MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

DSL for Data Structures Manipulation

Project report

Mentor: assoc.prof., Cojuhari Irina
Students: Chichioi Iuliana, FAF-221
Calugareanu Ana, FAF-221
Reabciuc Brianna, FAF-221

Chișinău, 2024

Abstract

The project titled "DSL for Linear Data Structures Manipulation" represents a collaborative effort by students Calugareanu Ana, Chichioi Iuliana, and Reabciuc Brianna from the Technical University of Moldova. This project delves into domain analysis, language design, implementation details, DSL evaluation, conclusions, and bibliography.

Keywords: Domain-Specific Language, Linear Data Structures, Data Manipulation, Software Engineering, Numerical Data Operations.

In the realm of software development, DSL have garnered considerable attention for their ability to provide tailored solutions to specific problem domain. This project focuses on developing a DSL for manipulation tasks related to linear data structures, including arrays, stacks, queues, and linked lists.

The domain under study encompasses fundamental tasks such as creating, modifying, and analyzing linear data structures, which are essential in various fields, including software engineering, data science, and computational research. Traditional programming languages often lack specialized constructs for efficient manipulation of linear data structures, necessitating the need for a DSL in this domain.

The project's objectives include designing and implementing a DSL for manipulation of linear data structures, developing intuitive syntax and specialized operations tailored to these structures, and evaluating the effectiveness and efficiency of the DSL compared to existing solutions. Research methodologies such as literature review, language design principles, and empirical evaluation will be employed to achieve these objectives.

Through this project, the aim is to contribute to the advancement of DSL engineering and provide a valuable tool for simplifying and enhancing manipulation tasks related to linear data structures across various domains. The subsequent chapters will delve into domain analysis, language design, implementation details, and DSL evaluation, providing a comprehensive exploration of the project's objectives and outcomes.

Content

Introduction

Data structures play a fundamental role in organizing and managing data efficiently in software development. Among these, linear data structures hold particular significance due to their sequential arrangement of elements. In this project, the focus lies on the manipulation of linear data structures, including both static and dynamic variants.

Static data structures, exemplified by arrays, offer a fixed-size collection of elements with contiguous memory allocation. These structures provide fast access to elements based on their indices, making them suitable for scenarios where the size of the data set is known in advance.

On the other hand, dynamic data structures such as queues, stacks, and linked lists provide flexibility in managing data by allowing elements to be added or removed dynamically during program execution. Queues adhere to the First-In-First-Out (FIFO) principle, stacks follow the Last-In-First-Out (LIFO) principle, while linked lists offer efficient insertion and deletion operations through their node-based structure.

To facilitate efficient manipulation of numerical data within these data structures, a DSL will be developed. This DSL aims to provide intuitive syntax and specialized operations tailored to numerical data manipulation, enhancing productivity and code readability for developers.

The project will involve comprehensive domain analysis to understand the requirements and operations needed for numerical data manipulation within linear data structures. Following this, a grammar will be created to define the syntax of the DSL, ensuring clarity and conciseness in expressing data manipulation tasks.

Implementation of the DSL will be carried out in Python, leveraging the flexibility and ease of use of the language. Additionally, ANTLR will be employed to generate a parser for the DSL grammar, enabling seamless interpretation and execution of DSL commands [1].

Through this project, the aim is to provide developers with a tool for efficiently manipulating numerical data within linear data structures, thereby enhancing code quality and facilitating software development in various domains.

Abbreviations

- [DSL] Domain-Specific Language
- [ANTLR] ANother Tool for Language Recognition

1 Domain Analysis of Data Structures Manipulation

Domain analysis is a crucial step in software engineering that involves understanding the problem domain, identifying relevant concepts, and defining the scope of the system to be developed. In the context of this project, the focus will be on developing a DSL for manipulating linear data structures, particularly numerical ones.

1.1 What are Data Structures?

Data structures are fundamental components in computer science that enable efficient organization, storage, and manipulation of data. They provide a way to represent complex data in a structured manner, allowing for easy access and modification, fig. 1.1.1.

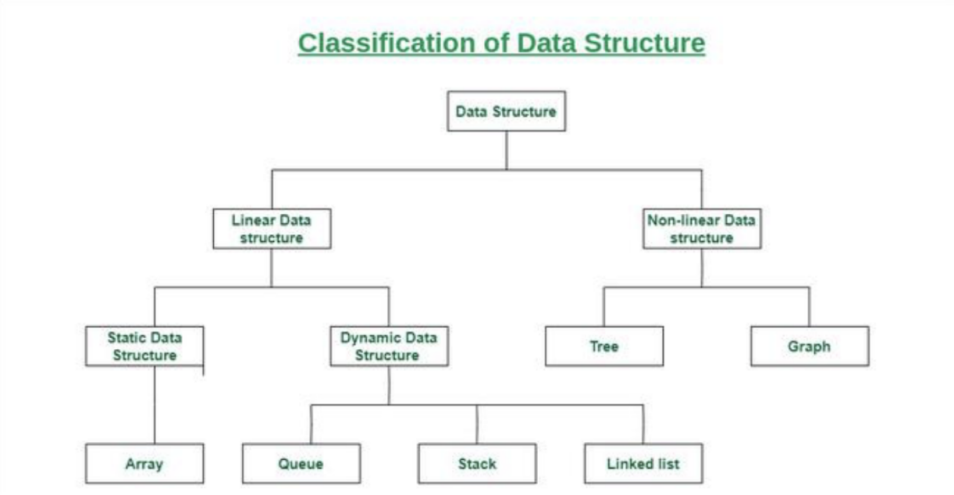

Figure 1.1.1 - Classification of Data Structures[2]

In this DSL project, the emphasis will be on linear data structures, which arrange elements sequentially, such as arrays, linked lists, stacks, and queues.

1.2 Importance of Data Structures

The importance of data structures lies in their ability to optimize various operations on data, such as searching, sorting, and retrieving. By choosing the right data structure for a given problem, developers can significantly improve the performance and efficiency of their software.In this project, the goal is to harness the power of data structures to provide users with a simple yet powerful tool for manipulating linear data.

1.3 Overview of the Domain

The domain encompasses the manipulation of linear data structures, with a focus on numerical ones. Linear data structures are fundamental components in computer science that arrange elements sequentially, enabling efficient organization, storage, and manipulation of data. This includes operations such as inserting, deleting, searching, and updating elements in arrays, linked lists, stacks, queues, and other linear data

structures.

In the context of this project, the aim is to develop a DSL tailored to this domain. The DSL will provide users with an intuitive and efficient way to work with numerical data in their software projects. By leveraging the power of data structures, the DSL will enable developers to perform complex operations on numerical data with ease, optimizing performance and efficiency.

The development of this DSL involves analyzing the requirements of the domain, identifying common tasks and operations performed on linear data structures, and designing a language that simplifies and streamlines these tasks. The DSL will offer a range of features and functionalities to support various use cases, including algorithms for sorting, searching, and manipulating numerical data [3].

