ELEMENTS OF COMPUTING SYSTEMS-II

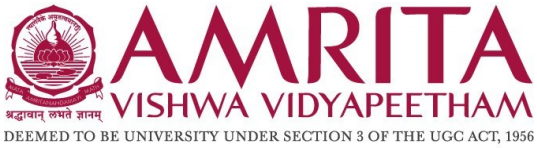
Standard Library in Jack Language

Submitted by

Akhilesh Mohanasundaram (CB.SC.U4AIE23111)
B V Dharsini Sri (CB.SC.U4AIE23115)
Chanjhana Elango (CB.SC.U4AIE23120)
Kaniska M (CB.SC.U4AIE23136)

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY
IN
CSE(AI)



Centre for Computational Engineering and Networking

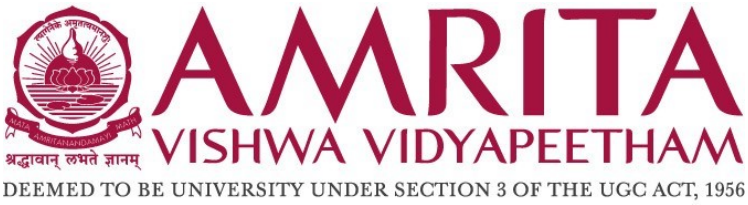
AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112 (INDIA)

JUNE - 2024

BONAFIDE CERTIFICATE



This is to certify that the thesis entitled "Standard Library in Jack Language" submitted by Akhilesh Mohanasundaram (CB.SC.U4AIE23111), B V Dharsini Sri (CB.SC.U4AIE23115), Chanjhana Elango (CB.SC.U4AIE23120), and Kaniska M (CB.SC.U4AIE23136) for the award of the Degree of Bachelor of Technology in the "CSE(AI)" is a bonafide record of the work carried out by them under our guidance and supervision at Amrita School of Artificial Intelligence, Coimbatore.

Ms. Sreelakshmi K
Project Guide

Dr. K.P.Soman
Professor and Head CEN

Submitted for the university examination held on June 2, 2024

Centre for Computational Engineering and Networking

AMRITA SCHOOL OF ARTIFICIAL INTELLIGENCE

AMRITA VISHWA VIDYAPEETHAM
COIMBATORE - 641 112 (INDIA)

DECLARATION

We, Akhilesh Mohanasundaram (CB.SC.U4AIE23111), B V Dharsini Sri (CB.SC.U4AIE23115), Chanjhana Elango (CB.SC.U4AIE23120) and Kaniska M (CB.SC.U4AIE23136) hereby declare that this thesis entitled **"Standard Library in Jack Language"**, is the record of the original work done by me under the guidance of Ms. Sreelakshmi K, Assistant Professor, Centre for Computational Engineering and Networking, Amrita School of Artificial Intelligence, Coimbatore. To the best of our knowledge this work has not formed the basis for the award of any degree/diploma/associate ship/fellowship/or a similar award to any candidate in any University.

Place: Coimbatore

Date: 02-06-2024

Signature of the Students

Acknowledgement

We would like to express our special thanks of gratitude to our teacher (MS. SREELAKSHMI K ma'am), who gave us the golden opportunity to do this wonderful project on the topic **Standard Library in Jack Language**, which also helped us in doing a lot of Research and we came to know about so many new things. We are thankful for the opportunity given. We would also like to thank our group members, as without their cooperation, we would not have been able to complete the project within the prescribed time.

Abstract

Jack is a simple, low-level object-oriented programming language. The language does not have a built-in standard library like other high-level languages (Java or Python). The goal of this project is to create custom implementations of fundamental data structures and other utility functions using the Jack programming language. This includes data structures like list, vector, queues, pqueues, matrices, binary search trees, hash tables, and sets, and utility functions that include sorting algorithms (bubble sort, insertion sort, selection sort, and quick sort), string manipulation (string comparison, toUppercase, toLowercase and String Reversal), and mathematical operations (gcd (greatest common divisor, exponentiation, finding the hypotenuse of a triangle and cube root finding function). By building this standard library, we enhance the language's capabilities and empower developers to work more efficiently.

Contents

List of Figures

List of Tables

1 Introduction

In the domain of low-level object-oriented programming, while the Jack language offers a streamlined paradigm for efficient coding, it lacks a comprehensive standard library typically found in more mature languages. To address this limitation, we propose the development of a custom standard library for Jack. This initiative aims to enrich Jack's functionality by implementing essential data structures and incorporating crucial utility functions for sorting algorithms , string manipulations, and mathematical operations. Through this project, we anticipate a significant expansion of Jack's capabilities, empowering developers with a robust toolkit that facilitates more efficient and effective programming practices within the Jack environment.

2 Methodology

2.1 Sorting Algorithms

2.1.1 Introduction

Efficient organization of data is paramount in computer science. Sorting algorithms provide the essential tools for arranging elements within a collection in a specific order, typically numerical or lexicographical (dictionary order). The standard library for Jack aims to empower developers with a robust suite of sorting algorithms. This introduction delves into the core concepts and considerations surrounding these algorithms, paving the way for a deeper exploration of their implementation within the Jack environment.

2.1.2 Bubble Sort

Bubble sort is a simple sorting technique that iterates through a list, repeatedly comparing adjacent elements and swapping them if they are in the wrong order. This process resembles bubbles rising to the surface of water, hence the name.

Algorithm

Initialization: Define array A with n elements, initialize integer variables i and j to 0, create an integer temp for swaps, and set a boolean swapped flag to false.

Outer Loop: Iterate a while loop as long as i is less than n - 1.

Inner Loop: Within the outer loop, iterate another while loop as long as j is less than n - 1 - i.

Comparison and Swap: If A[j] is greater than A[j + 1]: Swap A[j] and A[j + 1] using a temporary variable. Set swapped to true.

Early Termination: After the inner loop, if swapped is false, the list is sorted, and the algorithm terminates using return.

Iteration: Increment i by 1 to move to the next element in the outer loop.

Sorted List: Upon loop termination, A is sorted in ascending order.

Complexity of the Algorithm

Time Complexity of Bubble Sort Algorithm: O(n^2)

In the worst-case scenario, the algorithm needs to compare each element with all other elements in the list. This requires $n * (n - 1)/2$ comparisons, which is dominated by the n^2 term as n grows large. Additionally, there are n iterations in the outer loop. Even in the average case, the time complexity remains O(n^2).

2.1.3 Insertion Sort

Insertion Sort is a sorting algorithm that works by iteratively building a sorted sub-list at the beginning of an array. It compares each element with its sorted predecessors, ”shifting” it to its correct position within the sub-list. Imagine organizing playing cards in your hand, where you take each card and insert it in its rightful place among the already sorted cards.

Algorithm

Initialisation: The ‘insert‘ function of the InsertionSort class accepts two parameters: an array A of integers and an integer n representing the number of elements in array A. Within the function, several variables are declared, including i, j, and key as integer variables to control loops and store temporary values, and continueLoop as a boolean variable to control the inner loop.

Sorting Algorithm: To sort the array A, the function follows a step-by-step approach. It begins by initializing i to 1, allowing the function to start from the second element of the array. It then iterates through the array from the second element to the last. During each iteration, the function assigns the value of the current element to key and sets j to one position before the current element. Additionally, continueLoop is initialized to true for the inner loop.

Inner Loop: Within the inner loop, the function continues iterating until continueLoop becomes false. This loop is responsible for comparing the value of A[j] with key. If j becomes less than 0 during this comparison, the function sets continueLoop to false, breaking out of the loop. Otherwise, if A[j] is greater than key, the function shifts A[j] one position to the right and decrements j. If neither condition is met, continueLoop is set to false, and the loop terminates.

Placement of Key: After the inner loop concludes, the function places key in the correct position in the array, ensuring that the array remains sorted in non-decreasing order. Finally, i is incremented to move to the next element in the array, and the process repeats until all elements are sorted.

Output: Upon executing the algorithm, the array A is sorted in non-decreasing order, providing a sorted version of the input array as the output of the function.

Complexity of the Algorithm

Time Complexity: O(n^2)

In the worst case, each element might need to be compared with all the preceding elements in the sorted sub-list, leading to $n*(n-1)/2$ comparisons (dominated by n^2 as n grows large). This occurs when the array is in descending order. The average-case complexity is also O(n^2).

2.1.4 Selection Sort

Selection Sort is a sorting algorithm that works by repeatedly finding the minimum element from the unsorted portion of the list and swapping it with the first element in that unsorted portion. Imagine selecting the shortest person in a line and moving them to the front, then repeating this process for the remaining people.

Algorithm

Initialization:

Define array A with n elements to be sorted. Initialize integer variables i to 0 (referencing the first element) and j to i + 1 (referencing the element after i). Initialize an integer minIndex to store the index of the minimum element found so far. Initialize an integer temp for temporary storage during swaps.

Outer Loop:

Iterate a while loop as long as i is less than n. This ensures all elements are processed.

Inner Loop (Finding Minimum):

Initialize minIndex to the current index i. This assumes the first element in the unsorted portion is the minimum initially. Iterate another while loop as long as j is less than n. This loop iterates through the unsorted portion. Within the inner loop, if A[j] is less than A[minIndex]: Update minIndex to store the index of the new minimum element found.

Swapping:

After the inner loop completes, minIndex points to the index of the minimum element within the unsorted portion. Swap the element at A[i] (first element in the unsorted portion) with the element at minIndex using a temporary variable temp.

Iteration:

Increment i by 1 to move to the next element in the unsorted portion. This effectively reduces the unsorted portion by one element.

Sorted List:

Upon exiting the outer loop, the array A will be sorted in ascending order.

Complexity of the Algorithm

Time Complexity: O(n^2)

In the worst case (array in descending order), each element might need to be compared with all remaining elements to find the minimum. This leads to n*(n-1)/2 comparisons (dominated by n^2 as n grows large). The average-case complexity is also O(n^2).

While Selection Sort has the same time complexity as Bubble Sort and Insertion Sort, it can be slightly more efficient in practice due to fewer swaps being required on average.

2.1.5 Quick Sort

Quick Sort is a divide-and-conquer sorting algorithm that excels in efficiency for large datasets. It works by selecting a pivot element, rearranging the list such that elements less than the pivot are placed before it, and elements greater than the pivot are placed after it. The algorithm then recursively sorts the sub-lists on either side of the pivot. Imagine dividing a group of people by height, then sorting the shorter and taller groups independently.

Algorithm Description

Base Case:

If the low index (low) is greater than or equal to the high index (high), the sub-list has zero or one element and is already sorted. Simply return.

Pivot Selection:

Choose an element from the sub-list (often the last element) as the pivot (pivot).

Partitioning:

Initialize variables: i is one less than the low index (i = low - 1). It keeps track of the element before the swap position. j starts at the low index (j = low). It iterates through the sub-list. Iterate a while loop as long as j is less than high (excluding the

pivot): If the element at A[j] is greater than the pivot: Set a flag (condition) to false. This indicates a swap is not needed. If the condition is true (element at j is less than or equal to the pivot): Increment i by 1 to find the swap position. Swap the element at i + 1 with the element at j using a temporary variable temp. Increment j to move to the next element in the sub-list.

