array. This feature allows users to efficiently track the behavior of array indices throughout program execution and simplifies debugging operations. As shown in Figure 2, this approach highlights the movement and interaction of indices within an array structure.



```
fun binarySearch(arr: IntArray, target: Int): Int? {
    var left = 0   left: 0
    var right = arr.size - 1   right: 3
    var mid = right / 2   mid: 4

    while (left <= right) {
        if (arr[mid] == target) {
            return mid
        }
        if (arr[mid] < target) {   arr: [1, 6, 7, 9, 11,
            left = mid + 1
        }
        else {
            right = mid - 1
        }
        mid = left + (right - left) / 2   left: 0     rig
    }
```

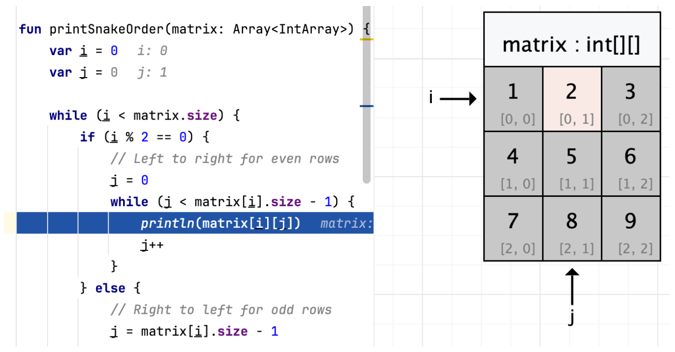
Figure 2 Visualization of an array within the plugin during debugging of binary search algorithm, showcasing references to specific indices for better tracking.

3.2. Matrices

Similar to arrays, matrices are visualized with row and column indices to provide clarity regarding their structure and individual element positions. This representation, illustrated in Figure 3, is designed to assist developers in debugging algorithms that manipulate two-dimensional data, such as those involved in matrix arithmetic or grid-based computations.

3.3. Linked Lists

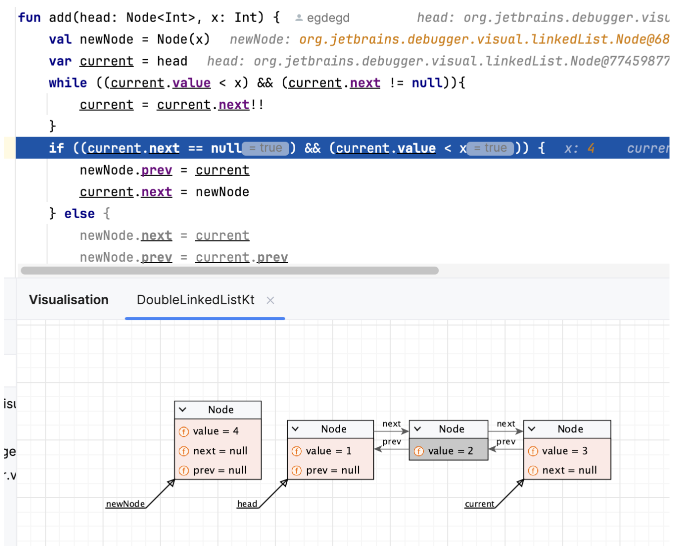
For linked lists, the visualization focuses on the connections between nodes, representing them as directed pointers that graphically illustrate how the nodes are linked. This visualization simplifies the identification of issues such as incorrect or missing connections, which are common during the implementation of these data structures. An example of a doubly linked list visualization is shown in Figure 4.



```
fun printSnakeOrder(matrix: Array<IntArray>) {
    var i = 0   i: 0
    var j = 0   j: 1

    while (i < matrix.size) {
        if (i % 2 == 0) {
            // Left to right for even rows
            j = 0
            while (j < matrix[i].size - 1) {
                println(matrix[i][j])   matrix:
                j++
            }
        } else {
            // Right to left for odd rows
            j = matrix[i].size - 1
```

1 [0, 0]	2 [0, 1]	3 [0, 2]
4 [1, 0]	5 [1, 1]	6 [1, 2]
7 [2, 0]	8 [2, 1]	9 [2, 2]