Overall, the goal of this project is to empower developers to work more effectively with linear data structures, particularly in the context of numerical data manipulation.

1.3.1 Problem Statement

In software engineering, efficient manipulation of data structures is crucial for the development of high-performance applications. However, working with linear data structures, especially numerical ones, often involves writing repetitive and error-prone code. This becomes increasingly challenging as the complexity of the data and operations grows.

To address this problem, there is a need for a DSL that simplifies the manipulation of linear data structures while providing the flexibility and efficiency required for various software projects. The DSL should offer intuitive syntax and built-in functions tailored to the specific requirements of handling numerical data within linear structures.

The aim of this project is to develop such a DSL, focusing on operations like inserting, deleting, searching, and updating elements in arrays, linked lists, and other linear data structures commonly used in software development. By providing a specialized language for this domain, developers can streamline their coding process, reduce errors, and improve the performance of their applications.

1.3.2 General Description

The general description section provides an overview of the problem domain and the objectives of the project. It outlines the scope of the project and the intended outcomes.

The project focuses on developing a DSL tailored to the manipulation of linear data structures, with a particular emphasis on numerical ones. Linear data structures such as arrays, linked lists, stacks, and queues are fundamental components of software systems, widely used for organizing and processing data in various applications. However, efficiently manipulating these data structures, especially when dealing with numerical data, can pose significant challenges for developers.

1.3.3 Background of the problem and its impact

Understanding the background of the problem and its impact is crucial for identifying the need for a solution. This section explores the challenges associated with manual manipulation of linear data structures, especially numerical ones, and discusses the consequences of inefficient data handling in software development.

In traditional software development, developers often need to implement algorithms and data manipulation routines for working with linear data structures from scratch. This manual approach not only requires significant time and effort but also increases the risk of introducing bugs and errors into the codebase. Furthermore, optimizing these routines for performance and scalability can be complex and time-consuming, leading to suboptimal outcomes.

The impact of inefficient data handling in software development can be profound. Poorly designed or implemented algorithms for manipulating linear data structures can result in slower execution times, increased memory usage, and reduced overall performance of software applications. In mission-critical systems or applications where performance is paramount, such inefficiencies can have serious consequences, including system failures, degraded user experience, and financial losses.

By developing a DSL tailored to the manipulation of linear data structures, we aim to mitigate these challenges and their associated impacts. By providing developers with a high-level, expressive language for working with linear data structures, we can improve the efficiency, reliability, and performance of software applications across a wide range of domains and use cases.

1.3.4 Fields of Use

The fields of use section outlines the various domains and applications where the proposed DSL can be applied effectively. It explores the diverse range of scenarios and industries where efficient manipulation of linear data structures, especially numerical ones, is essential for achieving optimal outcomes.

The DSL developed in this project can find applications in a wide range of fields, including scientific computing and numerical simulations, financial modeling and analysis, engineering and computational modeling, data analytics and machine learning, game development and computer graphics, embedded systems and real-time processing, cryptography and security algorithms, bioinformatics and genomic analysis, Internet of Things (IoT) devices and sensor networks, and education and academic research [4].

By providing a versatile and customizable tool for manipulating linear data structures, the DSL can empower developers in these fields to streamline their workflows, improve code readability, and enhance the performance of their software applications.

1.3.5 Challenges that can be encountered

Despite its potential benefits, the development and adoption of a DSL for manipulating linear data structures can pose several challenges. This section explores some of the key obstacles that developers may encounter during the design, implementation, and deployment phases of the project.

Some of the challenges that can be encountered include design complexity, language semantics, tooling and integration, and performance optimization.

Design complexity involves striking the right balance between expressiveness, simplicity, and flexibility. Language semantics must be defined carefully to avoid ambiguities or inconsistencies. Tooling support and integration into existing workflows are essential for successful adoption. Performance optimization requires attention to algorithm design, data structures, and implementation details. By anticipating and addressing these challenges proactively, developers can increase the likelihood of success and adoption for the DSL project.

1.4 Examples of data structures manipulation

This section explores examples of manipulating data structures, fundamental to computer science and software development. It covers arrays, which store items of the same data type in contiguous memory, and linked lists, comprising nodes with data and references to subsequent nodes. Additionally, it touches upon stack and queue structures, crucial for various computational tasks. Understanding these concepts is essential for effective problem-solving and software engineering.

Array

An array is a collection of items of the same data type stored at contiguous memory locations. It is characterized by having homogeneous elements, meaning all elements within an array must be of the same data type. In most programming languages, elements in an array are stored in contiguous (adjacent) memory locations, fig. 1.4.1. [5]

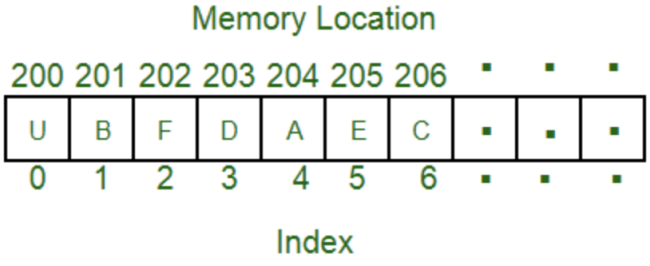

Figure 1.4.1 - Array [2]

Arrays typically use zero-based indexing, where the first element is accessed with an index of 0, the second with an index of 1, and so on. This allows for constant-time (O(1)) access to elements based on their index. Arrays can be one-dimensional, two-dimensional, or multi-dimensional, depending on the number

of dimensions required. Common operations on arrays include accessing elements, insertion, deletion, and searching.

Linked List

A Linked List is a linear data structure consisting of a chain of nodes, each containing data and a reference to the next node. Unlike arrays, linked list elements are not stored at contiguous memory locations. Each element in a linked list, or node, contains two components: the actual data or value associated with the element, and a reference or pointer to the next node in the linked list, fig. 1.4.2.[6]

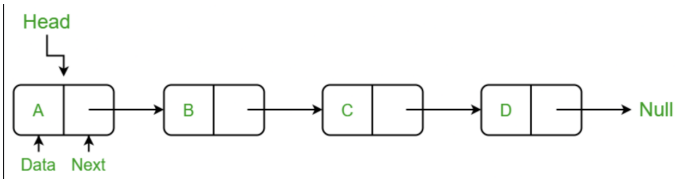

Figure 1.4.2 - Linked List [2]

Linked lists can be singly linked, doubly linked, or circular linked, depending on the type of references they contain. Operations on linked lists include accessing elements, searching, insertion, and deletion.

Stack Data Structure

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It can be of two types: fixed-size stack and dynamic-size stack. In a fixed-size stack, the size is predetermined and cannot be changed during runtime, while a dynamic-size stack can grow or shrink dynamically as elements are added or removed,fig. 1.4.3.

