

**Data Structure Course Projects Report**

Class：19G231

Student ID：20231003496

Name：Liner Zhang

Date：2024\9\9

Table of Contents

[Maze Problem 2](#_Toc11868)

[1.Course Design Topics and Requirements 2](#_Toc20253)

[2.Requirement Analysis 2](#_Toc1201)

[3.Design 3](#_Toc16475)

[3.1 Design Concept 3](#_Toc19544)

[3.1.1 Data Structure Design 3](#_Toc12004)

[3.1.2 Algorithm Design 3](#_Toc26986)

[3.2 Design Presentation 5](#_Toc29254)

[3.2.1 Non-recursive Program 5](#_Toc20885)

[3.2.2 Recursive Program 5](#_Toc7280)

[3.3 Detailed Design 6](#_Toc15929)

[3.3.1 Non-recursive Program 6](#_Toc23489)

[3.3.2 Recursive Program 10](#_Toc7344)

[3.3.3 External Program 12](#_Toc26300)

[4. Debugging Analysis 17](#_Toc5565)

[5. User Manual 18](#_Toc1595)

[6. Test Data And Test Results 18](#_Toc3434)

[7. Source Program Inventory 22](#_Toc14933)

[Application of Binary Tree -– Expression Evaluation 34](#_Toc18142)

[1.Course Design Topics and Requirements 34](#_Toc5970)

[2.Requirement Analysis 34](#_Toc29691)

[3.Design 35](#_Toc9058)

[3.1 Design Concept 35](#_Toc30395)

[3.1.1 Data Structure Design 35](#_Toc756)

[3.1.2 Algorithm Design 35](#_Toc2729)

[3.2 Design Presentation 37](#_Toc28897)

[3.3 Detailed Design 38](#_Toc32484)

[3.3.1 Basic Functionality Program 38](#_Toc10436)

[3.3.2 External Program 46](#_Toc11672)

[4. Debugging Analysis 47](#_Toc23859)

[5.User Manual 48](#_Toc970)

[6.Test Data And Test Results 48](#_Toc24767)

[7.Source Program Inventory 51](#_Toc17636)

# Maze Problem

## 1.Course Design Topics and Requirements

【**Problem Description**】

A rectangular matrix of m\*n represents the maze, and 0 and 1 represent the pathway and obstacle in the maze, respectively. Design a program to find a path from the entrance to the exit for an arbitrarily maze, or to conclude that there is no pathway.

【**Basic Requirements**】

(1) Write a stack-based non-recursive program to solve the maze problem. The pathway is output as a triplet (i, j, d), where (i, j) indicates a coordinate in the maze and d represents the direction to the next coordinate.

(2) Write a recursive program to solve the maze problem.

(3) Several data sets are used for testing, and the size of the data sets grows from small to large, i.e., the grid is getting smaller and the obstacles are getting more complex.

【**Extended Requirements**】

Realize the visualization interface of the maze problem. You can step out of the maze with a mouse click, or end up with a message box with no pathway.

## 2.Requirement Analysis

【Basic functional requirements】

(1) Enter the length and width of the maze, the entrance and exit of the maze, and the specific walls and pathways of the maze.

(2) Determine whether the maze has a solution and output the specific path of the maze.

【User interface requirements】

(1) Draw the maze when there is a solution, and display the maze path step by step by clicking with the mouse.

(2) Users can directly clear the data, redesign the maze, and solve it by pressing the button.

## 3.Design

### 3.1 Design Concept

#### 3.1.1 Data Structure Design

(1) Stack: Based on the requirements of the problem, we need to design a storage structure that utilizes a stack to represent the correct path. Given the Last-In-First-Out (LIFO) characteristic of a stack, we start by pushing the exit coordinates onto the stack as the starting point. We then search for paths from the exit coordinate to the entrance coordinate, pushing each visited coordinate onto the stack. Finally, by popping the elements from the stack in sequence, we can obtain the correct path from the entrance to the exit.

(2) Two dimensional array: In order to effectively represent the maze structure, we utilize a two-dimensional array. This array aligns with the maze’s layout of rows and columns. Each element in the first dimension of the array corresponds to a row of the maze, while each element in the second dimension represents an individual cell within that row.

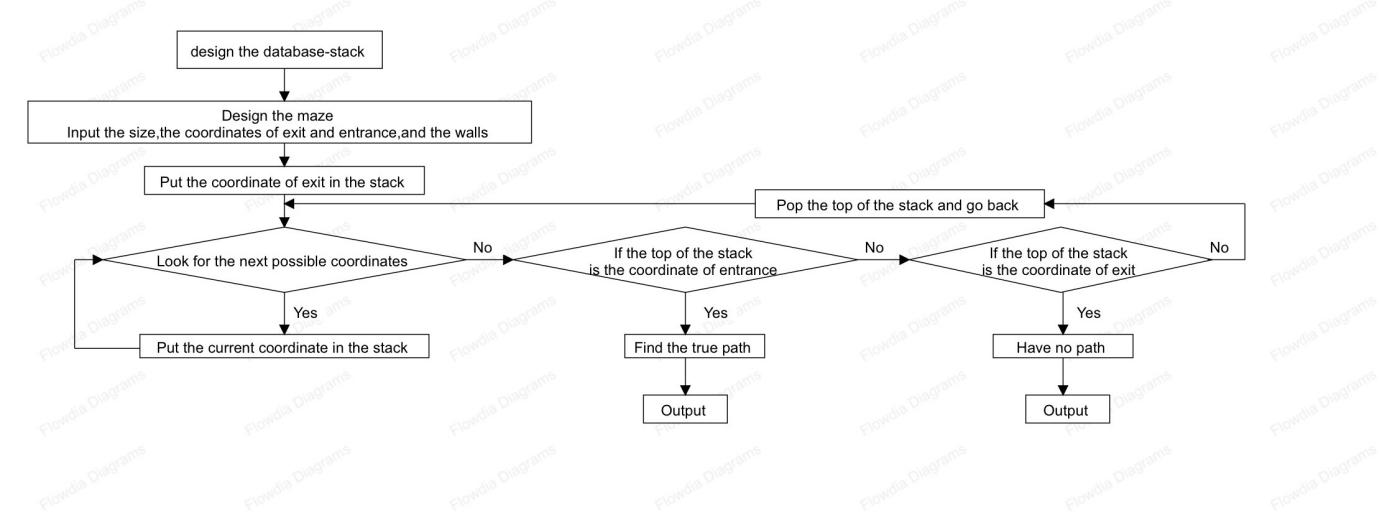
(3) List: To track whether cell coordinates have been visited, we use a list to store all coordinates that have been traversed. This list serves as a record of the cells that have already been checked during the maze traversal process.