Figure 3 Visualization of a matrix, annotated with row and column indices during debugging of a snake order traversal.



```
fun add(head: Node<Int>, x: Int) {  egdegd          head: org.jetbrains.debugger.visu
    val newNode = Node(x)   newNode: org.jetbrains.debugger.visual.linkedList.Node@68
    var current = head   head: org.jetbrains.debugger.visual.linkedList.Node@77459877
    while ((current.value < x) && (current.next != null)){
        current = current.next!!
    }
    if ((current.next == null = true ) && (current.value < x = true )) {   x: 4      curre
        newNode.prev = current
        current.next = newNode
    } else {
        newNode.next = current
        newNode.prev = current.prev
```

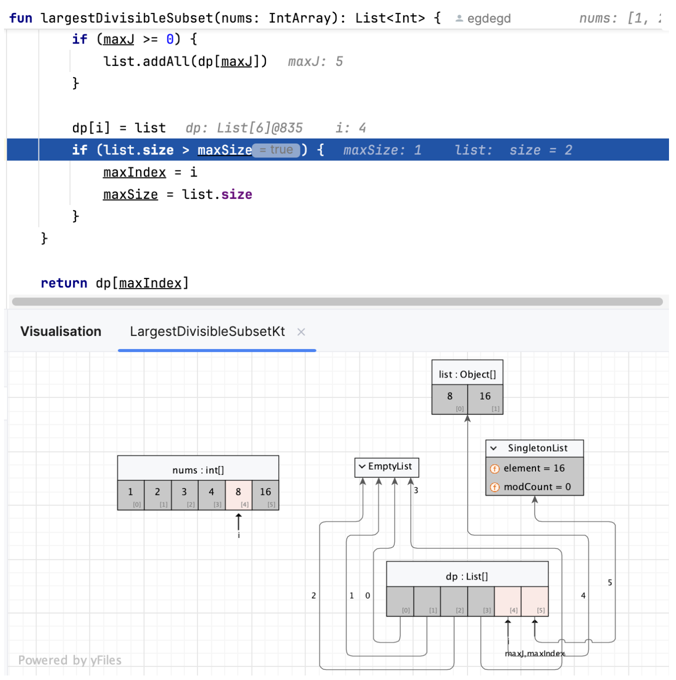
Figure 4 Visualization of a doubly linked list, demonstrating the connections between nodes to aid in debugging logical errors in the structure.

3.4. Limitations

While the plugin significantly aids in visualizing data structures, it is not without limitations. The current version of the tool can become cluttered when visualizing complex or large-scale structures, as shown in Figure 5. The inherent issue of inaccuracies and reasoning failures in large language models (LLMs), collectively referred to as "hallucinations," is also a well-documented problem. Addressing these limitations, such as improving scalability and simplifying representations for larger datasets, remains part of our planned future work.

4. Future Work

To evaluate the effectiveness of our real-time data structure visualization plugin, we plan to conduct a controlled user study with novice programmers. Participants will be randomly assigned to two groups: one group will use our visualization-enhanced debugger implemented as a plugin for IntelliJ IDEA, while the other group will use IntelliJ's standard debugger without additional visual support.



```
fun largestDivisibleSubset(nums: IntArray): List<Int> {  egdegd          nums: [1, 
        if (maxJ >= 0) {
            list.addAll(dp[maxJ])   maxJ: 5
        }

        dp[i] = list   dp: List[6]@835     i: 4
        if (list.size > maxSize = true ) {  maxSize: 1     list:  size = 2
            maxIndex = i
            maxSize = list.size
        }
    }

    return dp[maxIndex]
```

Figure 5 Example of a cluttered visualization when debugging more complex set of data structures, highlighting a key area for improvement.

During the study, participants will debug short algorithmic problems involving arrays, linked lists, and trees. We aim to compare the groups based on multiple metrics: the time taken to find bugs, the accuracy of bug detection, the quality of their written algorithm explanations, and their self-reported cognitive load, as measured by the NASA Task Load Index (NASA-TLX) (Hart & Staveland 1988).

Our hypothesis is that students using real-time visualizations will identify bugs more quickly, detect them more accurately, and produce clearer algorithm descriptions, while also reporting lower cognitive load. This study will allow us to rigorously assess whether dynamic visual feedback on key data structures can significantly improve code comprehension and debugging performance in an educational context. Based on the study outcomes, we plan to extend our system to support additional data structures commonly used in algorithms, allowing us to cover a broader range of programming concepts.