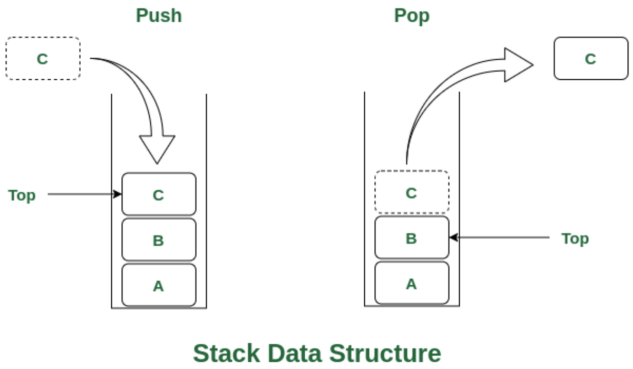

Figure 1.4.3 - Stack [2]

Stack operations include push() for inserting elements, pop() for removing elements, top() for accessing the topmost element without removing it, size() for getting the size of the stack, and isEmpty() for checking if the stack is empty. [7]

Queue Data Structure

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. It can be of various types, including input restricted queue, output restricted queue, circular queue, double-ended queue (Dequeue), and priority queue, fig. 1.4.4.

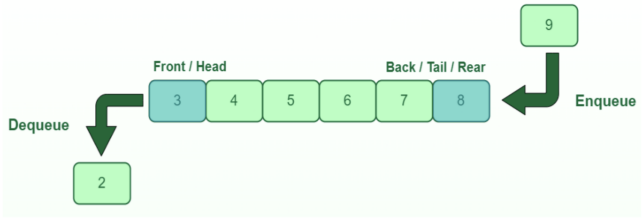

Figure 1.4.4 - Queue [2]

Operations on queues include Enqueue() for adding elements to the queue, Dequeue() for removing elements from the queue, Peek() or front() for accessing the front element without removing it, rear() for accessing the rear element without removing it, isFull() for checking if the queue is full, and isNull() for checking if the queue is empty.[8]

1.5 Conceptualizing a Solution

To embark on the creation of a DSL for data structure manipulation in Python, a thorough examination of the domain of numerical data structures is imperative. Primarily focusing on static and dynamic data structures is essential.

Static Data Structures: Array

Static data structures, exemplified by arrays, serve as optimal vessels for storing fixed-size collections of numerical elements. Arrays boast efficient random access to elements based on their indices, rendering them ideal for scenarios necessitating constant-time access. The DSL under consideration will encompass functionalities for array creation, access, update, and deletion.

Dynamic Data Structures: Queue, Stack, Linked List

Dynamic data structures, including queues, stacks, and linked lists, prove indispensable in scenarios where data collection sizes fluctuate dynamically. These structures offer flexibility in memory allocation and management, catering to applications with varying data input/output requirements. The DSL blueprint will encompass operations for element enqueueing and dequeueing in queues, element pushing and popping in stacks, and node insertion and deletion in linked lists.

The implementation of the DSL will harness ANTLR, a potent parser generator facilitating the definition of DSL grammar and automatic generation of Python code for parsing and interpreting DSL expressions.

ANTLR enables syntax rule definition, semantic action specification, and parser generation, ensuring efficient processing of DSL statements.

By conceptualizing the solution in this manner, the aspiration is to forge a robust DSL for data structure manipulation in Python, specifically tailored to numerical data scenarios. This DSL promises an intuitive interface for developers grappling with linear data structures, fostering heightened productivity and facilitating efficient data manipulation in numerical applications.

1.5.1 Identification and Classification of the Target Group

In order to develop a DSL for data structure manipulation in Python, it's essential to identify and classify the target group accurately. This step involves understanding the demographics, preferences, and specific needs of the users who will benefit from the DSL.

Presentation of the Target Group

The primary target group for the DSL encompasses a diverse range of professionals, predominantly developers and data scientists operating within specialized domains. These domains span critical fields such as scientific computing, data analysis, machine learning, and computational finance. Within these domains, practitioners grapple with intricate numerical computations, algorithmic complexities, and data manipulation tasks of varying scales.

For developers and data scientists in these domains, the efficient manipulation of data structures is paramount. Whether they're working on optimizing algorithms for large-scale data processing, implementing advanced machine learning models, or conducting complex financial analysis, the ability to leverage powerful data structures is indispensable. These professionals seek tools and frameworks that not only facilitate the manipulation of data structures but also streamline their workflow and enhance productivity.

To effectively address the needs of this target group, it is imperative for the project team to conduct a comprehensive analysis of their requirements and preferences. This entails delving into the intricacies of their workflows, understanding the specific challenges they face, and identifying opportunities for optimization. By gaining insights into the unique demands of developers and data scientists in different domains, the project team can tailor the DSL to meet their diverse needs effectively.

Key considerations in this process include designing an intuitive syntax that aligns with the programming paradigms prevalent in these domains. Additionally, the DSL should offer robust functionality, providing a comprehensive suite of operations and algorithms commonly used in data manipulation tasks. Furthermore, seamless integration with existing Python frameworks and libraries is essential to ensure interoperability and facilitate the adoption of the DSL within the target group's existing workflow environments.

1.6 Solution Proposal

In response to the identified needs and challenges within the target group, the solution proposal aims to develop a DSL for efficient manipulation of data structures in Python. This proposal entails a comprehensive

framework that encompasses domain analysis, grammar creation, and implementation using ANTLR.

The solution will revolve around the creation of a Python-based DSL tailored specifically for numerical data manipulation tasks. Leveraging the flexibility and versatility of Python, the DSL will offer a seamless and intuitive interface for developers and data scientists to work with various types of data structures effectively.

The solution will comprise several key components, including:

1. **Domain Analysis:** Conducting an in-depth analysis of the target domains, including scientific computing, data analysis, machine learning, and computational finance. This analysis will involve understanding the specific requirements, challenges, and use cases prevalent within each domain.

2. **Grammar Creation:** Designing a comprehensive grammar for the DSL that reflects the syntax and semantics of Python while incorporating specialized constructs for data structure manipulation. The grammar will be crafted to ensure clarity, conciseness, and ease of use for developers.

3. **Implementation with ANTLR:** Utilizing ANTLR, a powerful parser generator, to implement the DSL. ANTLR will enable the efficient parsing and interpretation of DSL code, facilitating seamless integration with Python codebases and existing libraries.

Overall, the solution proposal aims to provide developers and data scientists with a powerful and flexible tool for manipulating data structures in Python. By addressing the specific needs and challenges of the target group, the proposed DSL will empower users to streamline their workflow, enhance productivity, and tackle complex numerical computations with ease.

1.6.1 Project Objectives

The project is guided by the following key objectives:

- Designing a DSL for data structure manipulation in Python.
- Conducting domain analysis to understand the specific needs of developers and data scientists in numerical computing, scientific computing, data analysis, machine learning, and computational finance domains.
- Creating a grammar specification for the DSL using ANTLR to ensure syntactic correctness and flexibility.
- Implementing the DSL with functionality for manipulating static data structures like arrays and dynamic data structures such as queues, stacks, and linked lists.
- Providing seamless integration of the DSL with existing Python frameworks and libraries commonly used in numerical computing and data analysis, such as NumPy, SciPy, Pandas, and scikit-learn.
- Ensuring the DSL's usability, efficiency, and scalability to handle large-scale numerical datasets and computations.