#### 3.1.2 Algorithm Design

1. Non-Recursive Program

Initially, place the exit coordinates onto the stack, with the top of the stack representing the current coordinates. The process involves using a while loop to continuously update the current coordinates by modifying the top elements of the stack. Within this loop, the current coordinate is first recorded in a list to mark it as visited. Next, an if structure is employed to identify all passable neighboring coordinates. If such coordinates are found, they are pushed onto the stack, making them the new top coordinates for further exploration. This means the current coordinate is revisited and re-evaluated. When the current coordinate matches the entrance coordinate, the loop terminates, indicating that the correct path has been found. Conversely, if no passable neighboring coordinates are available, the current coordinate is removed from the stack, and the algorithm backtracks to the previous top coordinate for re-evaluation. If the process returns to the exit coordinate and no valid path has been found, the loop ends, signifying that no viable path exists.

The flowchart for the non-recursive programming approach is illustrated in Figure 1.1.



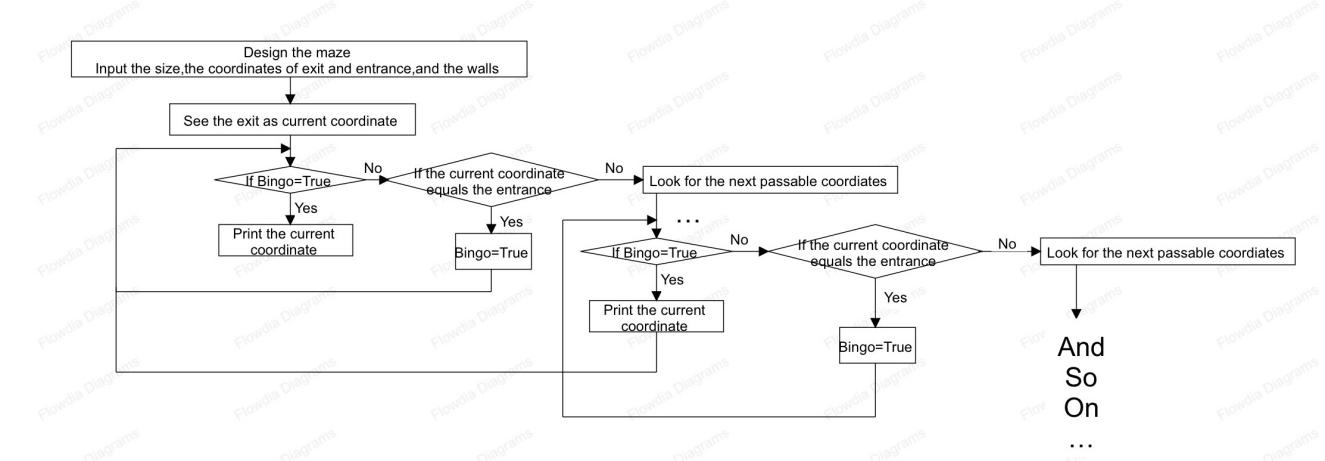
**Figure 1.1 Non recursive program flowchart for maze problem**

1. Recursive Program

Since the filtering process for the next coordinate is repeatedly performed, it is conceptualized as a recursive function with the current coordinate as the parameter. In recursive programs, output order proceeds from the innermost layer to the outermost layer. Here, the exit coordinate is considered the outermost layer, while the entrance coordinate is the innermost layer. When the current coordinate matches the entrance coordinate, it indicates that the correct path has been found, and the function returns `True` as the base case.

If the current coordinate is not the entrance coordinate, the function uses an `if` structure to identify all passable neighboring coordinates. Each of these potential coordinates is then passed into the function for evaluation. The return value from each recursive call is propagated back through the layers. When the innermost layer eventually reaches the entrance coordinate, it returns `True`, which then propagates outward through each layer, resulting in the overall function returning `True` when the correct path is found.

The flowchart for the recursive programming approach is illustrated in Figure 1.2.