Improving the quality of the LLM responses - both in terms of accuracy and speed - remains a key priority. One promising direction for enhancing speed is transitioning from a remote LLM to a local model, which would require training a lightweight version optimized for this task. Another direction for future research involves enhancing the accuracy of LLM responses through model-agnostic or model-specific validation methods to mitigate occurrences of "hallucinations." This could include optimizing LLM prompts, post-processing model outputs by comparing them with prior responses, and similar strategies. Using task-specific probes, as demonstrated in (Orgad et al. 2024), represent one potential technique to address this class of problems, although certain limitations remain in their application.. Addressing complex or large-scale visualizations represents another important area for future research. This will require investigation into techniques for aggregation, filtering, and the optimal distribution of visualized components on the screen.

Yet another avenue for future work is expanding the intended audience. While our current focus is on beginners learning algorithms, the tool could be adapted to assist all novice programmers, regardless of their specialization. Furthermore, although our implementation currently targets Java/Kotlin within IntelliJ IDEA, the underlying approach can be generalized to support other programming languages independently of the development environment, opening the door to broader adoption in educational settings.

5. Related Work

Visual debugging has been studied extensively for decades (Hanson & Korn 1997; Jerding & Stasko 1994; Mukherjea & Stasko 1994) and several tools are actively used today for visualizing program execution during debugging.

The *Data Display Debugger* (DDD) (GNU Project 1995) is one of the earliest tools to introduce graphical data visualizations, presenting program memory as box-and-pointer diagrams. DDD operates as a standalone graphical front-end to command-line debuggers like the *GNU Project Debugger* (GDB) (GNU Project 1986) and visualizes the full memory state. However, it offers limited guidance on which parts of the memory are most relevant to current execution, making it difficult for beginners to focus.

JIVE (Jayaraman et al. 2010) is an Eclipse plugin that provides detailed object diagrams and compacted sequence diagrams during Java program execution. It supports advanced features such as reverse stepping, execution history queries, and property checking. At the same time, it assumes that users can interact with complex diagrams and formulate queries, which may overwhelm novice programmers.

The *Java Visualizer* plugin by Lipsitz (Lipsitz 2022) integrates into IntelliJ IDEA to assist in teaching. It visualizes the program's call stack and heap across debugging steps. While effective in exposing runtime structures, it visualizes all accessible heap objects, often resulting in cluttered and dense diagrams even for simple cases.

The *Visual Debugger* by Kräuter et al. (Krauter et al. 2024; Kräuter et al. 2022) focuses on improving usability by integrating object diagram visualizations directly into IntelliJ IDEA's debugging workflow. It adds features like dynamic loading of variables, change highlighting, and a debug history. However, it primarily visualizes all stack frame variables and related objects without semantic filtering of what is most relevant to the current step in execution.

The *VS Code Debug Visualizer* by Dieterichs (Dieterichs n.d.) enables users to visualize arbitrary expressions during debugging by converting them into structured JSON objects. While the tool is powerful and flexible, it requires users to manually define expressions or implement custom data extractors. This approach works well for experienced developers, but it demands additional effort and technical expertise that may not be feasible for beginners.

While these tools provide valuable support for debugging, they often visualize either full program states or require manual interaction, making them less suited for beginners. In contrast, our approach focuses on simplifying program comprehension by automatically deciding which objects should be visualized and highlighting the most critical elements of the program state, such as the currently accessed array index or the active node in a linked list. We leverage a lightweight large language model (LLM) to perform this analysis during debugging. We believe that this approach minimizes the cognitive load for students by removing unnecessary information and emphasizing only the key aspects of the program's behavior. Additionally, by concentrating on specific data structures, we are able to produce clearer and more effective visualizations. For example, in the case of doubly linked lists, we query the LLM to determine the head and tail nodes, allowing us to lay out the list in a clean and readable way, whereas generic visualizations often result in cluttered graphs due to recursive links between nodes.

6. Conclusion

We have introduced a novel approach to data structure visualization during debugging that leverages large language models (LLMs) to semantically identify and render relevant program elements in real time. Our key contribution is a plugin for IntelliJ IDEA that integrates these visualizations into the debugging workflow, currently supporting arrays, matrices, and linked lists.