1.6.2 Functionality

The DSL for data structure manipulation in Python will offer the following functionalities:

- Creation and initialization of arrays with specified dimensions and data types.
- Efficient insertion, deletion, and manipulation of elements in arrays.
- Implementation of common algorithms and operations on arrays, such as sorting, searching, and element-wise operations.
- Dynamic resizing and manipulation of queues and stacks to accommodate changing data requirements.
- Implementation of linked lists with support for single, doubly, and circular linked list variants.

1.6.3 Types of Data

The DSL for data structure manipulation primarily deals with numerical data, specifically integers, which are commonly used in various computational tasks such as scientific computing, data analysis, and machine learning. Integers are whole numbers without any fractional or decimal component, making them suitable for representing discrete quantities and indices in arrays, queues, stacks, and other linear data structures.

1.7 Existing Solutions

Several existing solutions for data structure manipulation exist, each with its own set of strengths and weaknesses. Understanding these can provide insights into areas where improvements can be made in the proposed DSL.

1.7.1 Arrays (C++)

Arrays in C++ are fundamental data structures used to store elements of the same data type in contiguous memory locations. They are declared with a fixed size and can hold elements of any data type, such as integers, characters, or custom objects. Accessing elements in an array is done using zero-based indexing, where the first element has an index of 0, the second has an index of 1, and so on. Here's an example of declaring and accessing elements in an integer array in C++ [9]:

```
#include <iostream>
int main() {
    int arr[5] = {1, 2, 3, 4, 5};

    // Accessing elements of the array
    std::cout << "Element at index 0:" << arr[0] << std::endl;
    std::cout << "Element at index 3:" << arr[3] << std::endl;

    return 0;}
```

1.7.2 Linked Lists (Java)

Java provides a built-in LinkedList class in the java.util package, which implements a doubly linked list. Linked lists in Java allow for dynamic resizing and efficient insertion and deletion of elements. Here's an example of creating and manipulating a linked list of strings in Java [10]:

```
import java.util.LinkedList;

public class Main {
    public static void main(String[] args) {
        // Creating a linked list
        LinkedList<String> linkedList = new LinkedList<>();

        // Adding elements to the linked list
        linkedList.add("Java");
        linkedList.add("Python");
        linkedList.add("C++");

        // Removing an element from the linked list
        linkedList.remove("Python");

        // Displaying elements of the linked list
        System.out.println("Linked List: " + linkedList);
    }
}
```

1.7.3 Stacks (Python)

In Python, stacks can be implemented using lists. The list class provides methods such as append() and pop() which can be used to mimic the behavior of a stack. Here's an example of implementing a stack of integers in Python [11]:

```
stack = []

# Pushing elements onto the stack
stack.append(1)
stack.append(2)
stack.append(3)
```

```
# Popping elements from the stack
print("Popped element:", stack.pop())
print("Popped element:", stack.pop())
```

1.7.4 Queues (C #)

C# provides the Queue class in the System.Collections.Generic namespace for implementing queues. Queues in C# support operations such as enqueueing elements, dequeueing elements, peeking at the front element, and checking if the queue is empty. Here's an example of using a Queue in C# [12]:

```
using System;
using System.Collections.Generic;

class Program {
    static void Main() {
        // Creating a queue
        Queue<int> queue = new Queue<int>();

        // Enqueueing elements
        queue.Enqueue(1);
        queue.Enqueue(2);
        queue.Enqueue(3);

        // Dequeueing elements
        Console.WriteLine("Dequeued element: " + queue.Dequeue());

        // Peeking at the front element
        Console.WriteLine("Front element: " + queue.Peek());
    }
}
```

These examples illustrate how arrays, linked lists, stacks, and queues can be implemented and manipulated in different programming languages, providing developers with versatile tools for managing linear data structures.

1.8 Domain Analysis Conclusions

The domain analysis reveals several key insights into the requirements and challenges of data structure manipulation for numerical data. Understanding the strengths and weaknesses of existing solutions highlights areas where the proposed DSL can provide added value, such as offering a more intuitive syntax for common operations, optimizing performance for numerical computations, and providing seamless integration with existing Python frameworks and libraries. By addressing these needs, the DSL can serve as a valuable tool for developers working with numerical data in various domains.

2 Grammar Description

DSL for data structure manipulation includes grammar rules for defining programs, statements, and operations related to arrays and linked lists. It allows for creating programs with statements such as array operations (insertion, deletion, searching, sorting), linked list operations (insertion, deletion, searching), as well as print statements, insert statements, if statements, for loops, stack operations, and queue operations. Each rule specifies the syntax for performing specific actions or operations on data structures, providing a structured way to manipulate and work with arrays and linked lists within the DSL.

In implementation, was utilized ANTLR to embody the grammar of database manipulation language. ANTLR serves as a powerful parser generator capable of translating formal grammars into functional parsers. Through ANTLR, the grammar specification transcends mere rules, transforming into a tangible tool for parsing and interpreting queries with accuracy and efficiency. By harnessing the capabilities of ANTLR, the team has crafted a grammar that forms the foundation of their database manipulation language, offering a structured framework for seamless query parsing and processing.

2.1 Lexical Consideration

When designing the lexical elements of the DSL grammar, it's important to consider the following aspects:

- **Keywords**: The DSL uses keywords such as `ARRAY`, `LINKEDLIST`, `STACK`, and `QUEUE` to define different data structures and operations. Ensure these keywords are clearly defined and reserved for their specific purposes.
- **Identifiers**: Define rules for identifiers, such as variable names or labels within the DSL. These rules should specify valid characters and any naming conventions that need to be followed.
- **Literals**: Determine the types of literals supported in the DSL, such as integers represented by the `INT` rule. Ensure that the grammar adequately handles these literals.
- **Whitespace Handling**: Specify rules for handling whitespace characters like spaces, tabs, and newlines. In the provided grammar, whitespace is ignored using the `WS` rule.
- **Comments**: Define rules for comments to enhance readability and allow users to add explanatory notes. In this DSL, comments are defined using the `COMMENT` rule.
- **Special Symbols**: Identify and define rules for special symbols or punctuation marks used in the DSL syntax, such as brackets, commas, or semicolons.
- **Error Handling**: Consider how errors and invalid input should be handled in the DSL. Define rules for reporting errors and providing meaningful feedback to users.
- **Reserved Words**: Determine if there are any words that should be reserved and cannot be used as

identifiers. These may include language keywords or future extensions.

By addressing these lexical considerations, a clear and well-defined DSL grammar can be created, ensuring ease of use and understanding for users.

2.2 Reference Grammar

To establish the framework of DSL aimed at facilitating table manipulation within a Python environment, a detailed reference grammar is outlined. This grammar delineates the structural composition of the language, dictating the assembly of statements via the utilization of reserved keywords, data types, and previously articulated syntax. The grammar is articulated using BNF, a formal notation system employed for describing language syntax with precision and clarity.