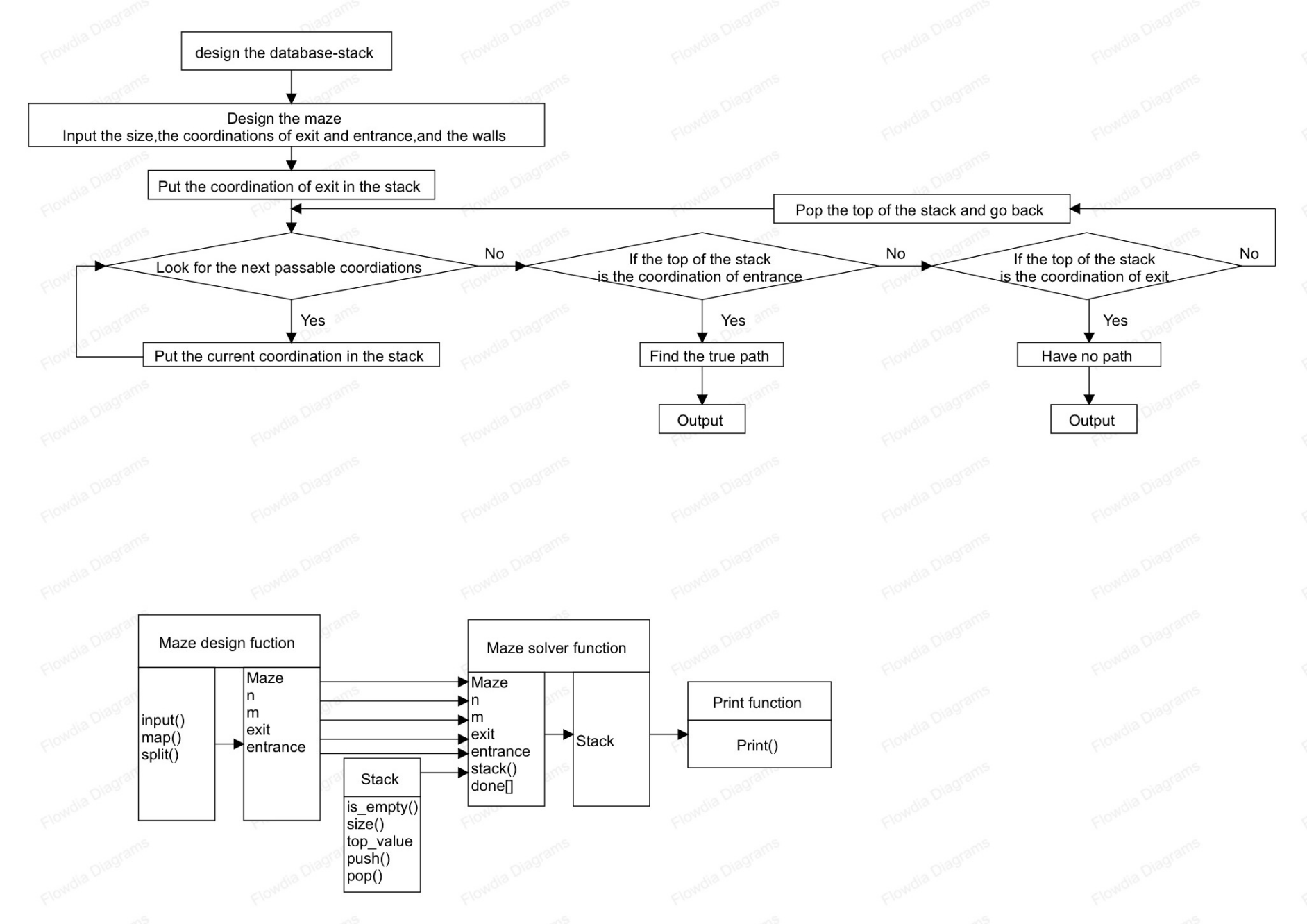


**Figure 1.2 Recursive program flowchart for maze problem**

### 3.2 Design Presentation

#### 3.2.1 Non-recursive Program

The calling relationship of the main functions is shown in Figure 1.3.

**Figure 1.3 Main function call relationship diagram of non recursive program**

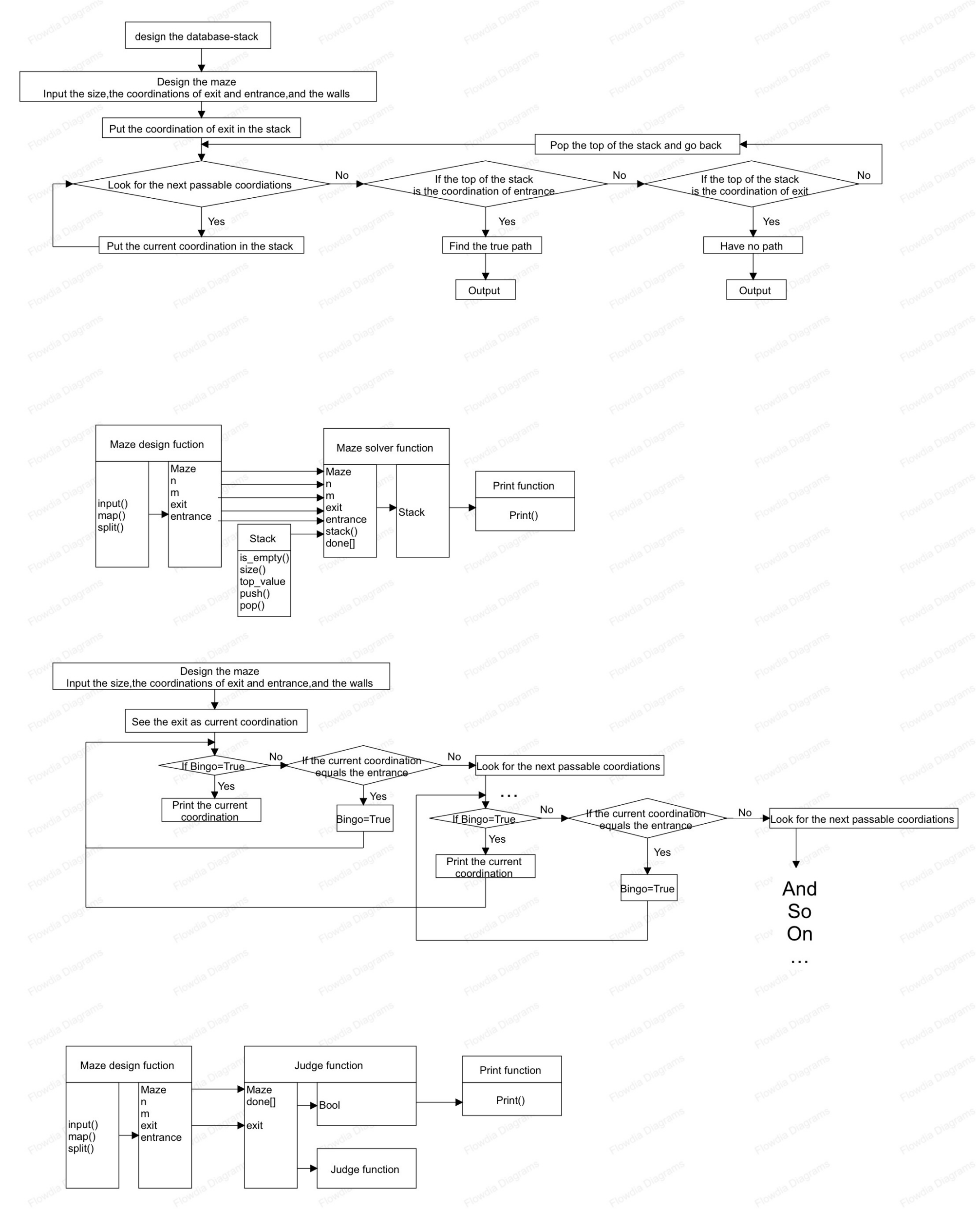
(1) Stack module: is\_ empty(), returns a Boolean value indicating whether the stack is empty; Size() returns the number of elements in the stack; Top-value(), returns the top element of the stack, throws an exception if the stack is empty; Push (value), placing a value at the top of the stack; Pop(), delete the top element of the stack and return. If the stack is empty, throw an exception.