Pivot Placement:

After the loop completes, swap the pivot element with the element at i + 1. This positions the pivot in its final sorted place.

Recursive Sorting:

Recursively call the sort function on the sub-list to the left of the pivot (low to pi - 1), where pi is the new index of the pivot element. Recursively call the sort function on the sub-list to the right of the pivot (pi + 1 to high).

Complexity of the Algorithm

Average Time Complexity: O(n log n)

In the average case, Quick Sort performs well, with time complexity dominated by the recursive calls and comparisons needed to partition the sub-lists. This leads to a logarithmic growth rate as the number of elements (n) increases.

Worst-Case Time Complexity: O(n^2)

In the worst case (e.g., the pivot is always the largest or smallest element), Quick Sort can degrade to Bubble Sort-like behavior. This can occur when the chosen pivot consistently partitions the list into unbalanced sub-lists.

2.2 Data Structures

2.2.1 Introduction

[1]Data structures are a specific way of organizing data in a specialized format on a computer so that the information can be organized, processed, stored, and retrieved quickly and effectively. The following is an implementation of eight commonly used data structures.

2.2.2 BST - Binary Search Tree

Overview

A hierarchical structure organized like a tree, where each node contains a key and a value. Keys are meticulously sorted, enabling efficient search operations by continually halving the search space. Insertions and deletions also leverage this sorted order for optimized performance.

Algorithm Description

The binary search tree (BST) defines two main classes: ‘BST‘ and ‘Node‘. The ‘BST‘ class includes methods for inserting, searching, and deleting nodes, along with functions for tree traversal and memory management. The ‘insert‘ method starts at the root and traverses the tree to find the appropriate position for the new node, ensuring the tree maintains its sorted property. If the value to be inserted is less than the current node’s value, it moves to the left child; otherwise, it moves to the right. The ‘search‘ method similarly traverses the tree, returning the node if found, or indicating non-existence if the search path ends. The ‘delete‘ method handles three cases: deleting a leaf node, deleting a node with one child, and deleting a node with two children, where it finds the in-order successor to maintain the BST property. The ‘Node‘ class encapsulates node-specific attributes and methods, including pointers to its children and parent, its value, and boolean flags indicating the presence of children. Additionally, methods for node memory deallocation ensure efficient memory usage. The code also includes a ‘Set‘ class, which manages a dynamic array of integers, providing insertion, deletion, and existence checking functionalities. This comprehensive BST implementation ensures efficient data management and supports fundamental BST operations with memory management protocols.

Usecase

This implementation of a binary search tree (BST) can be effectively utilized in a scenario where an application requires efficient storage and retrieval of sorted data. One specific use case is in an online bookstore inventory management system.

2.2.3 Hashtable

Overview

A data structure that employs a hashing function to efficiently map keys to unique locations within a fixed-size array. This mapping allows for remarkably fast retrieval, insertion, and deletion of key-value pairs on average. However, potential collisions (multiple keys mapping to the same location) necessitate a strategy to handle them.

Algorithm Description

The implementation of hashtable is done using an array of linked lists (chaining) to handle collisions. The Hashtable class initializes with a specified capacity, creating an array where each element is an instance of a List class. The hashfunction method computes the index for storing and retrieving elements based on their keys, ensuring the elements are distributed across the array. The get method searches for an element by traversing the linked list at the calculated index; if the element is not found, it inserts a new node at the front of the list. The set method updates the element's value if it exists; otherwise, it prints an error message. Memory management is handled by the dispose method, which deallocates the memory used by the hashtable and its elements. The print method provides a visual representation of the hashtable's contents by printing each list. The List class supports standard linked list operations, including insertion at both the back and front, retrieval of the head and tail nodes, and disposal of the list's nodes.

Usecase

Using this hashtable implementation, an e-commerce platform can efficiently manage and retrieve user session data, where each session is identified by a unique session ID. The e-commerce platform can this way ensure a smooth and responsive user experience.

2.2.4 Matrix

Overview

A two-dimensional array meticulously arranged in rows and columns, offering a grid-like representation. Elements can be directly accessed using their row and column indices, ensuring swift retrieval. Matrices are commonly used in various applications, including mathematical calculations, image representation, and modeling connections in graphs.

Algorithm Description

The code defines a suite of classes (Matrix, Set, and Vector) to handle fundamental operations on matrices, sets, and vectors. The Matrix class supports matrix creation, element manipulation, matrix arithmetic (addition, subtraction, multiplication), and various utility functions such as calculating the determinant, trace, and transpose of a matrix. The Set class implements a dynamic set data structure with standard operations including insertion, existence check, deletion, and resizing to maintain efficient storage and retrieval. The Vector class allows for vector operations including addition, subtraction, cross product, scalar product, and the calculation of norms. Each class ensures memory management through appropriate allocation and deallocation methods. These classes enable the construction, manipulation, and mathematical operations on complex data structures, adhering to a systematic and structured approach.

Usecase

A scientific computing software needs to handle large datasets for simulations involving linear algebra operations, set manipulations, and vector calculations. Efficient and reliable implementations of these fundamental operations are crucial for the performance and accuracy of the simulations.
The code offers robust implementations of matrices, sets, and vectors that can be used in this scientific computing software. For instance, matrices are often used to represent data transformations and linear equations. The Matrix class allows for building matrices from user input, performing matrix arithmetic, and calculating important properties like determinants and traces, essential for solving systems of linear equations. The Set class can manage unique elements dynamically, beneficial for scenarios where the size of the dataset is not known in advance and efficient checking for element existence is required. The Vector class supports operations needed for vector calculus, including cross and scalar products, which are critical in physics simulations involving forces and motions. By leveraging these classes, the software can efficiently handle and process large datasets, ensuring accurate simulations and analyses.

2.2.5 Pqueue

Overview

An abstract data structure where elements are assigned priorities. These elements are meticulously ordered, guaranteeing that the element with the highest priority is always readily accessible. This structure facilitates efficient insertion of new elements and retrieval of the highest priority elements.

Algorithm Description

The code implements a data structure system in a simplified programming language, which includes three main classes: List, PQueue, and Set. The List class is a doubly linked circular list, where each node holds an integer element and pointers to the

next and previous nodes. It supports basic operations like push_back, push_front, pop_head, and insert at a specific position, along with utility methods for retrieving the head and tail elements, getting the size, and printing the list. The PQueue class implements a priority queue using the List class, ensuring that elements are enqueued in descending order. The enqueue method inserts elements based on their priority, and dequeue removes the head of the list. The Set class implements a dynamic array-based set, handling insertions, deletions, and membership checks, resizing the array as necessary to maintain efficiency. Each class includes methods for disposing of allocated memory to prevent memory leaks.

Usecase

The use case for this data structure system is to manage collections of integer elements efficiently in different contexts. The List class can be used in scenarios where bidirectional traversal is needed, such as in implementing navigable menus or managing a sequence of tasks that require easy insertion and deletion at both ends. The PQueue class is suitable for scenarios where prioritized processing is required, such as task scheduling systems, where tasks with higher priority need to be processed first. The Set class is useful for managing collections of unique elements, providing efficient membership checks, insertions, and deletions, which is beneficial in applications like symbol tables in compilers, managing unique user IDs, or any scenario where duplicate entries need to be avoided. Each class's methodical approach to memory management ensures that resources are efficiently utilized and freed when no longer needed.

2.2.6 Queue

Overview

A linear data structure that adheres to a strict order of insertion. Elements are added at the back (enqueue) and retrieved from the front (dequeue). This ensures that the first element added is also the first element retrieved, maintaining the order of

operations.

Algorithm Description

The code implements a Queue and a Set class using a simplified programming language. The Queue class uses an array to manage elements in a circular buffer style, with fields for capacity, size, front, and top indices. It includes methods to enqueue elements, dequeue elements, peek at the front element, get the size, clear the queue, and dispose of the allocated memory. The enqueue method adds elements to the top of the queue, wrapping around if necessary, while the dequeue method removes elements from the front. The Set class is an array-based implementation that dynamically resizes the array to accommodate elements while preventing duplicates. It includes methods to insert elements, check for the existence of elements, delete elements, clear the set, print the elements, and dispose of the allocated memory. The insert method doubles the array's capacity when it reaches its limit, ensuring efficient insertion, and the exists method checks for membership using a linear search.

Usecase

The Queue class is suitable for scenarios requiring first-in, first-out (FIFO) processing, such as managing tasks in a printer queue or handling customer service requests where the order of arrival determines the order of service. The circular buffer implementation ensures efficient use of space and consistent performance. The Set class is ideal for managing collections of unique elements, such as maintaining a list of unique visitors to a website or managing unique tags in a tagging system. Its dynamic resizing ensures that it can handle varying numbers of elements efficiently, while the membership check prevents duplicate entries. Both classes provide essential data structure functionalities with efficient memory management through their respective dispose methods.

2.2.7 Set

Overview

A collection that meticulously avoids duplicates, ensuring each element is unique. It offers efficient operations for checking membership, adding new elements, and removing existing ones. Sets are frequently implemented using hash tables or specialized tree structures for optimal performance.

Algorithm Description

The code implements a dynamic array-based Set class designed to store unique integers. The Set class features methods to insert elements, check for the existence of elements, delete elements, clear the set, get the size of the set, print the elements, and dispose of the allocated memory. When inserting an element, the Set class dynamically doubles its capacity when necessary to ensure efficient storage and resizing. The exists method performs a linear search to determine if an element is already in the set, ensuring that all elements are unique. The delete method removes an element if it exists, and dynamically adjusts the array size, reducing capacity if needed. The clear method resets the set to its initial state, and the dispose method frees the allocated memory.

Usecase

The Set class is suitable for applications that require managing collections of unique elements, such as maintaining a list of unique IDs, tags, or any other distinct items. Its dynamic resizing ensures efficient use of memory while accommodating varying numbers of elements. This implementation is particularly useful in scenarios where the number of elements can fluctuate significantly, and where duplicates must be avoided. The Set class's methods provide a straightforward interface for adding, checking, deleting, and printing elements, making it a versatile and reliable choice for managing unique collections in a variety of software applications.

2.2.8 Vector

Overview

A resizable array-based data structure that can dynamically adapt its size as needed. It grants efficient access to any element by its index, providing direct retrieval. However, insertions and deletions might require more processing when the size needs to be adjusted to accommodate changes.

Algorithm Description

The code implements two classes: Set and Vector, each encapsulating different functionalities. The Set class manages a dynamic collection of unique integers, with methods to insert, check existence, delete, clear, and print elements, and to get the set's size. It dynamically adjusts its capacity to accommodate new elements and uses linear search to maintain uniqueness. The Vector class represents a mathematical vector with methods to build, get, set elements, compute the cross product, scalar product, addition, subtraction, and calculate the L1 norm. The class ensures vector operations are performed only on vectors of matching dimensions, dynamically resizing and managing internal storage through an array.