2.2.1 Production Rule for Data Structure DSL

The grammar consists of various production rules, each defining a symbol in relation to other symbols and literals. Non-terminal symbols, indicated by <symbol>, can be decomposed into sequences comprising terminal symbols (keywords and literals) and additional non-terminal symbols. Terminal symbols are identified by lowercase notation for keywords or designated symbols (e.g., (,), *).

- **Production Rules:**
  - **Program Structure**

    <program> ::= <statement>+
  - **Statements**

    <statement> ::= <arrayStatement> | <linkedListStatement> | <stackStatement> | <queueStatement>
  - **Array Statement**

    <arrayStatement> ::= 'ARRAY' '[' INT (',' INT)* ']' (insert | delete | search | sortArray)? ';'
  - **Linked List Statement**

    <linkedListStatement> ::= 'LINKEDLIST' (insert | delete | search)? ';'
  - **Print Statement**

    <printStatement> ::= 'PRINT' ('ARRAY' | 'LINKED LIST' | 'QUEUE' | 'STACK') ';'
  - **Insert Statement**

    <insertStatement> ::= 'INSERT' ('ARRAY' | 'LINKED LIST' | 'QUEUE' | 'STACK') 'ELEMENT' ';'
  - **If Statement**

    <ifStatement> ::= 'IF' <condition> 'THEN' <action> ('ELSE' <action>)? 'END IF'

    <condition> ::= <expression> ('==' | '!=' | '<' | '>' | '<=' | '>=') <expression>

    <expression> ::= <variable> | <value>

    <action> ::= 'PRINT' <message> ';' | <insertStatement> | | <deleteStatement> | 'LOOP' ('ARRAY' | 'LINKED LIST' | 'QUEUE' | 'STACK') <operation> 'END LOOP'

    <message> ::= <string> | <variable>

- **For Loop**

  <forLoop> ::= ’FOR’ <variable> ’IN’ (’ARRAY’ | ’LINKED LIST’ | ’QUEUE’ | ’STACK’) ’DO’ <action> ’END FOR’

  <action> ::= ’PRINT’ <message> ’;’ | <insertStatement> | <deleteStatement>

  <message> ::= <string> | <variable>

- **Stack Statement**

  <stackStatement> ::= ’STACK’ (pushStack | popStack | topStack | isEmptyStack)? ’;’

- **Queue Statement**

  <queueStatement> ::= ’QUEUE’ (enqueueQueue | dequeueQueue | peekQueue | isFullQueue | isNullQueue)? ’;’

- **Array Operations**

  <insertArray> ::= ’INSERT’ ’[’ INT ’]’ ’INTO’ ’ARRAY’ ’[’ INT ’]’ ’;’

  <deleteArray> ::= ’DELETE’ ’FROM’ ’ARRAY’ ’[’ INT ’]’ ’;’

  <searchArray> ::= ’SEARCH’ ’ARRAY’ ’[’ INT ’]’ ’FOR’ INT ’;’

  <sortArray> ::= ’SORT’ ’ARRAY’ ’[’ INT ’]’ (’ASCENDING’ | ’DESCENDING’) ’;’

- **Linked List Operations**

  <insertLinkedList> ::= ’INSERT’ ’INTO’ ’LINKEDLIST’ ’[’ INT ’]’ ’VALUE’ INT ’;’

  <deleteLinkedList> ::= ’DELETE’ ’FROM’ ’LINKEDLIST’ ’[’ INT ’]’ ’;’

  <searchLinkedList> ::= ’SEARCH’ ’LINKEDLIST’ ’[’ INT ’]’ ’FOR’ INT ’;’

- **Stack Operations**

  <pushStack> ::= ’PUSH’ INT ’TO’ ’STACK’ ’;’

  <popStack> ::= ’POP’ ’FROM’ ’STACK’ ’;’

  <topStack> ::= ’TOP’ ’ELEMENT’ ’OF’ ’STACK’ ’;’

  <isEmptyStack> ::= ’CHECK’ ’IF’ ’STACK’ ’IS’ ’EMPTY’ ’;’

- **Queue Operations**

  <enqueueQueue> ::= ’ENQUEUE’ INT ’TO’ ’QUEUE’ ’;’

  <dequeueQueue> ::= ’DEQUEUE’ ’FROM’ ’QUEUE’ ’;’

  <peekQueue> ::= ’PEEK’ ’FRONT’ ’ELEMENT’ ’OF’ ’QUEUE’ ’;’

  <isFullQueue> ::= ’CHECK’ ’IF’ ’QUEUE’ ’IS’ ’FULL’ ’;’

  <isNullQueue> ::= ’CHECK’ ’IF’ ’QUEUE’ ’IS’ ’EMPTY’ ’;’

Explanation

This reference grammar methodically outlines the DSL’s syntax, elucidating the construction of statements pertinent to data structure manipulation. Each rule encapsulates distinct facets of the language, from

the articulation of data structures and operations to the processes of data insertion, deletion, and querying. Designed for comprehensiveness and intuitiveness, the grammar aims to facilitate ease of use and understanding for DSL users. Through this structured specification, users are empowered to precisely organize their commands for effective manipulation and management of data structures within the DSL environment.

2.2.2 Parsing

Parsing is a crucial process in understanding and interpreting the DSL grammar for data structure manipulation. It involves several key components that work together to transform raw code into a structured representation.

The first step in parsing is Lexical Analysis, performed by the Lexer. During this phase, the input string is divided into tokens, which are the smallest units of meaning in the language. The Lexer handles tasks such as removing whitespace, excluding comments, and categorizing fragments like keywords, identifiers, and literals.

After Lexical Analysis, Syntax Analysis takes place, performed by the Parser. This phase involves analyzing the token stream produced by the lexer against the grammar rules of the DSL. The Parser ensures syntactic correctness and constructs a hierarchical structure known as a parse tree or AST. This structure captures the syntactic organization of the program and its nested relationships.

The Parse Tree or AST serves as a fundamental representation of the code's syntactic structure. While the parse tree accurately reflects the code's structure, the AST provides a more abstract view, focusing on the logical and grammatical aspects of the syntax. Together, these components facilitate machine understanding and interpretation of DSL programs, enabling efficient execution and processing of code.