(2) Maze design module: input(), receives user input; Map(), allocate user input content; Split() divides user input content into segments; N and m, integers store the length and width of the maze entered by the user; Exit and entrance, the list stores the coordinates of the maze exit and entrance entered by the user, and the elements are integers; Maze,A two-dimensional array stores the specific content of the maze entered by the user.

(3) Maze solving module: ST, stack class stores the correct path; done, List the accessed coordinate points; next\_coordinations, The temporary stack stores possible coordinates for the next step.

#### 3.2.2 Recursive Program

The calling relationship of the main functions is shown in Figure 1.3.

**Figure 1.4 Main function call relationship diagram of recursive program**

(1) Maze design module: input(), receives user input; Map(), allocate user input content; Split() divides user input content into segments; N and m, integers store the length and width of the maze entered by the user; Exit and entrance, the list stores the coordinates of the maze exit and entrance entered by the user, and the elements are integers; Maze, A two-dimensional array stores the specific content of the maze entered by the user.

(2) Coordinate judgment module: done, storing a list of accessed coordinate points; if-else, Select the statement to find all possible path coordinates; Print(), output the correct path coordinates; Return Boolean value.

### 3.3 Detailed Design

#### 3.3.1 Non-recursive Program

1. Stack Class Design

stack uses a list (`data`) to store its elements and an integer (`top`) to represent the index of the top element, which is initialized to -1.

The `is\_empty()` function determines whether the stack is empty by checking if `top` is less than 0. If `top` is less than 0, it indicates that the stack is empty, and the function returns `True`. Otherwise, it returns `False`.

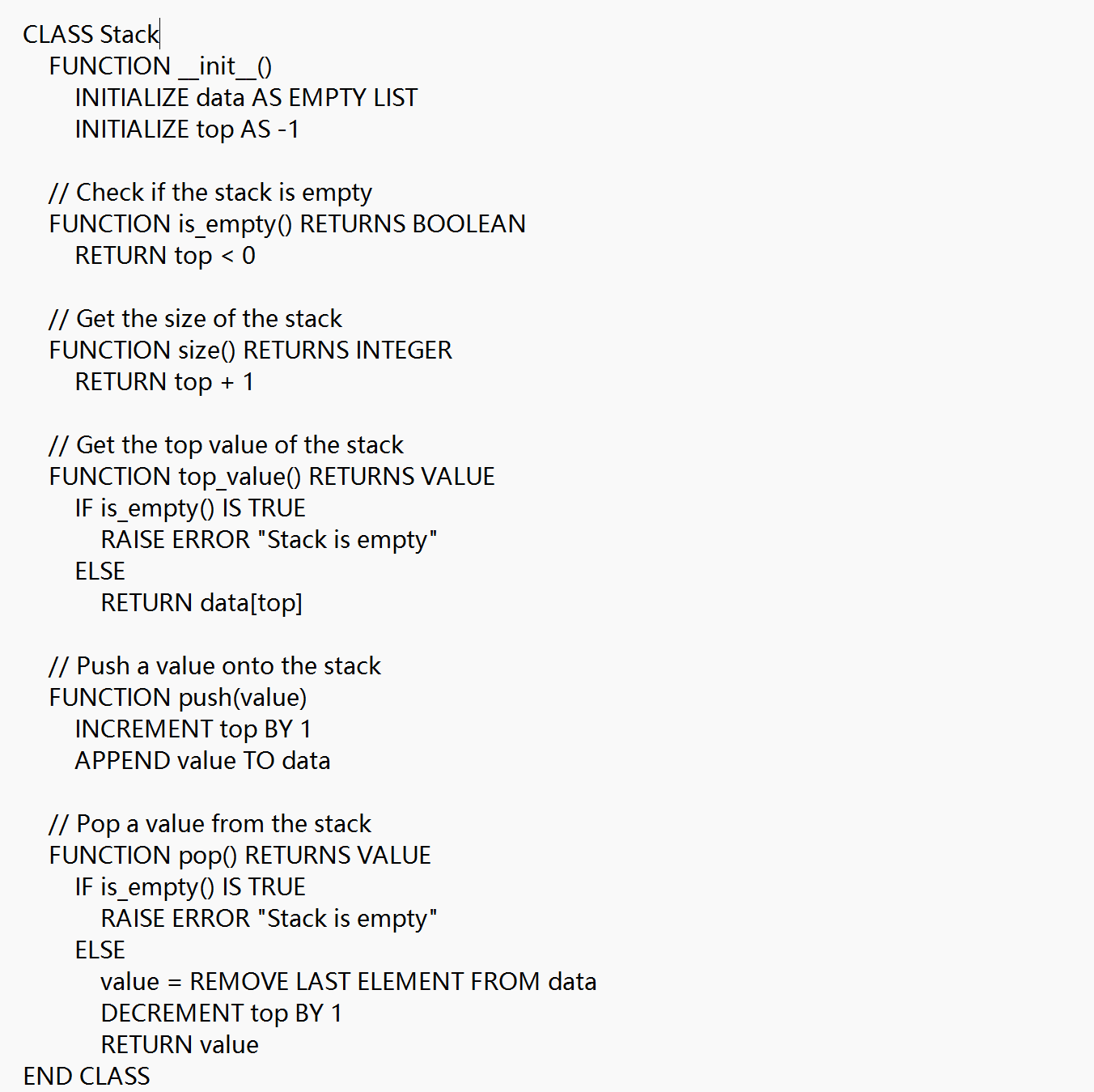
The `size()` function returns the number of elements in the stack. Since `top` represents the index of the last element, adding 1 to `top` gives the count of elements, or it can simply return the length of the `data` list.