Usecase

The Set class is useful in applications requiring efficient management of unique items, such as tracking user IDs, tags, or any unique entities within a system. Its dynamic resizing feature makes it suitable for scenarios with fluctuating data sizes. The Vector class serves in mathematical and scientific computations where vector operations are essential, such as physics simulations, 3D graphics, and machine learning algorithms. It facilitates various vector operations and norms, ensuring accuracy and efficiency in calculations while maintaining the integrity of vector dimensions. Both classes provide robust methods for managing data, making them versatile tools in software development for data management and mathematical computations.

2.2.9 List

Overview

A linear data structure where elements are ordered but not necessarily stored contiguously in memory. It allows for efficient insertion and deletion at specific positions within the order. Random access can be slower compared to vectors as it might involve traversing the list to locate the desired element.

Algorithm Description

The code defines three classes: List, Node, and Set, each with specific data management capabilities. The List class implements a doubly-linked list with a circular structure. It includes methods to add elements to the front or back, insert elements after a specified position, retrieve the head and tail elements, get the size of the list, print the list contents, remove the head element, clear the list, and dispose of the list. The Node class, used by the List class, encapsulates an integer element and pointers to the next and previous nodes, along with methods to get and set these values and dispose of the node. The Set class manages a dynamic array of unique integers, providing methods to insert elements, check existence, delete elements, clear the set, print its contents, get its size, and dispose of it. The Set class dynamically adjusts its capacity as elements are added or removed.

Usecase

The List class is particularly useful in applications requiring dynamic sequential data management, such as task scheduling, playlist management, or undo/redo functionality in software applications. Its ability to insert elements at both ends and manage elements in a circular manner makes it versatile for various use cases. The Node class supports the List class by encapsulating the details of each list element, enabling efficient traversal and modification of the list. The Set class is ideal for applications needing efficient management of unique items, such as maintaining a collection of unique identifiers, tags, or items where duplicates are not allowed. Its

dynamic resizing capability ensures efficient use of memory while accommodating varying data sizes. These classes provide robust and efficient data management solutions for a wide range of software development scenarios.

2.3 String Operations

2.3.1 Introduction

String functions serve as the cornerstone for this manipulation, enabling programmers to perform critical operations on textual information. There are four essential string functions within the StringOperations class: reverse (reorders string characters), strcmp (compares strings lexicographically), toLowerCase (converts to lowercase), and toUpperCase (converts to uppercase).

2.3.2 Function: reverse(String str)

Overview

The reverse function takes a string str as input and reverses the order of its characters.

Functionality

The function iterates through half of the string, swapping characters at opposite indices. It utilizes two pointers, i and j, starting at the beginning and end of the string respectively. These pointers move towards each other, and at each iteration, the characters they point to are swapped.

Time Complexity

The time complexity of the reverse function is O(n), where n is the length of the input string. This is because the function iterates through approximately half of the string (n/2) times, with each iteration involving constant time operations like swapping characters and incrementing/decrementing pointers.

2.3.3 Function: strcmp(String s1, String s2)

Overview

The strcmp function compares two strings s1 and s2 lexicographically.

Functionality

The function iterates through the shorter of the two strings and compares the characters at each index. If a difference is found, the function returns the difference between the ASCII values of the mismatched characters. If no difference is found within the shorter string's length, the function returns the difference in lengths of the two strings.

Time Complexity

The time complexity of the strcmp function is, in the worst case, O(n), where n is the length of the shorter string. This is because the function iterates through the entire shorter string in the worst-case scenario where the strings differ at the beginning. In the average case, however, the complexity might be lower if the strings differ earlier in the comparison.

2.3.4 Function: toLowerCase(String str)

Overview

The toLowerCase function converts all uppercase characters in the input string str to lowercase.

Functionality

The function iterates through each character in the string. If the character is an uppercase letter (between 'A' and 'Z' in ASCII values), it is converted to lowercase by adding 32 to its ASCII value. The modified character is then set back in the string at the corresponding index.

Time Complexity

The time complexity of the toLowerCase function is O(n), where n is the length of the input string. This is because the function iterates through each character in the string, performing constant time comparisons and modifications.

2.3.5 Function: toUpperCase(String str)

Overview

The toUpperCase function converts all lowercase characters in the input string str to uppercase.

Functionality

The function operates similarly to toLowerCase, iterating through each character and converting lowercase letters (between 'a' and 'z' in ASCII values) to uppercase by subtracting 32 from their ASCII values. The modified character is then set back in the string at the corresponding index.

Time Complexity

The time complexity of the toUpperCase function is also O(n), where n is the length of the input string, for the same reasons as the toLowerCase function.

2.4 Math Operations

2.4.1 Introduction

The Math Operations library presents the implementation of four functions designed to serve as a basic mathematical library for the Jack programming language. These functions address fundamental mathematical operations, adhering to a consistent and well-documented structure.

2.4.2 Function: calculateGCD(int n1, int n2)

Overview

The GCD of two integers n1 and n2 is the largest positive integer that divides both n1 and n2 with no remainder.

Functionality

This function calculates the greatest common divisor (GCD) of two integers, n1 and n2. The GCD is the largest positive integer that is a divisor of both n1 and n2. The implementation employs a loop that iterates through potential divisors, checking if they divide both n1 and n2 without remainder. If a common divisor is found, it is stored in the gcd variable. The loop continues until all potential divisors have been checked. Finally, the function returns the calculated GCD.

Time Complexity

The implementation employs a loop that iterates through potential divisors, up to the smaller of n1 and n2. In the worst case, the loop needs to iterate until i reaches the value of the smaller number. Therefore, the time complexity of this function is O(min(n1, n2))

2.4.3 Function: x_to_the_n(int x, int n)

Overview

This function calculates the value of x raised to the power of n.

Functionality

The implementation utilizes a loop that iterates n times. During each iteration, the current value of the result (number) is multiplied by x. This effectively performs the exponentiation operation. After the loop completes, the final value of number represents x raised to the power of n, and it is returned by the function.

Time Complexity

The implementation utilizes a loop that iterates n times. The loop body performs a constant-time multiplication operation. Hence, the time complexity of this function is directly proportional to the exponent n. The time complexity is O(n).

2.4.4 Function: calculate(double x, double y) - hypotenuse calculation

Overview

This function calculates the hypotenuse of a right triangle given the lengths of two sides, x and y.

Functionality

The hypotenuse is the side opposite the right angle in a right triangle. The implementation leverages the Pythagorean theorem, which states that the square of the hypotenuse is equal to the sum of the squares of the other two sides. The function calculates the squares of x and y and then uses the Math.sqrt function from the Jack standard library to compute the square root of their sum. This square root represents the length of the hypotenuse, which is returned by the function.

Time Complexity

The overall time complexity of this function is dominated by the square root operation, resulting in a complexity of O(log n).

2.4.5 Function: calculateCubeRoot(int n)

Overview

This function determines the cube root of an integer n, if it exists.

Functionality

A cube root is a number that, when multiplied by itself three times, equals the original number. The implementation employs a loop that iterates through potential cube roots. During each iteration, the cube of the current counter (i) is compared to n. If a match is found, indicating n is a perfect cube, the function returns the value of i. Otherwise, the loop continues until all potential cube roots have been exhausted. If no match is found, the function returns -1, signifying that n is not a perfect cube.

Time Complexity

The implementation employs a loop that iterates through potential cube roots. In the worst case, the loop needs to iterate until i reaches the cube root of n (or until i^3 is greater than n). Since raising a number to the power of 3 is a constant-time operation within the loop, the time complexity is dominated by the number of iterations. In the worst case, this can be up to the cube root of n. Therefore, the time complexity of this function is O($\sqrt[3]{n}$).

2.5 API

Class Name	Function Name	Description
SORTING ALGORITHMS		
BubbleSort	bubble(A, 5)	Performs bubble sort on array A of size 5.
InsertionSort	insert(A, 5)	Performs insertion sort on array A of size 5.
QuickSort	quick(A, 5)	Performs quick sort on array A of size 5.
SelectionSort	selection(A, 5)	Performs selection sort on array A of size 5.

Table 2.1: API for Sorting Algorithms

Class Name	Function Name	Description
BST		
BST	new(root)	Initializes a new BST with root.
BST	insert(3)	Inserts the element 3 into the BST.
BST	inorder_print(root)	Prints the BST in inorder traversal.
BST	delete(3)	Deletes the element 3 from the BST.
BST	disposeTree(root)	Disposes the BST starting from root.
BST	dispose()	Disposes the BST.

Table 2.2: API for BST

Class Name	Function Name	Description
Hashtable		
Hashtable	get(3, 123)	Retrieves the value associated with key 3 and value 123 from the hashtable.
Hashtable	print()	Prints the hashtable.
Hashtable	dispose()	Disposes the hashtable.

Table 2.3: API for Hashtable

Class Name	Function Name	Description
List		
List	push_back(int)	Appends an integer to the end of the list.
List	push_front(int)	Prepends an integer to the front of the list.
List	print()	Prints the list.
List	insert(int, index)	Inserts an integer at the specified index in the list.
List	clear()	Clears the list.
List	dispose()	Disposes the list.

Table 2.4: API for List

Class Name	Function Name	Description
Matrix		
Matrix	new(m, n)	Creates a new matrix with dimensions m by n.
Matrix	build_matrix()	Builds the matrix.
Matrix	determinant(A)	Calculates the determinant of matrix A.
Matrix	print(A)	Prints matrix A.
Matrix	transpose(A)	Transposes matrix A.
Matrix	dispose()	Disposes the matrix.

Table 2.5: API for Matrix

Class Name	Function Name	Description
PQueue		
PQueue	new()	Creates a new priority queue.
PQueue	enqueue(int)	Enqueues an integer into the priority queue.
PQueue	dequeue()	Dequeues an integer from the priority queue.
PQueue	print()	Prints the priority queue.
PQueue	clear()	Clears the priority queue.
PQueue	dispose()	Disposes the priority queue.

Table 2.6: API for PQueue

Class Name	Function Name	Description
Queue		
Queue	new(size)	Creates a new queue of specified size.
Queue	enqueue(int element)	Enqueues an integer element into the queue.
Queue	dequeue()	Dequeues an integer element from the queue.
Queue	peek()	Peeks at the front element of the queue.
Queue	dispose()	Disposes the queue.

Table 2.7: API for Queue

Class Name	Function Name	Description
Set		
Set	new()	Creates a new set.
Set	insert(int element)	Inserts an integer element into the set.
Set	print()	Prints the set.
Set	delete(int element)	Deletes an integer element from the set.
Set	dispose()	Disposes the set.

Table 2.8: API for Set

Class Name	Function Name	Description
Vector		
Vector	new(dimension, true)	Creates a new vector with specified dimension.
Vector	scalarProduct(B)	Calculates the scalar product with vector B.
Vector	crossProduct(B)	Calculates the cross product with vector B.
Vector	norm()	Calculates the norm of the vector.
Vector	dispose()	Disposes the vector.