Parsing Example

Consider the following code snippet in our data structure DSL:

```
arr = ARRAY[1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
PRINT arr;
INSERT[11] INTO arr;
PRINT arr;
INSERT[int] INTO arr;
PRINT arr;
DELETE[10] FROM arr;
PRINT arr;
isThere = SEARCH[4] IN arr;
PRINT isThere;
SORT arr DESCENDING;
PRINT arr;
```

Lexical Analysis: The lexer converts the code into tokens:

The lexer tokenizes the code as follows: Keyword('arr'), Assign('='), Keyword('ARRAY'), LeftSquareBracket('['), Number('1'), Comma(','), Number('2'), Comma(','), Number('3'), Comma(','), Number('4'), Comma(','), Number('5'), Comma(','), Number('6'), Comma(','), Number('7'), Comma(','), Number('8'), Comma(','), Number('9'), Comma(','), Number('10'), RightSquareBracket(']'), Semicolon(';'), Keyword('PRIN Identifier('arr'), Semicolon(';'), Keyword('INSERT'), LeftSquareBracket('['), Number('11'), RightSquareBracket(']'), Keyword('INTO'), Identifier('arr'), Semicolon(';'), Keyword('PRINT'), Identifier('arr'), Semicolon(';'), Keyword('INSERT'), LeftSquareBracket('['), Identifier('int'), RightSquareBracket(']'), Keyword('INTO'), Identifier('arr'), Semicolon(';'), Keyword('PRINT'), Identifier('arr'), Semicolon(';'), Keyword('DELETE'), LeftSquareBracket('['), Number('10'), RightSquareBracket(']'), Keyword('FROM'), Identifier('arr'), Semicolon(';'), Keyword('PRINT'), Identifier('arr'), Semicolon(';'), Identifier('isThere'), Assign('='), Keyword('SEARCH'), LeftSquareBracket('['), Number('4'), RightSquareBracket(']'), Keyword('IN'), Identifier('arr'), Semicolon(';'), Keyword('PRINT'), Identifier('isThere'), Semicolon(';'), Keyword('SORT'), Identifier('arr'), Keyword('DESCENDING'), Semicolon(';'), Keyword('PRINT'), Identifier('arr'), Semicolon(';') Syntax Analysis:

The parser examines the token stream against the grammar rules and constructs an AST [6]. A simplified representation is as follows: Program [ArrayDeclaration, [Number: '5', Number: '10', Number: '15', Number: '20']] [SearchInArray, [Index: Number: '3', Value: Number: '15']]

Syntax Analysis: The parser analyzes the token stream against the grammar rules and constructs an AST. Here's a simplified representation:

```
Program
    [CreateArray, [Identifier: 'arr', Elements: [Number: '1', Number: '
       2', Number: '3', Number: '4', Number: '5', Number: '6', Number:
       '7', Number: '8', Number: '9', Number: '10']]]
    [PrintArray, [Identifier: 'arr']]
    [InsertIntoArray, [Identifier: 'arr', Value: Number: '11']]
    [PrintArray, [Identifier: 'arr']]
    [InsertIntoArray, [Identifier: 'arr', Value: Identifier: 'int']]
    [PrintArray, [Identifier: 'arr']]
    [DeleteFromArray, [Identifier: 'arr', Value: Number: '10']]
    [PrintArray, [Identifier: 'arr']]
    [SearchInArray, [Identifier: 'arr', Value: Number: '4']]
    [PrintValue, [Identifier: 'isThere']]
```

```
[SortArray, [Identifier: 'arr', Order: 'DESCENDING']]
[PrintArray, [Identifier: 'arr']]
```

The root of the tree is a Program node, with each statement represented as a child node.
Each statement node further branches into specific operation nodes, such as CreateArray, PrintArray, InsertIntoArray, DeleteFromArray, SearchInArray, and SortArray.
The CreateArray node contains information about the array's identifier ('arr') and its elements ('1', '2', '3', '4', '5', '6', '7', '8', '9', '10').
The PrintArray node specifies the array ('arr') to be printed.
The InsertIntoArray node specifies the value ('11') to be inserted into the array ('arr').
The InsertIntoArray node also specifies the identifier ('int') to be inserted into the array ('arr').
The DeleteFromArray node indicates the value ('10') to delete from the array ('arr').
The SearchInArray node denotes the value ('4') to search for within the array ('arr').
The SortArray node indicates the array ('arr') to be sorted in 'DESCENDING' order.
The PrintValue node specifies the variable ('isThere') to be printed.

The AST represents the syntactic structure of the DSL code, facilitating further analysis, interpretation, and execution of the program.

This parsing example illustrates how the DSL code is parsed and understood by the computer, transitioning from a sequence of tokens to a structured representation that captures the hierarchical nature of data structure manipulation operations.

2.2.3 Semantic Rules

The semantic rules of the DSL provide a framework for understanding the behavior and functionality of the language constructs within the context of manipulating data structures. These rules define the expected actions and outcomes of statements and operations performed on arrays, linked lists, stacks, and queues, ensuring clarity and correctness in data manipulation processes.

Array Operations

- **Array Insertion (insertArray):**
  Inserts an integer value into the array at the specified index.
- **Array Deletion (deleteArray):**
  Deletes an element from the array at the specified index.
- **Array Search (searchArray):**
  Searches for a specified integer value within the array and returns the index if found.
- **Array Sorting (sortArray):**
  Sorts the elements of the array in ascending or descending order.

Linked List Operations

- **Linked List Insertion (insertLinkedList):**

  Inserts an integer value into the linked list at the specified position.

- **Linked List Deletion (deleteLinkedList):**

  Deletes an element from the linked list at the specified position.

- **Linked List Search (searchLinkedList):**

  Searches for a specified integer value within the linked list and returns the position if found.

Stack Operations

- **Stack Push (pushStack):**

  Pushes an integer value onto the top of the stack.

- **Stack Pop (popStack):**

  Pops the top element from the stack.

- **Stack Top (topStack):**

  Retrieves the top element of the stack without removing it.

- **Stack Empty Check (isEmptyStack):**

  Checks whether the stack is empty.

Queue Operations

- **Queue Enqueue (enqueueQueue):**

  Enqueues an integer value into the queue.

- **Queue Dequeue (dequeueQueue):**

  Dequeues an element from the front of the queue.

- **Queue Peek (peekQueue):**

  Retrieves the front element of the queue without removing it.

- **Queue Full Check (isFullQueue):**

  Checks whether the queue is full.

- **Queue Empty Check (isNullQueue):**

  Checks whether the queue is empty.

These semantic rules define the expected behavior of operations in the DSL, ensuring consistency and correctness in data manipulation tasks performed using the language constructs.

2.2.4 Types

In the DSL, the grammar primarily deals with integers, which play a crucial role in specifying array indices and values in various operations. The DSL supports operations such as array insertion, deletion, searching, sorting, linked list insertion, deletion, searching, stack operations, and queue operations, all of

which involve integer parameters for specifying positions, values, or conditions.

2.2.5 Parsing Tree

This parse tree represents the syntactic structure of a domain-specific language (DSL) designed for data structure manipulation. The DSL supports operations such as array creation, pushing elements into an array, and printing the array contents. The parse tree visually breaks down these operations into their constituent components, demonstrating how the language's grammar processes each statement. This visual representation aids in understanding the hierarchical relationships and the sequence of operations defined in the program.