The novelty of our work lies in using an LLM to drive the visualization process - automatically determining which structures to visualize and how to highlight them meaningfully. While there are current limitations, such as handling complex or large-scale structures and reliance on a remote LLM, our early results show that this approach is both feasible and promising. We believe that this plugin has the potential to significantly improve the debugging experience for students, reduce cognitive load, and help bridge the gap between theoretical knowledge and practical coding skills.

References

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... Amodei, D. (2020). Language models are few-shot learners. In *Proceedings of the 34th international conference on neural information processing systems.* Red Hook, NY, USA: Curran Associates Inc.

Dieterichs, H. (n.d.). *Debug visualizer.* https://marketplace.visualstudio.com/items?itemName=hediet.debug-visualizer. (Accessed: 2025-04-25)

GNU Project. (1986). *Gdb: The gnu debugger.* https://www.sourceware.org/gdb/. (Accessed: 2025-05-02)

GNU Project. (1995). *Ddd - the data display debugger.* https://www.gnu.org/software/ddd/. (Accessed: 2025-04-25)

Hanson, D. R., & Korn, J. L. (1997). A simple and extensible graphical debugger. In *Proceedings of the annual conference on usenix annual technical conference* (p. 13). USA: USENIX Association.

Hart, S. G., & Staveland, L. E. (1988). Development of nasa-tlx (task load index): Results of empirical and theoretical research. *Human mental workload*, *1*(3), 139–183.

Jayaraman, B., Czyz, J. K., & Lessa, D. (2010). Jive: A pedagogic tool for visualizing the execution of java programs.. Retrieved from https://api.semanticscholar.org/CorpusID:686884

Jerding, D. F., & Stasko, J. T. (1994). Using visualization to foster object-oriented program understanding.. Retrieved from https://api.semanticscholar.org/CorpusID:18843030

JetBrains. (2025). *Jetbrains academy.* Retrieved from https://www.jetbrains.com/academy/ (Online; accessed 02 May 2025)

Krauter, T., Stunkel, P., Rutle, A., & Lamo, Y. (2024). The visual debugger: Past, present, and future. In *Proceedings of the 1st acm/ieee workshop on integrated development environments* (p. 1–6). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3643796.3648443 doi: 10.1145/3643796.3648443

Kräuter, T., König, H., Rutle, A., & Lamo, Y. (2022, 10). The visual debugger tool. In (p. 494-498). doi: 10.1109/ICSME55016.2022.00066

Lipsitz, E. (2022). *Java visualizer plugin for intellij idea.* https://plugins.jetbrains.com/plugin/11512-java-visualizer. (Version 2.2.1, released April 4, 2022. Accessed: 2025-04-26)

Mukherjea, S., & Stasko, J. T. (1994, September). Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger. *ACM Trans. Comput.-Hum. Interact.*, *1*(3), 215–244. Retrieved from https://doi.org/10.1145/196699.196702 doi: 10.1145/196699.196702

Orgad, H., Toker, M., Gekhman, Z., Reichart, R., Szpektor, I., Kotek, H., & Belinkov, Y. (2024). Llms know more than they show: On the intrinsic representation of llm hallucinations. *ArXiv*, *abs/2410.02707*. Retrieved from https://api.semanticscholar.org/CorpusID:273098472

Scott Grissom, T. N., Myles F. McNally. (2003). Algorithm visualization in cs education: comparing levels of student engagement. In *Proceedings of the 2003 acm symposium on software visualization* (p. 87-94). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/774833.774846 doi: 10.1145/774833

Zingaro, D., Taylor, C. B., Porter, L., Clancy, M. J., Lee, C. B., Liao, S. N., & Webb, K. C. (2018). Identifying student difficulties with basic data structures. *Proceedings of the 2018 ACM Conference on International Computing Education Research*. Retrieved from https://api.semanticscholar.org/CorpusID:51956348

About the authors

Grigorii Emdin PhD student at EPFL working on distributed algorithms and systems. You can contact the author at grigorii.emdin@epfl.ch.

Dmitrii Kirkhmeier Developer at JetBrains Academy, working around all aspects of Software Engineering Education. You can contact the author at dmitrii.kirkhmeier@jetbrains.com or visit www.jetbrains.com.

Nikita Koval Researcher in concurrent algorithms and program analysis, contributor to Kotlin coroutines, and project lead of the Lincheck framework for testing concurrency on JVM. You can contact the author at nikita.koval@jetbrains.com or visit https://nikitakoval.org.