Table 2.9: API for Vector

Class Name	Function Name	Description
STRING MANIPULATIONS		
StringOperations	reverse(str)	Reverses the string str.
StringOperations	strcmp(str1, str2)	Compares strings str1 and str2.
StringOperations	toLowerCase(str)	Converts the string str to lowercase.
StringOperations	toUpperCase(str)	Converts the string str to uppercase.

Table 2.10: API for String Manipulations

Class Name	Function Name	Description
MATH OPERATIONS		
CubeRoot	calculateCubeRoot(num)	Calculates the cube root of number.
GCD	calculateGCD(n1, n2)	Calculates the greatest common divisor of n1 and n2.
Hypotenuse	calculate(side1, side2)	Calculates the hypotenuse given side1 and side2.
PowerCalculator	x_to_the_n(x, n)	Calculates x raised to the power of n.

Table 2.11: API for Math Operations

3 Results

3.1 Sorting Algorithms

The VM TRANSLATOR output of the sorting algorithms implemented.

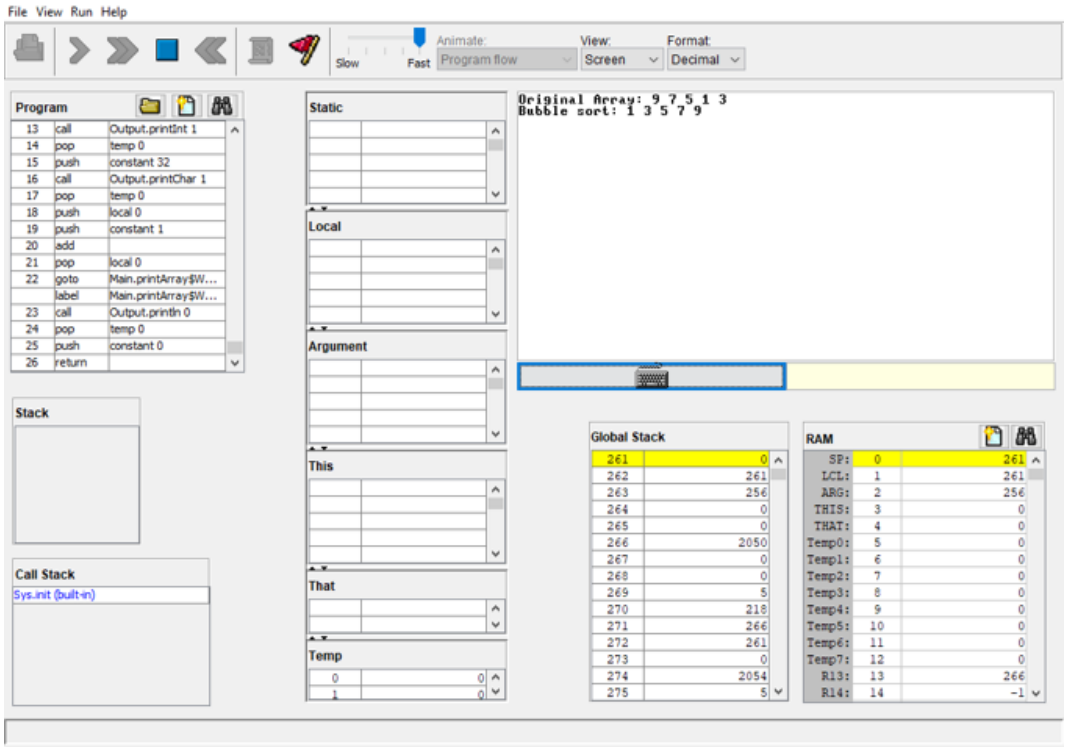


Figure 3.1: Working of Bubble Sort Algorithm

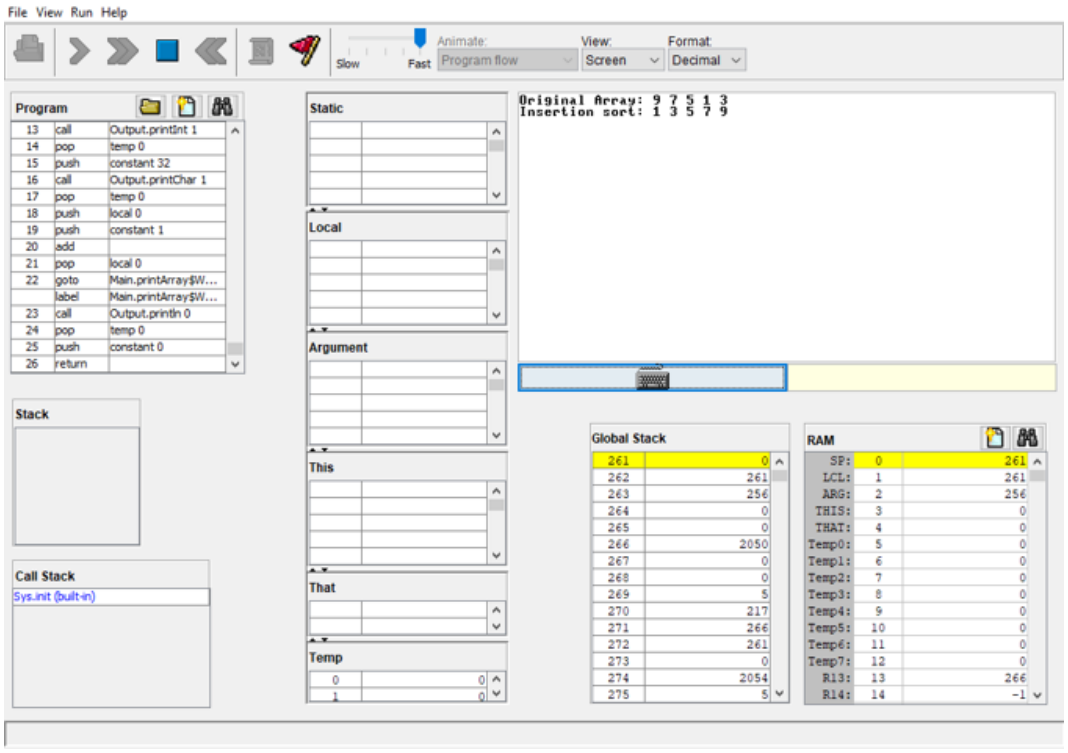


Figure 3.2: Working of Insertion Sort Algorithm

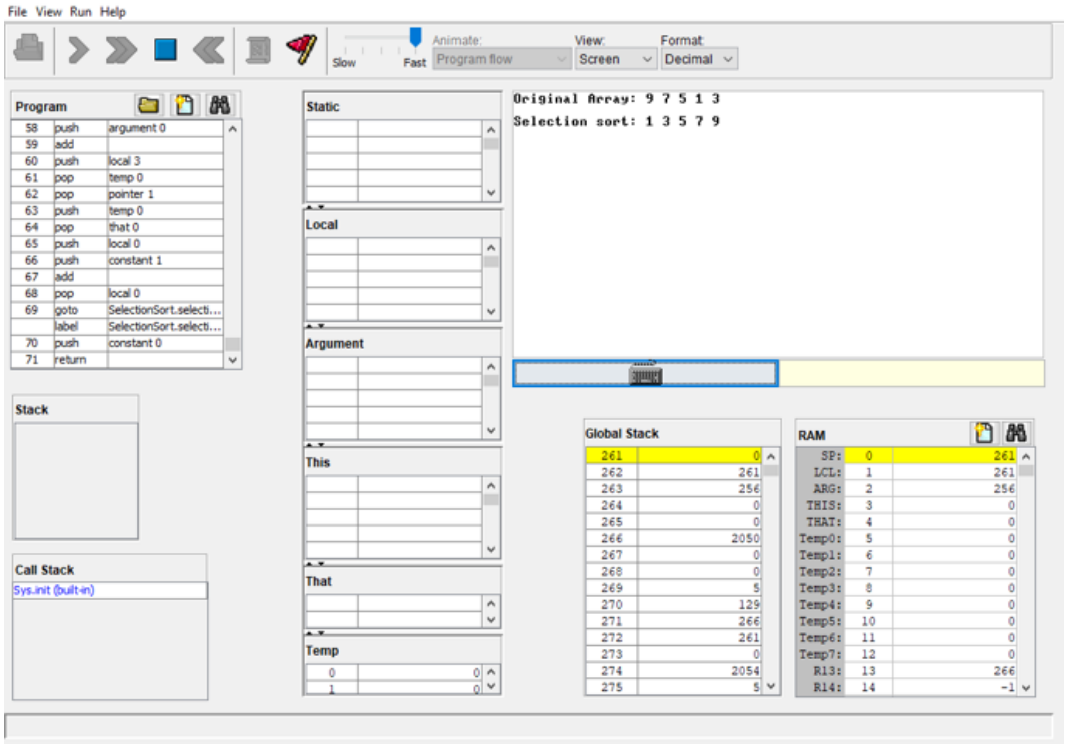


Figure 3.3: Working of Selection Sort Algorithm

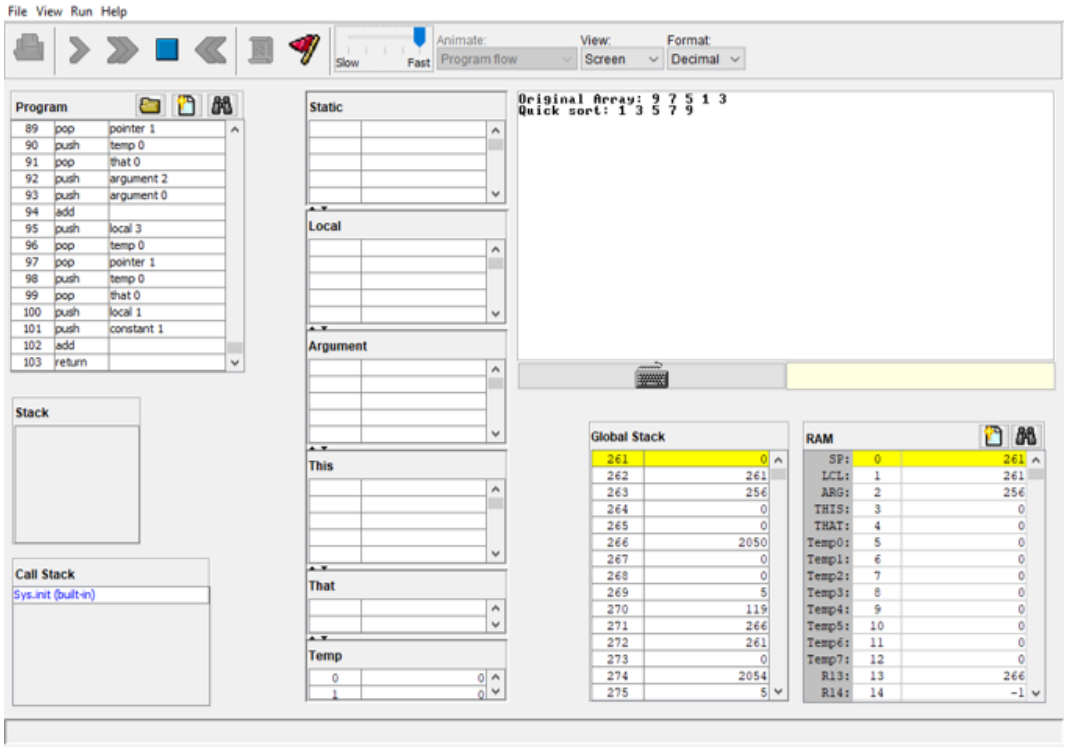


Figure 3.4: Working of Quick Sort Algorithm

3.2 Data Structures

The VM TRANSLATOR output of the data structures implemented.