The `top\_value()` function returns the current top element, which is the last element in the `data` list. It throws an exception if the stack is empty.

The `push(value)` function adds a new value to the top of the stack. It appends the value to the end of the `data` list and increments the `top` index by 1.

The `pop()` function removes and returns the top element of the stack. It deletes the last element from the `data` list and decrements the `top` index by 1. If the stack is empty, it throws an exception.

The pseudo code for these operations is illustrated in Figure 1.5.

**Figure 1.5 Pseudo code for Stack Class Design**

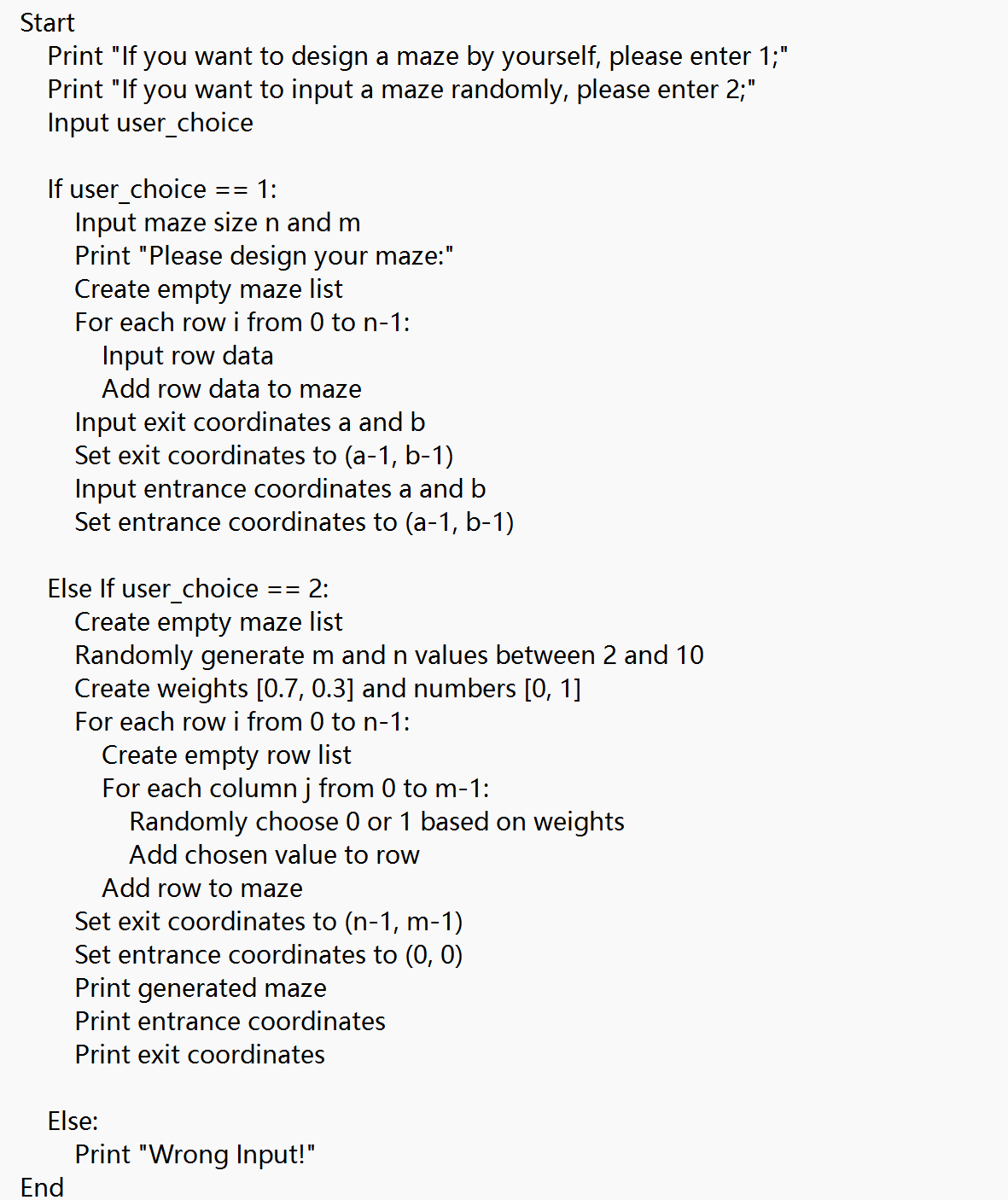
1. Input Design

At the start of the program, print a prompt asking the user to choose between manually entering a maze or generating one automatically. If the user inputs 1 for manual entry, print a prompt to ask for the maze size, read the user's input, and store the maze's length and width as `n` and `m`. Print a prompt to ask for the maze paths and walls, initialize a 2D array `maze` for maze storage, traverse the previously read rows, input and read each row's content, temporarily store it in the list `row`, and add it to the 2D array `maze`. Print a prompt to ask for the maze exit coordinates, read the user's input, and store the exit coordinates as `a` and `b`, combining them into the list `exit`. Print a prompt to ask for the maze entrance coordinates, read the user's input, and store the entrance coordinates as `a` and `b`, combining them into the list `entr`.

If the user inputs 2 for automatic generation, initialize a 2D array `maze` for maze storage, use the `random` library to generate random integers between 2 and 10 for the maze's length and width, stored as `n` and `m`. Use nested loops to traverse the rows, initializing the list `row` for each row. Within each row, traverse the columns and randomly generate numbers between 0 and 1 in a 7:3 ratio, adding them to the list `row`. After column traversal, add `row` to `maze`. The exit defaults to coordinates `[n-1, m-1]` and the entrance defaults to `[0, 0]`. Output the maze and the entry/exit points after assignment.

If the user inputs anything else, report an error.

The pseudo code for this process is shown in Figure 1.6.