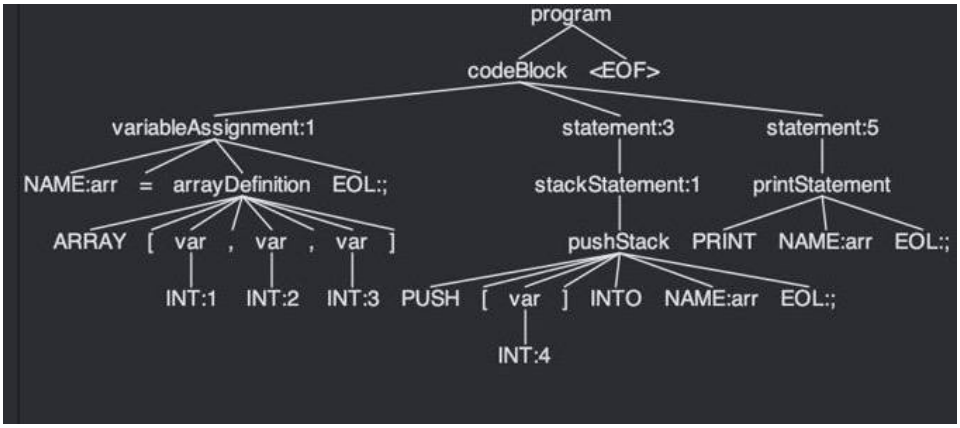

Figure 2.2.1 - Parsing tree

This parse tree represents the syntactic structure of a domain-specific language (DSL) designed for data structure manipulation. The DSL supports operations such as array creation, pushing elements into an array, and printing the array contents. The parse tree visually breaks down these operations into their constituent components, demonstrating how the language's grammar processes each statement. This visual representation aids in understanding the hierarchical relationships and the sequence of operations defined in the program.

This parse tree represents the syntactic structure of a domain-specific language designed for data structure manipulation. The DSL supports operations such as array creation, pushing elements into an array, and printing the array contents. The parse tree visually breaks down these operations into their constituent components, demonstrating how the language's grammar processes each statement. This visual representation aids in understanding the hierarchical relationships and the sequence of operations defined in the program.
Example:
arr = ARRAY [1, 2, 3]
PUSH [4] into arr;
PRINT arr;

3 Grammar Implementation

In the implementation of the DSL grammar, ANTLR was utilized to define the syntax and structure of the language. ANTLR serves as a powerful parser generator capable of transforming formal grammar specifications into functional parsers. With ANTLR, the grammar specification becomes more than just a set of rules; it becomes a practical tool for parsing and interpreting queries accurately and efficiently. By leveraging ANTLR, the development team has crafted a grammar that forms the backbone of the DSL, providing a structured framework for parsing and processing statements related to various data structures.

3.0.1 Grammar Definition

The provided grammar specification defines the syntax for manipulating arrays, linked lists, stacks, and queues within the DSL. It includes rules for array operations such as insertion, deletion, searching, and sorting, as well as operations for linked lists, stacks, and queues. Each operation is defined with its corresponding syntax and semantics to ensure precise interpretation and execution.

Code Implementation:

grammar DataStructureDSL;

// Parser rules

program: codeBlock+ EOF;

statement: arrayStatement | linkedListStatement | stackStatement | queueStatement | printStatement;

arrayDefinition: 'ARRAY' '[' (var | NAME)* (',' (var | NAME))* ']';

linkedListDefinition: 'LINKEDLIST' '[' ']';

stackDefinition: 'STACK' '[' (INT — NAME) ']';

queueDefinition: 'QUEUE' '[' (INT — NAME) ']';

printStatement: 'PRINT' NAME EOL;

variableAssignment: NAME '=' arrayDefinition EOL

| NAME '=' linkedListDefinition EOL

| NAME '=' stackDefinition EOL

| NAME '=' queueDefinition EOL

| NAME '=' search

| NAME '=' popStack

| NAME '=' dequeueQueue

| NAME '=' peek

| NAME '=' isFull

| NAME '=' isEmpty

| NAME ’=’ var EOL

| NAME ’=’ NAME EOL;

ifStatement: ’IF’ condition ’THEN’ statement (’ELSE’ statement)?;

forStatement: ’FOR’ loopCondition ’DO’ statement;

arrayStatement: insert | insertAt | delete | search | sortArray ;

linkedListStatement: insert | delete | search ;

stackStatement: pushStack | popStack | peek | isFull | isEmpty ;

queueStatement: enqueueQueue | dequeueQueue | peek | isFull | isEmpty ;

// Array operations

insert: ’INSERT’ ’[’ (var — NAME) ’]’ ’INTO’ NAME EOL;

delete: ’DELETE’ ’[’ (var — NAME) ’]’ ’FROM’ NAME EOL;

search: ’SEARCH’ ’[’ (var — NAME) ’]’ ’IN’ NAME EOL;

insertAt: ’INSERT’ ’[’ (var — NAME) ’]’ ’INTO’ NAME ’AT’ INT EOL;

// Corrected ascending/descending // Stack operations

pushStack: ’PUSH’ ’[’ (var — NAME) ’]’ ’INTO’ NAME EOL;

popStack: ’POP’ ’FROM’ NAME EOL;

// Queue operations

enqueueQueue: ’ENQUEUE’ INT ’TO’ ’QUEUE’ ’;’;

dequeueQueue: ’DEQUEUE’ ’FROM’ ’QUEUE’ ’;’;

isFull: NAME ’IS’ ’FULL’ EOL;

isEmpty: NAME ’IS’ ’EMPTY’ EOL;

peek: ’PEEK’ NAME EOL;

var: INT — FLOAT — CHAR — STRING — BOOL;

// Print, if, and for statements

condition: expression ((’<’ | ’>’ | ’==’ | ’!=’ | ’<=’ | ’>=’) expression);

loopCondition: INT;

// Lexer rules

INT: [0-9]+;

WS: [\t \r \n]+ - > skip;

// Comments

COMMENT: ’//’ [\r \n]* - > skip;

4 Implementation

In this chapter, it is provided a detailed explanation of the implementation of the data structures: `LinkedList`, `Stack`, `Queue`, and `Array`. These data structures are implemented as a Python library that can be imported and used in other Python projects. Each class was designed with methods for various operations such as insertion, deletion, searching, and manipulation. Exception handling was incorporated to manage scenarios like full stacks/queues or invalid index accesses in data structures like linked lists.

4.1 LinkedList

The `LinkedList` class is implemented as a nested class within the main `LinkedList` class. Each node of the linked list is represented by the `Node` class, which contains the data and a reference to the next node in the list.

4.1.1 Implementation Details

The `LinkedList` class provides methods to perform common operations on a linked list:

- `insert(data)`: Inserts a new node with the given data at the beginning of the list.
- `delete()`: Deletes the first node from the list.
- `insert_at_index(data, index)`: Inserts a new node with the given data at the specified index.
- `delete_at_index(index)`: Deletes the node at the specified index.
- `search(index)`: Returns the data at the specified index.
- `__str__()`: Returns a string representation of the list.
- `__repr__()`: Returns a string representation of the list (used for debugging).
- `__len__()`: Returns the number of nodes in the list.
- `__iter__()`: Allows iteration over the elements of the list.
- `__getitem__(index)`: Allows indexing to access elements of the list.
- `__bool__()`: Returns `True` if the list is not empty, `False` otherwise.