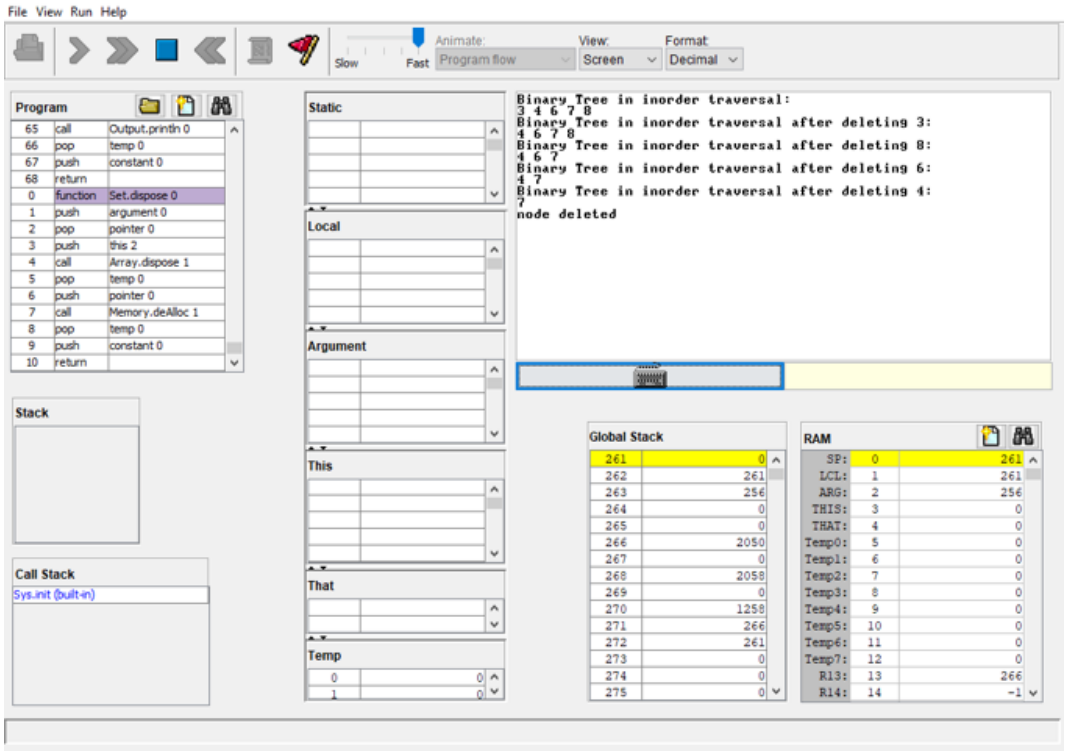


Figure 3.5: binary search tree

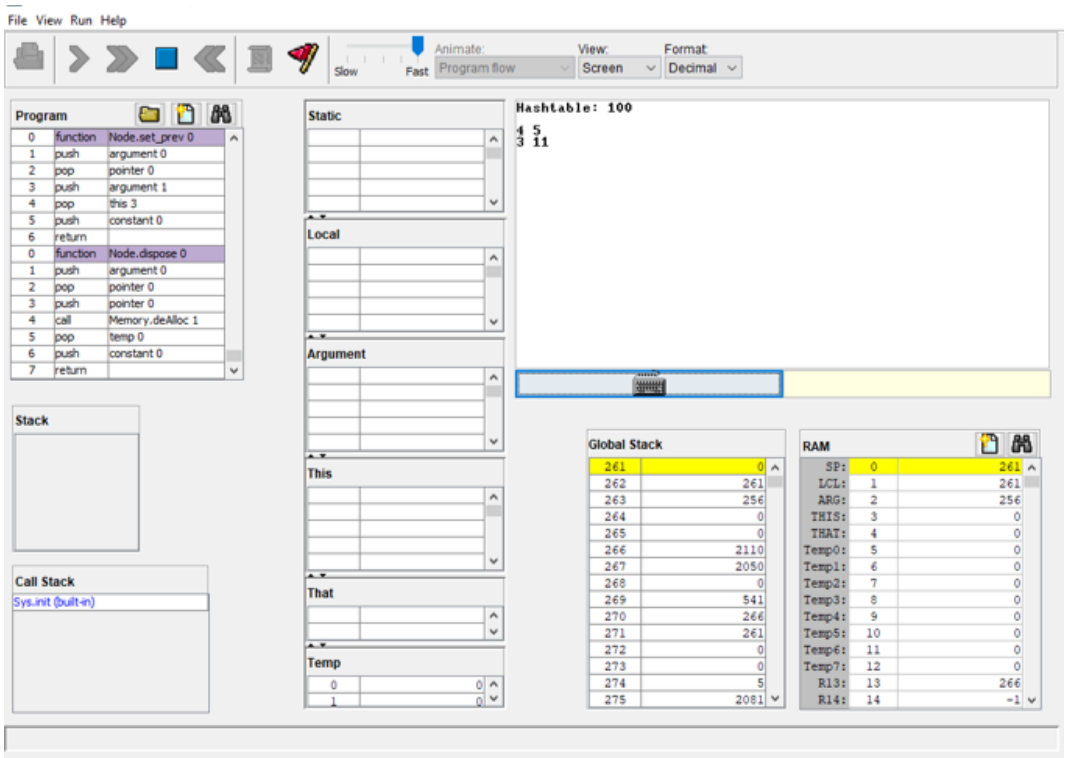


Figure 3.6: hashtable

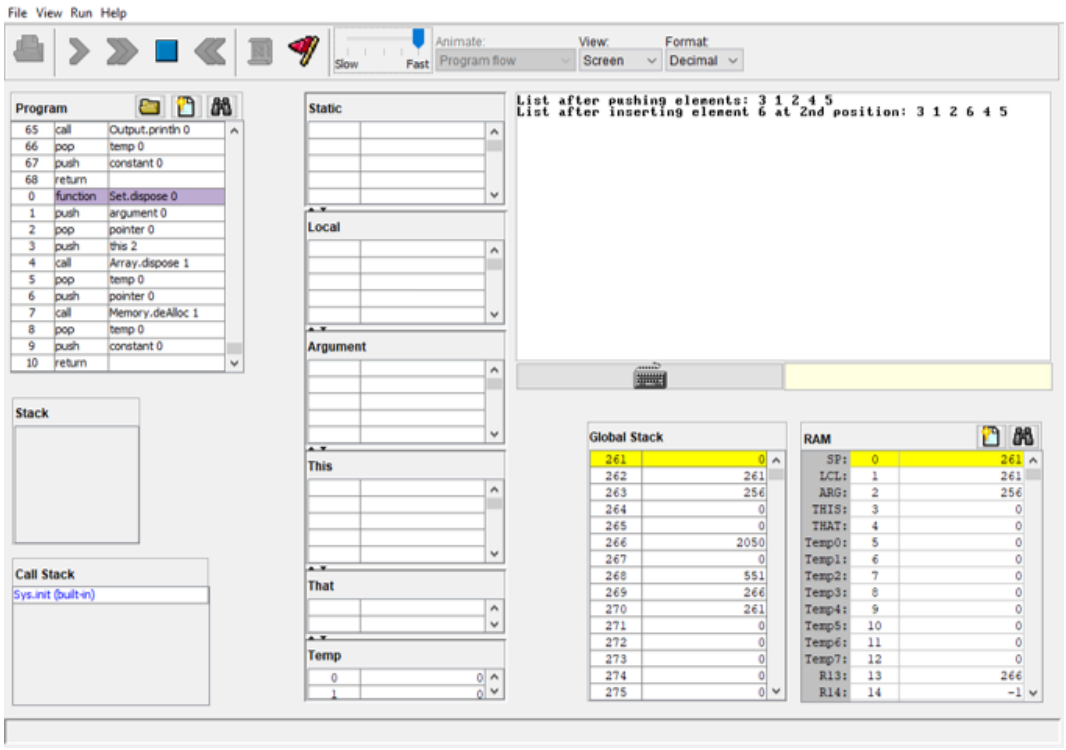


Figure 3.7: list

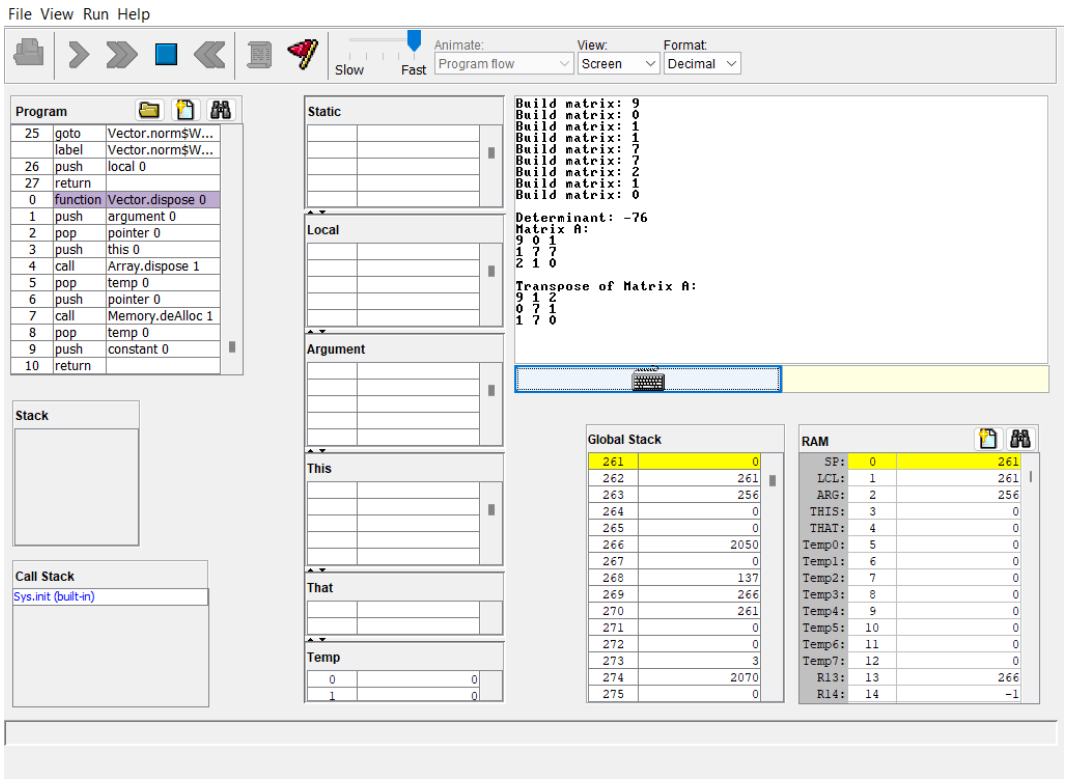


Figure 3.8: matrix

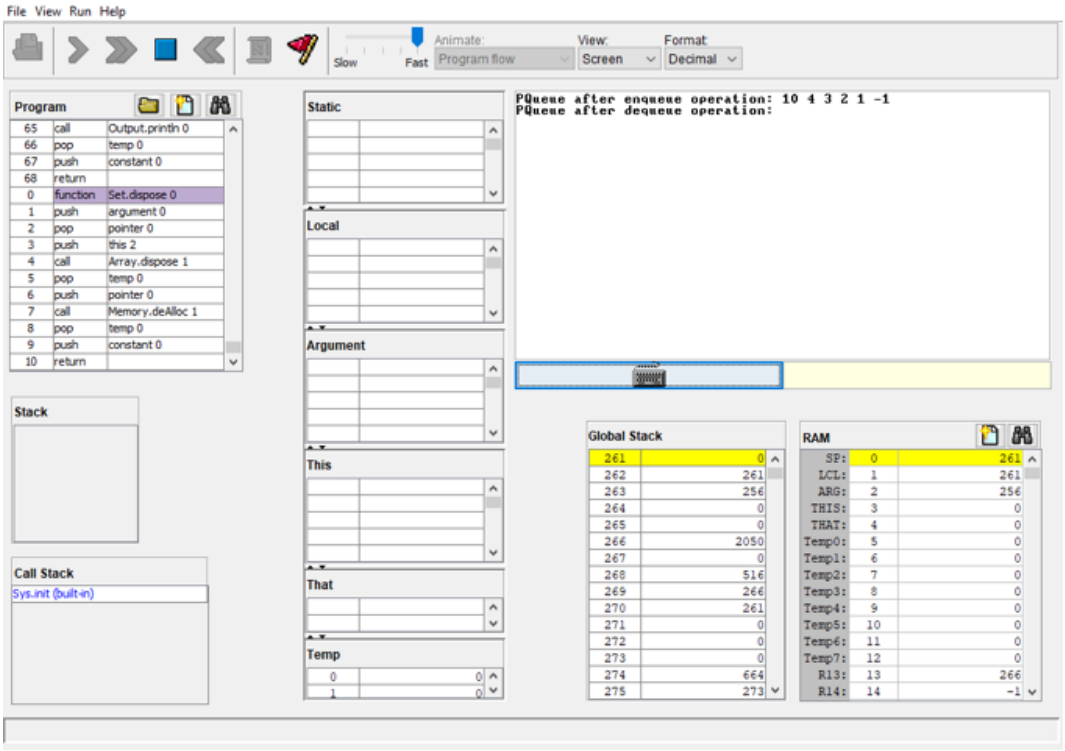