**Figure 1.6 Pseudo code for maze input design**

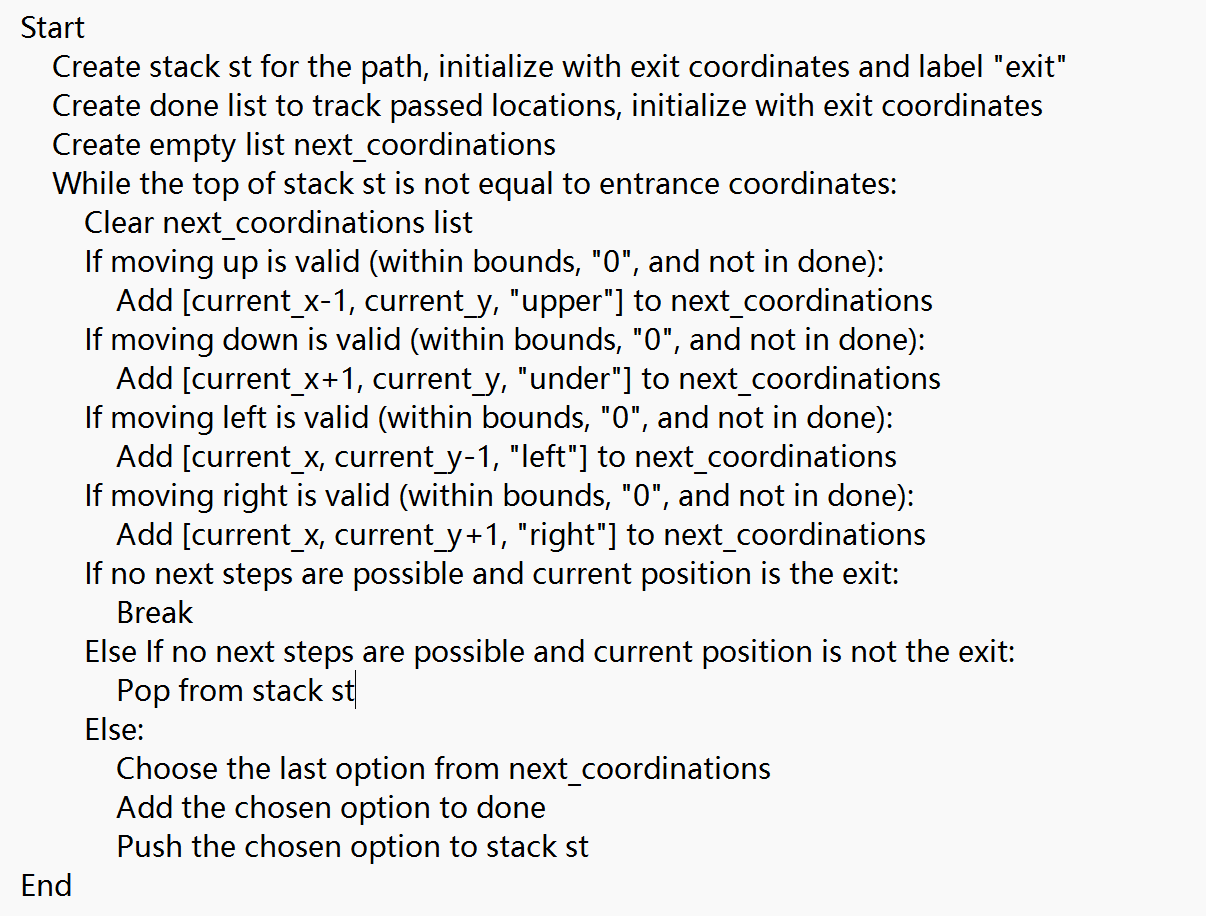
1. Maze Solving Design

A stack `st` is used to store the coordinates of the correct path, and a list `done` keeps track of coordinates that have already been visited to prevent revisiting. The top element of the stack `st` represents the current coordinate.

To find the path from the exit to the entrance, first, push the exit coordinate `exit` onto the stack `st` and add it to the list `done`. From the current coordinate, check the four possible moves: up, down, left, and right. Ensure that you do not move back, that the new coordinates are not in the `done` list, and that they are within the bounds of the maze by comparing them to the maze's length and width. The coordinates that meet these criteria are potential next steps. To facilitate direction output, add these directions to a list representing coordinates. Since there may be multiple valid directions at a junction, store these in a list `next\_coordinations`.

If `next\_coordinations` is not empty, take one of these coordinates as the next step, push it onto the stack `st`, and add it to the list `done` as the new top element for further analysis. If `next\_coordinations` is empty, it means the current coordinate has no valid paths, and thus it is not part of the correct path. Remove this coordinate from the top of the stack and return to the previous top element to reanalyze.

This process involves repeatedly backtracking until you either find the entrance or exhaust all possibilities. If, after returning to the exit coordinate through backtracking, the top of the stack is still the exit coordinate, it indicates that there is no valid path. Since the process of finding the next coordinate is the same for each step, use a `while` loop to handle this. The loop continues until the entrance is found (when the top of the stack is the entrance coordinate) or it is determined that there is no valid path, at which point the loop exits.

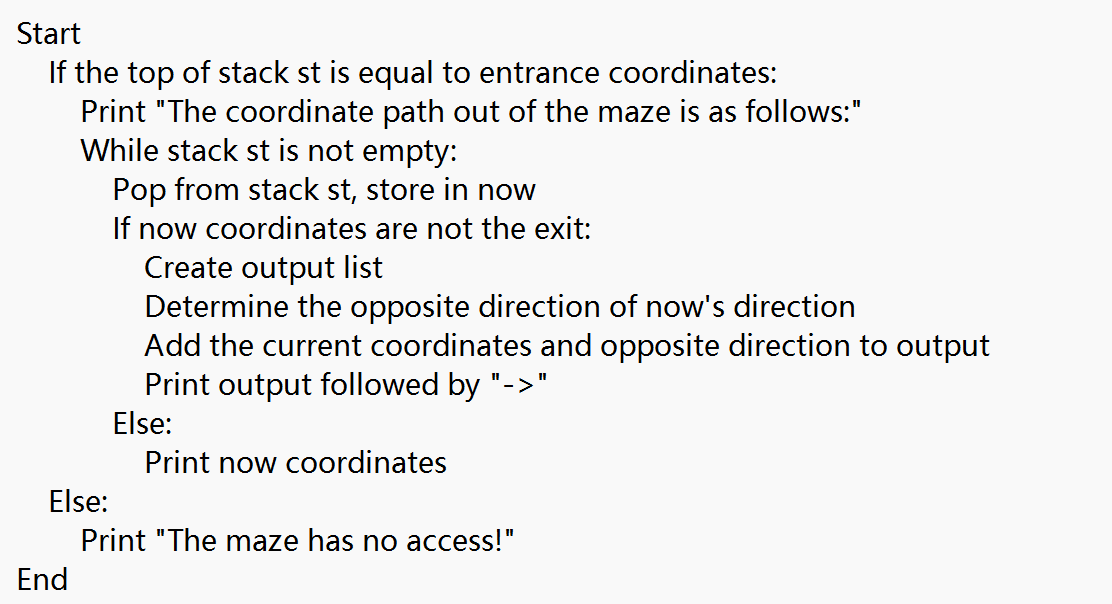
The pseudo code for this process is shown in Figure 1.7.