4.1.2 Usage

To use the `LinkedList` class, import it from the library and create an instance:

```
# Create a new linked list
linked_list = LinkedList()

# Insert elements into the linked list
linked_list.insert(1)
linked_list.insert(2)
```

```
#Delete an element from linked list
linked_list.delete()

# Print the linked list
print(linked_list)  # Output: [2, 1]
```

4.2 Stack

The `Stack` class represents a last-in, first-out (LIFO) data structure.

4.2.1 Implementation Details

The `Stack` class provides methods to perform common operations on a stack:

- `push(data)`: Pushes the given data onto the stack.
- `pop()`: Removes and returns the top element from the stack.
- `peek()`: Returns the top element of the stack without removing it.
- `is_empty()`: Returns `True` if the stack is empty, `False` otherwise.
- `is_full()`: Returns `True` if the stack is full (reached its maximum size), `False` otherwise.
- `__str__()`: Returns a string representation of the stack.
- `__repr__()`: Returns a string representation of the stack (used for debugging).
- `__len__()`: Returns the number of elements in the stack.
- `__bool__()`: Returns `True` if the stack is not empty, `False` otherwise.

4.2.2 Usage

To use the `Stack` class, import it from the library and create an instance:

```
from manipula import Stack

# Create a new stack
stack = Stack()

# Push elements onto the stack
stack.push(1)
stack.push(2)

# Pop an element from the stack
element = stack.pop()
print(element) # Output: 3
```

```
# Print the stack
print(stack)  # Output: [1, 2]
```

4.3 Queue

The `Queue` class represents a first-in, first-out (FIFO) data structure.

4.3.1 Implementation Details

The `Queue` class provides methods to perform common operations on a queue:

- `enqueue(data)`: Enqueues the given data into the queue.
- `dequeue()`: Dequeues and returns the front element from the queue.
- `peek()`: Returns the front element of the queue without removing it.
- `is_empty()`: Returns `True` if the queue is empty, `False` otherwise.
- `is_full()`: Returns `True` if the queue is full (if a maximum size is specified and the number of elements in the queue equals the maximum size), `False` otherwise.
- `__str__()`, `__repr__()`: Returns a string representation of the queue.
- `__len__()`: Returns the number of elements in the queue.
- `__bool__()`: Returns `True` if the queue is not empty, `False` otherwise.

4.3.2 Usage

To use the `Queue` class, import it from the library and create an instance:

```
from manipula import Queue

# Create a new queue
queue = Queue()

# Enqueue elements into the queue
queue.enqueue(1)
queue.enqueue(2)

# Dequeue an element from the queue
element = queue.dequeue()
print(element) # Output: 1

# Print the queue
print(queue)  # Output: [1, 2]
```

4.4 Array

The `Array` class represents a dynamic array.

4.4.1 Implementation Details

The `Array` class provides methods to perform common operations on an array:

- `append(item)`: Appends the given item to the end of the array.
- `insert(item, index=None)`: Inserts the item at the specified index. If no index is provided, appends the item to the end of the array.
- `remove(item)`: Removes the first occurrence of the given item from the array.
- `sort()`: Sorts the array in ascending order.
- `reverse()`: Reverses the elements of the array.
- `__getitem__(index)`: Allows indexing to access elements of the array.
- `__setitem__(index, value)`: Allows assignment to modify elements of the array.
- `__len__()`: Returns the number of elements in the array.
- `__iter__()`: Allows iteration over the elements of the array.
- `__contains__(item)`: Returns `True` if the item is present in the array, `False` otherwise.
- `__bool__()`: Returns `True` if the array is not empty, `False` otherwise.
- `search(item)`: Returns the index of the first occurrence of the item in the array. Raises an exception if the item is not found.

4.4.2 Usage

To use the `Array` class, import it from the library and create an instance:

```
from manip import Array

# Create a new array
array = Array()

# Append elements to the array
array.append(1)
array.append(2)

# Remove an element from the array
array.remove(2)

# Print the array
print(array)  # Output: [1, 2]
```

Conclusions

The domain analysis presented in this study offers a comprehensive understanding of the multifaceted challenges inherent in software development, with a particular emphasis on the manipulation and presentation of data structures. It underscores the critical role played by specialized tools in addressing various concerns such as efficiency, scalability, complexity, flexibility, and correctness. Traditional programming languages often fall short in providing built-in support for efficient data manipulation, resulting in the creation of verbose and error-prone code that significantly impacts developers' productivity and the overall quality of software products.

In response to these challenges, the proposed Domain-Specific Language (DSL) for data structure manipulation emerges as a promising solution. By providing specialized constructs specifically tailored to numerical data manipulation tasks, the DSL aims to streamline common operations, allowing developers to concentrate on core application logic and functionality. Through an intuitive syntax, powerful abstractions, and graphical representations, the DSL seeks to redefine the way developers interact with and manage data structures, fostering a culture of innovation and collaboration across diverse domains within the software development landscape.

The proposed DSL represents a significant step forward in addressing the pressing need for more efficient and expressive tools for data manipulation. By abstracting away low-level implementation details and providing high-level abstractions, the DSL empowers developers to write concise and expressive code, thereby reducing the potential for errors and improving overall productivity. Furthermore, the graphical representations offered by the DSL enhance developers' understanding of complex data structures, facilitating better communication and collaboration within development teams.

In conclusion, the domain analysis underscores the imperative for a specialized DSL dedicated to data structure manipulation. Such a language has the potential to significantly enhance developers' productivity, improve code quality, and accelerate innovation in software development practices. As the software development landscape continues to evolve, the adoption of specialized tools like the proposed DSL is essential to meet the growing demands for efficient and robust data manipulation solutions.

Bibliography

[1] Tamassia, R. (2013). Data Structures and Algorithms (6th Edition). Pearson.

[2] GeeksforGeeks. (Year of Publication). Introduction to Linear Data Structures. Retrieved from `https://www.geeksforgeeks.org/introduction-to-linear-data-structures/`.

[3] Gu, J. (1998). Linear Data Structures and Their Implementation. Springer.

[4] Brassard, G., Bratley, P. (2008). Algorithmics: Theory and Practice. Prentice Hall.

[5] Harris, M. (2002). Data Structures and Algorithms: Arrays. In Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX) (pp. 42-48).

[6] Weiss, M. A. (2013). Data Structures and Algorithm Analysis in Java (3rd Edition). Pearson.

[7] Sedgewick, R. (2011). Algorithms (4th Edition). Addison-Wesley Professional.

[8] Knuth, D. E. (1997). The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition). Addison-Wesley Professional.

[9] Bentley, J. L., Yao, A. C. (1982). An Almost Optimal Algorithm for Unbounded Searching. Information Processing Letters, 14(4), 170-173.

[10] Drozdek, A. (2012). Data Structures and Algorithms in C++ (4th Edition). Cengage Learning.

[11] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). Introduction to Algorithms (3rd Edition). The MIT Press.

[12] Tarjan, R. E. (1983). Data Structures and Network Algorithms. Society for Industrial and Applied Mathematics.