Figure 3.9: pqueue

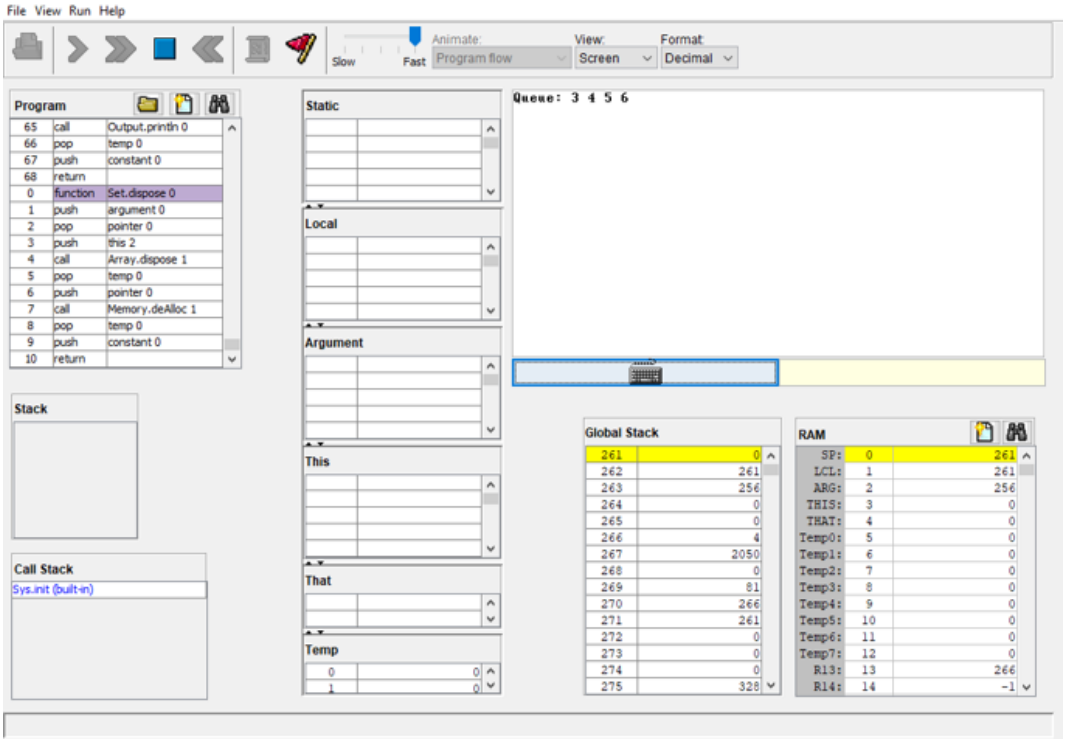


Figure 3.10: queue

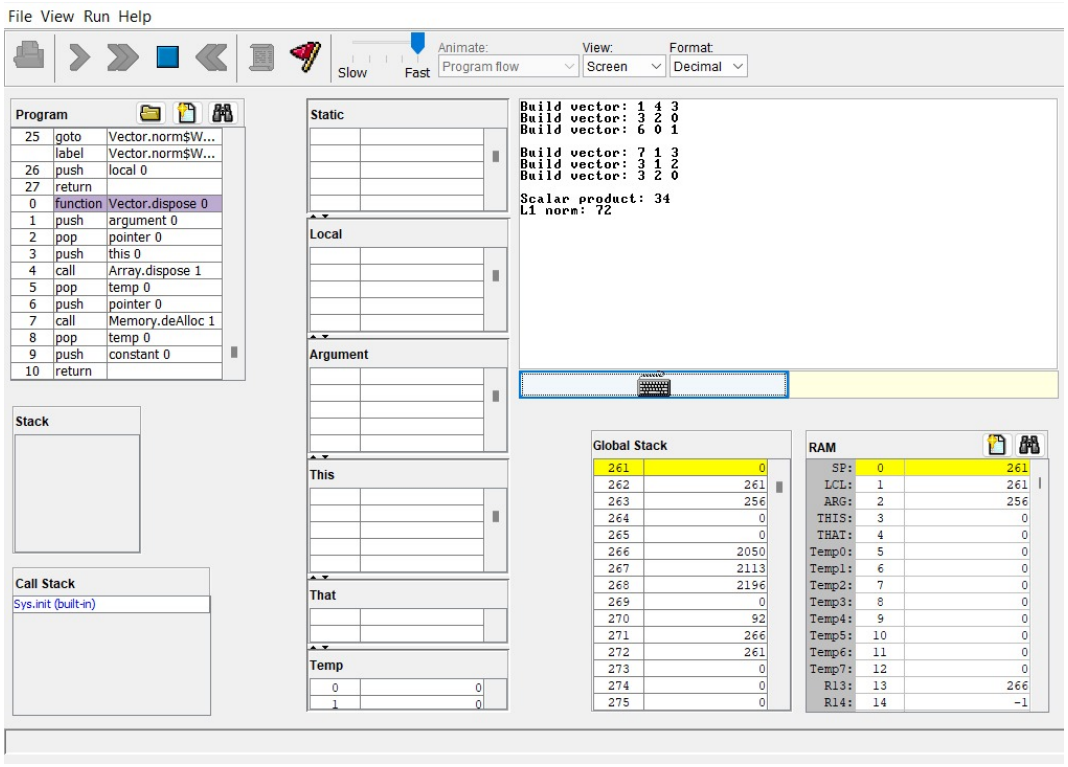


Figure 3.11: vector

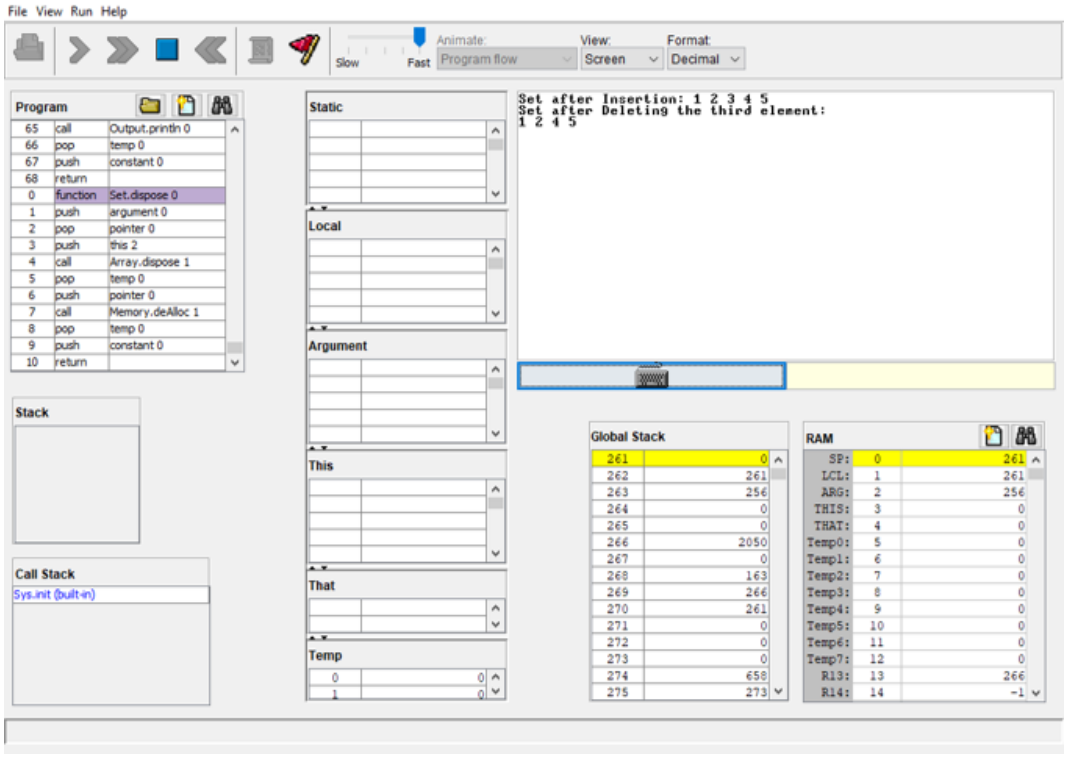


Figure 3.12: set

3.3 Math Operations

The VM TRANSLATOR output of the math operations implemented.