**Figure 1.7 Pseudo code for maze solving design**

1. Output Design

Ultimately, the output is determined by whether the top element of the stack `st` is the entrance coordinate. If the top element is the entrance coordinate, it means the maze has a solution. To output the correct path coordinates, print the elements of the stack `st` from top to bottom. However, to meet the output format of `[i, j, d]`, you need to use `pop` to remove and return the top element of the stack, taking the first two elements as the current coordinate `[i, j]`. Then, obtain the third element to determine the direction, and add the opposite direction to the current coordinate list as the direction `d` for the next coordinate.

The pseudo code for this process is shown in Figure 1.8.

**Figure 1.8 Output Design Pseudo Code**

#### 3.3.2 Recursive Program

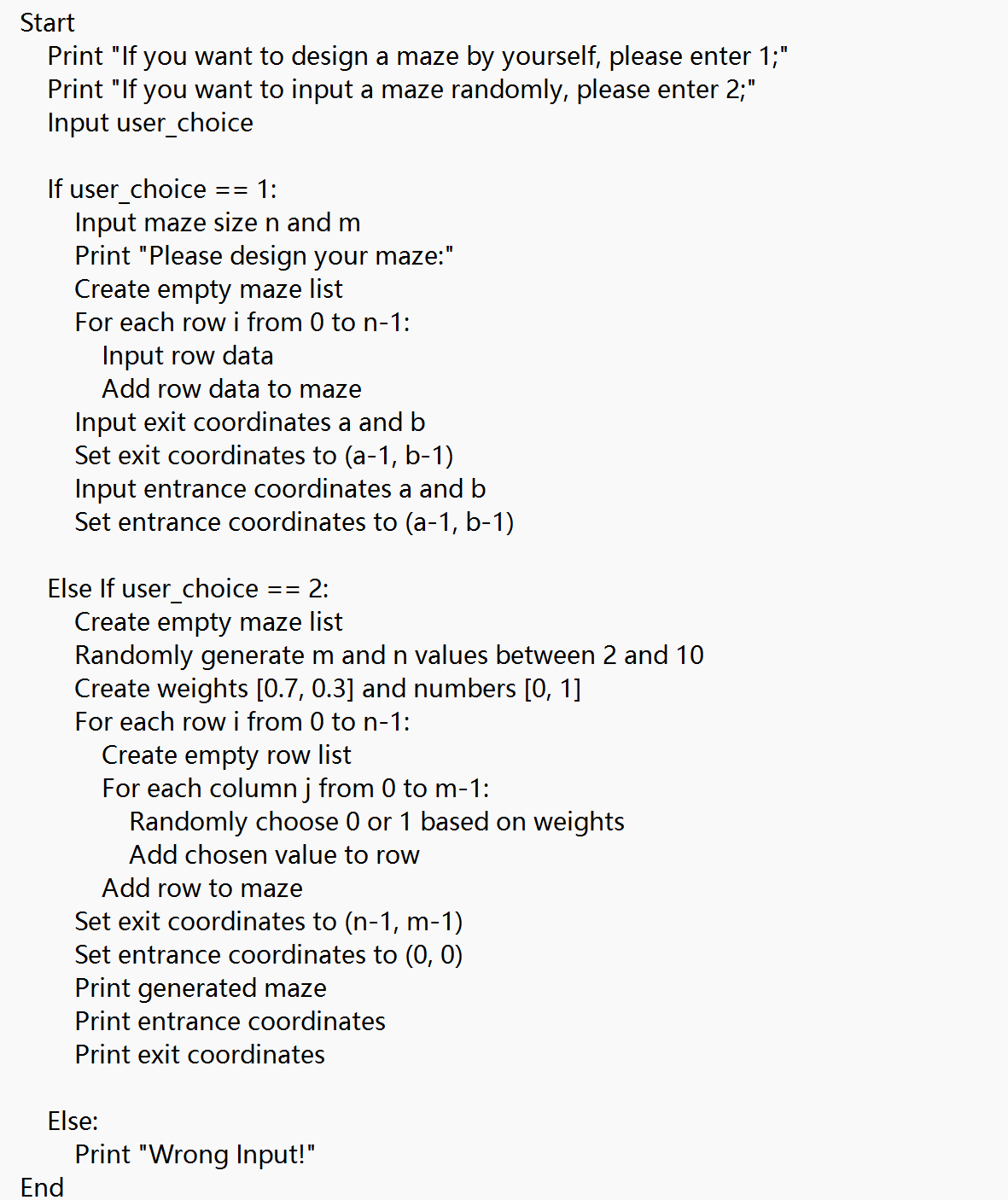
1. Input Design

At the start of the program, print a prompt asking the user to choose between manually entering a maze or generating one automatically. If the user inputs 1 for manual entry, print a prompt to ask for the maze size, read the user's input, and store the maze's length and width as `n` and `m`. Print a prompt to ask for the maze paths and walls, initialize a 2D array `maze` for maze storage, traverse the previously read rows, input and read each row's content, temporarily store it in the list `row`, and add it to the 2D array `maze`. Print a prompt to ask for the maze exit coordinates, read the user's input, and store the exit coordinates as `a` and `b`, combining them into the list `exit`. Print a prompt to ask for the maze entrance coordinates, read the user's input, and store the entrance coordinates as `a` and `b`, combining them into the list `entr`.