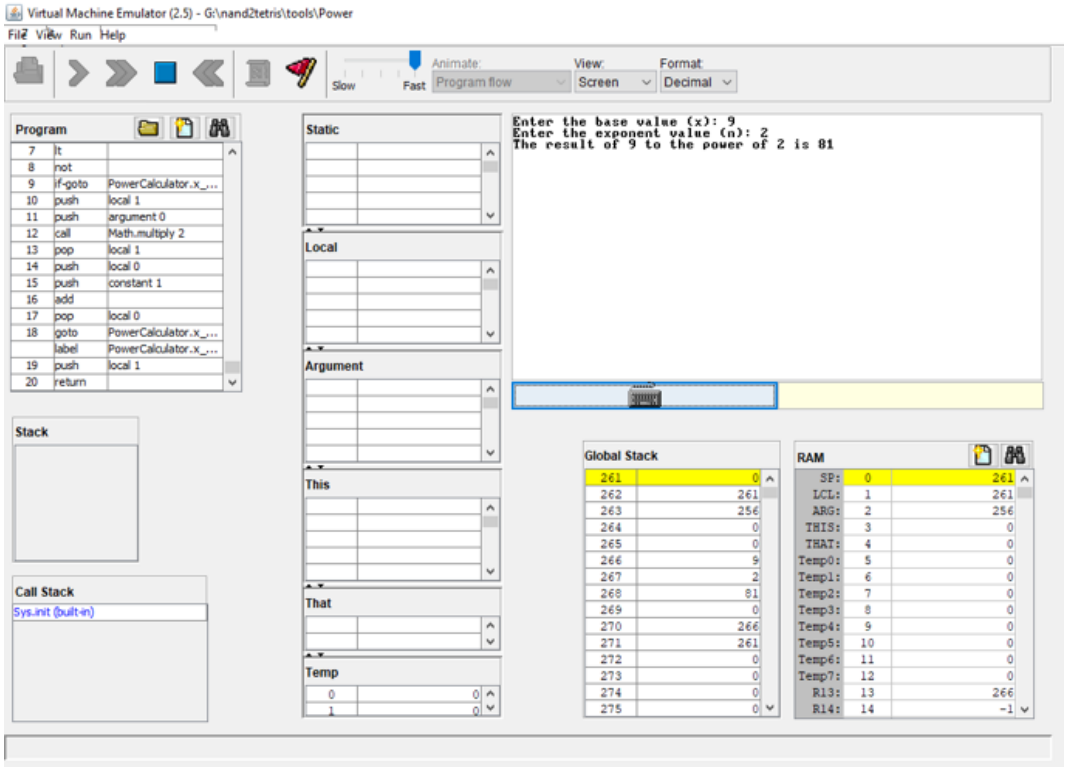


Figure 3.13: Computing the exponent

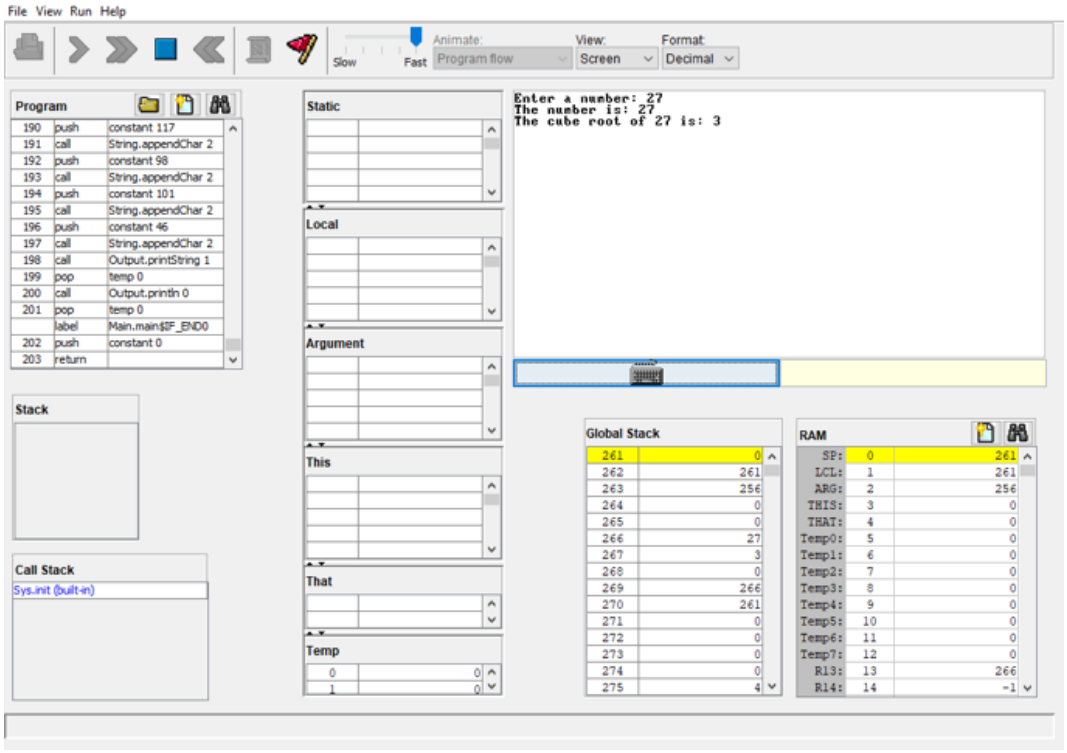


Figure 3.14: Computing the cube root of a number

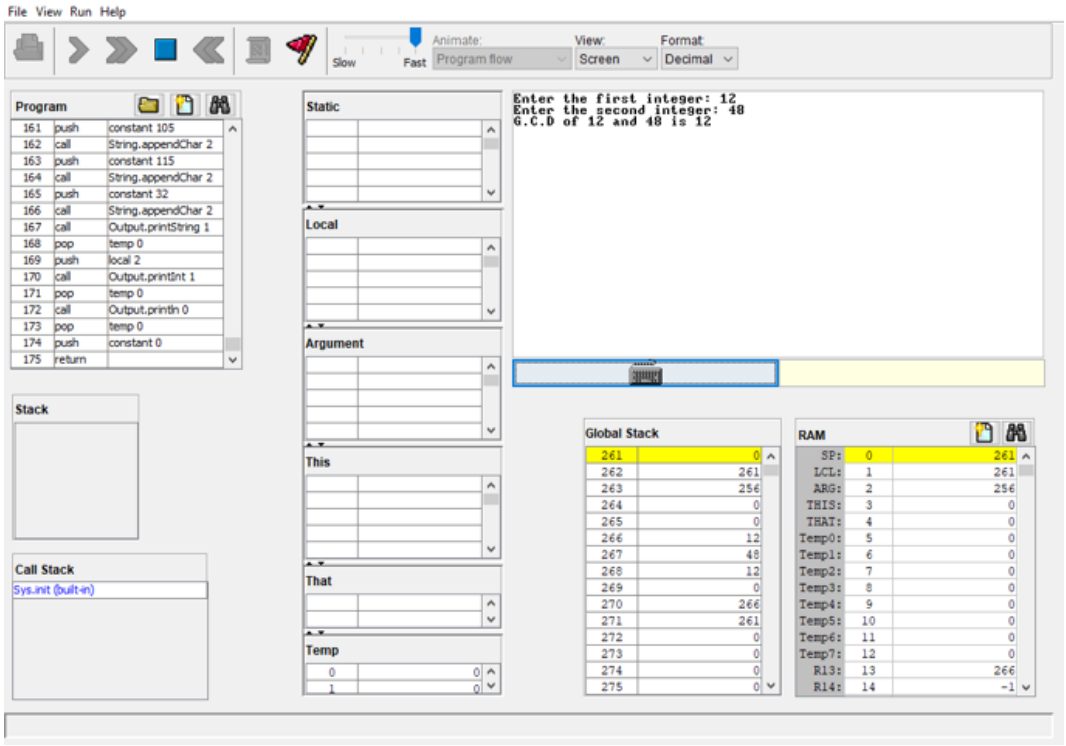


Figure 3.15: Computing the GCD of two numbers

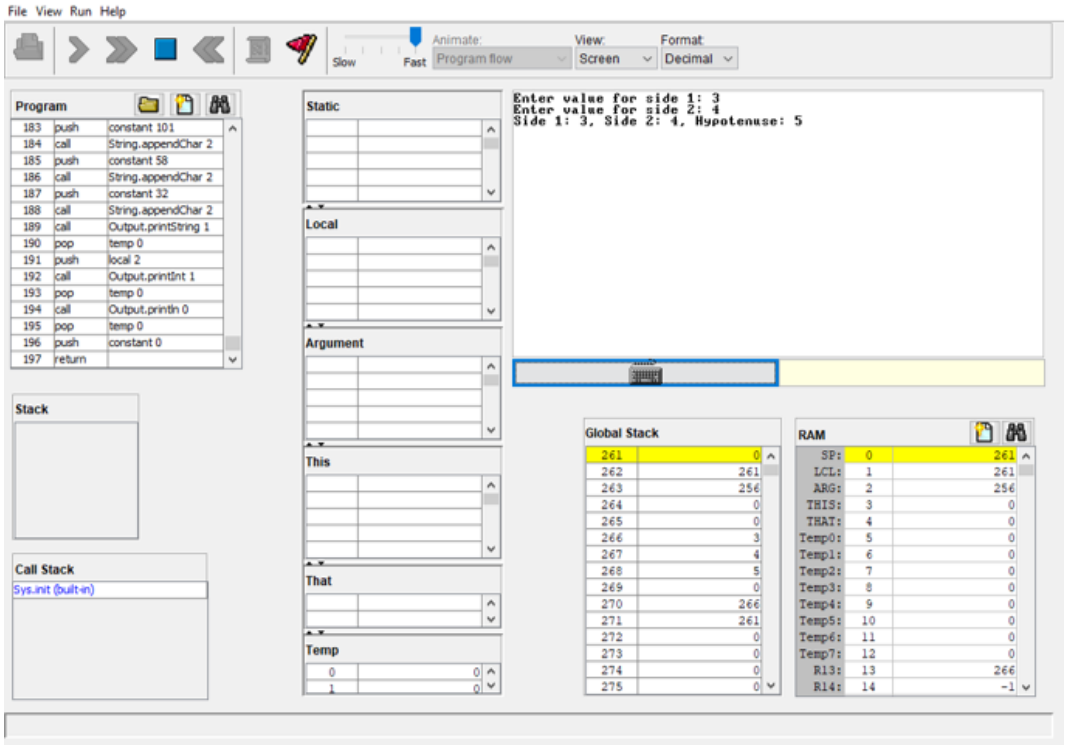


Figure 3.16: Computing the hypotenuse of a triangle

3.4 String Manipulations

The VM TRANSLATOR output of the string manipulations implemented.

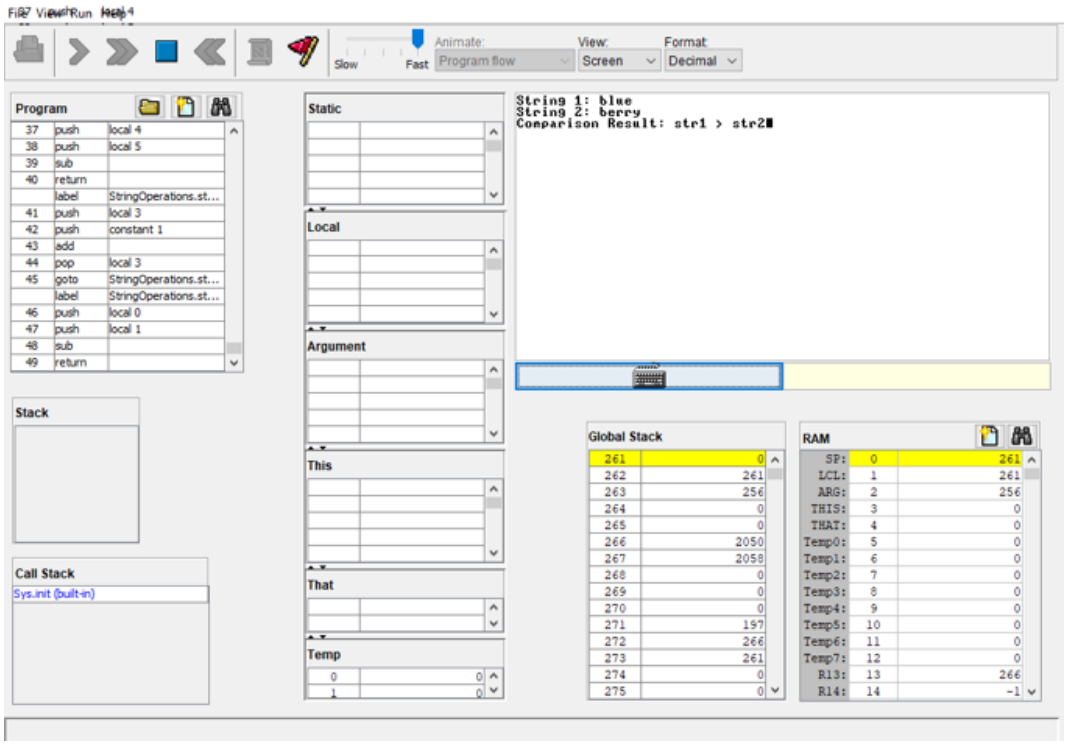


Figure 3.17: Comparison of two strings lexicographically

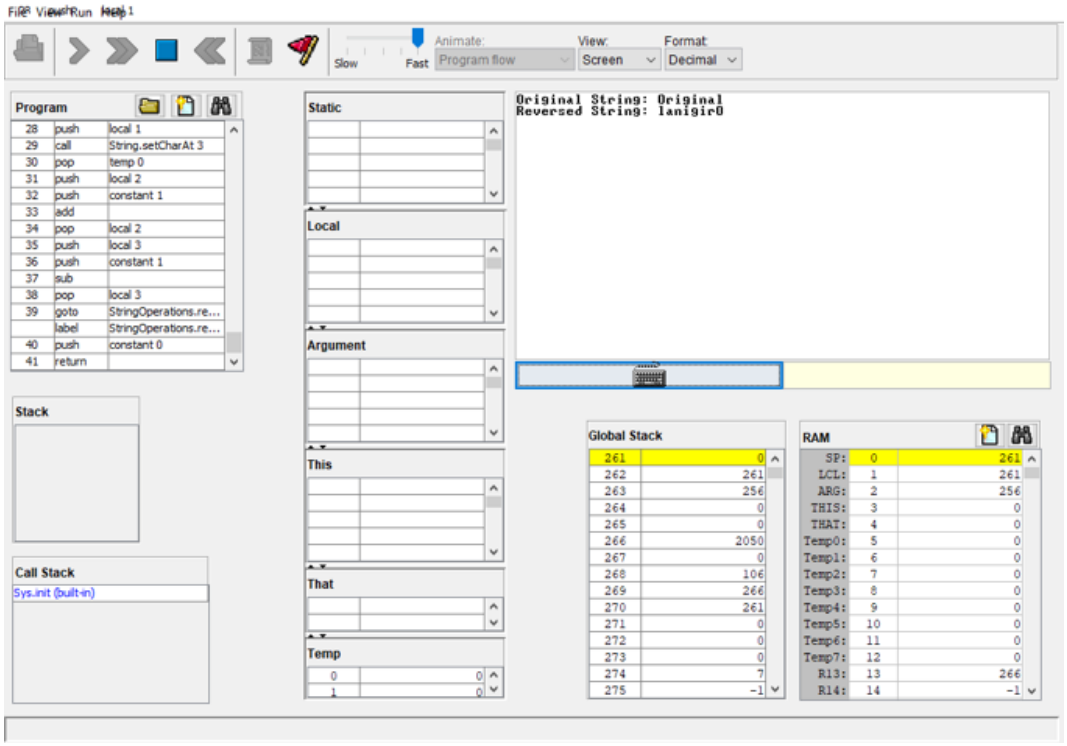


Figure 3.18: Computing the Reverse of a string

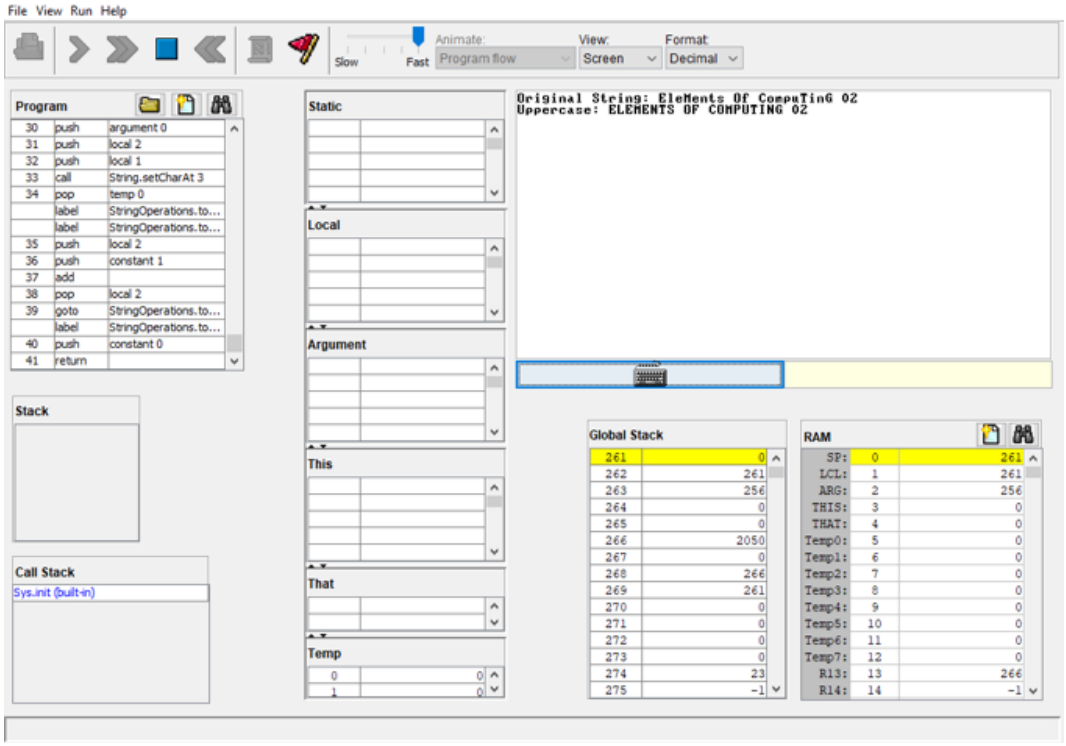


Figure 3.19: Converting a string to uppercase

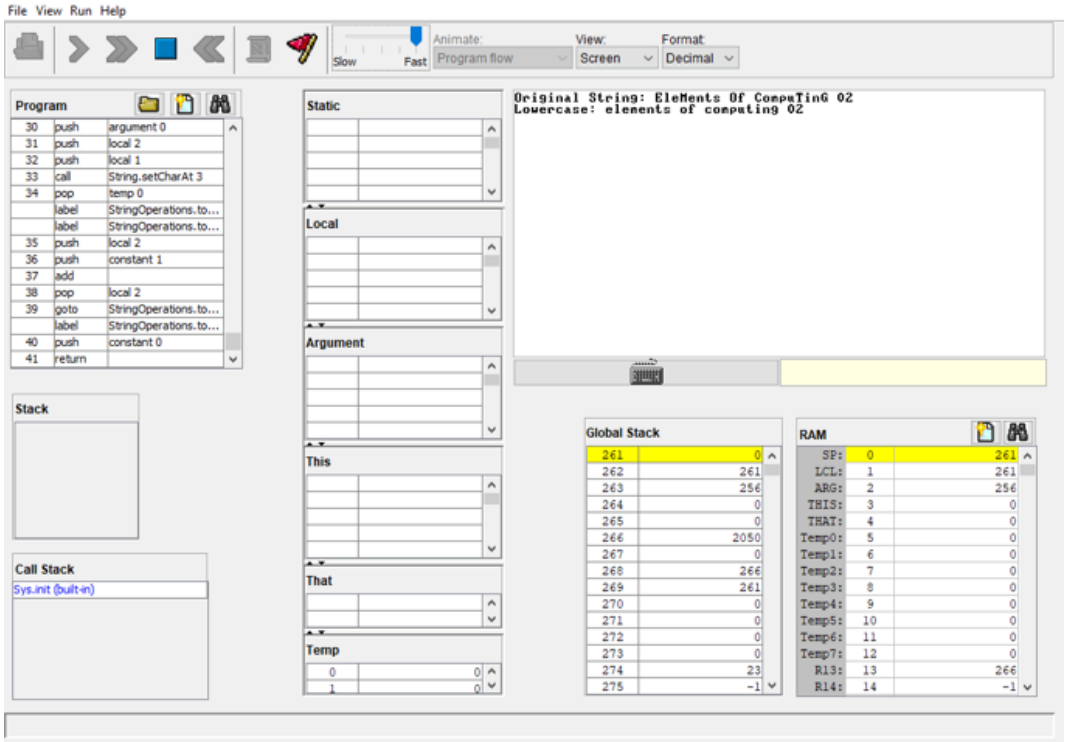


Figure 3.20: Converting a string to lowercase

4 Conclusion

In conclusion, this project has successfully addressed the lack of a built-in standard library in the Jack programming language by implementing essential data structures and utility functions. The creation of this custom standard library enhances the capabilities of Jack, empowering developers to work more efficiently. These functionalities provide a foundation for further development and exploration within the Jack programming environment.

Bibliography/References

[1] https://github.com/BornaGajic/jack-stl