If the user inputs 2 for automatic generation, initialize a 2D array `maze` for maze storage, use the `random` library to generate random integers between 2 and 10 for the maze's length and width, stored as `n` and `m`. Use nested loops to traverse the rows, initializing the list `row` for each row. Within each row, traverse the columns and randomly generate numbers between 0 and 1 in a 7:3 ratio, adding them to the list `row`. After column traversal, add `row` to `maze`. The exit defaults to coordinates `[n-1, m-1]` and the entrance defaults to `[0, 0]`. Output the maze and the entry/exit points after assignment.

If the user inputs anything else, report an error.

The pseudo code for this process is shown in Figure 1.9.

**Figure 1.9 Pseudo code for maze input design**

1. Maze Solving Design

Use a list `done` to keep track of coordinates that have already been visited to avoid redundant searches. Use a boolean flag `Bingo` to determine if a coordinate is part of the solution path, which will be used to decide whether to output the path.

Design a function that takes the current coordinate `coordination`, the maze `Maze`, and the list `done` as parameters. The function's base case for recursion is when the current coordinate is the entrance coordinate. If this is true, the function should return `True`.

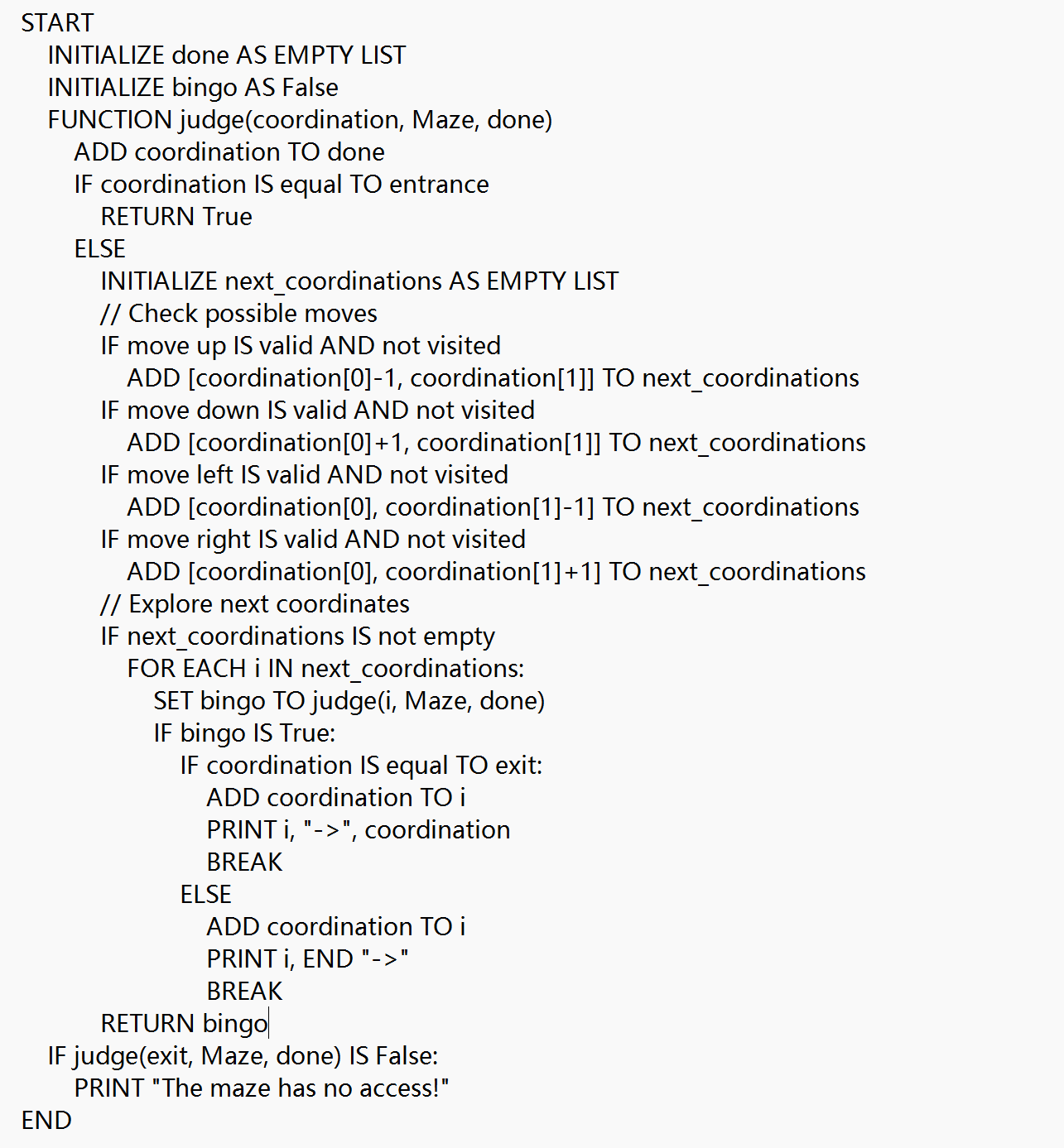
If the current coordinate is not the entrance, find all possible next coordinates and store them in a temporary list `next\_coordinations`. Use a `for` loop to iterate through each element in `next\_coordinations`. For each coordinate, recursively call the function with the new coordinate.Assign the returned value to `Bingo`.The function ultimately returns `Bingo`.

Each element in the `next\_coordinations` list represents a potential path. Only paths that eventually lead to the entrance return `True`. This `True` value propagates through the recursive calls, ensuring that all correct paths are marked as `True`.

When `Bingo` is `True`, it indicates that the coordinate is part of the correct path. To meet the problem requirements, in each recursive function call, add the current `coordination` (which is closer to the exit) to the list `i` (which is closer to the entrance) before outputting.

If at least one correct path exists, the function will return `True`. If the function ultimately returns `False` when called from outside, it signifies that no valid path exists in the maze.

The pseudo code for this process is shown in Figure 1.10.

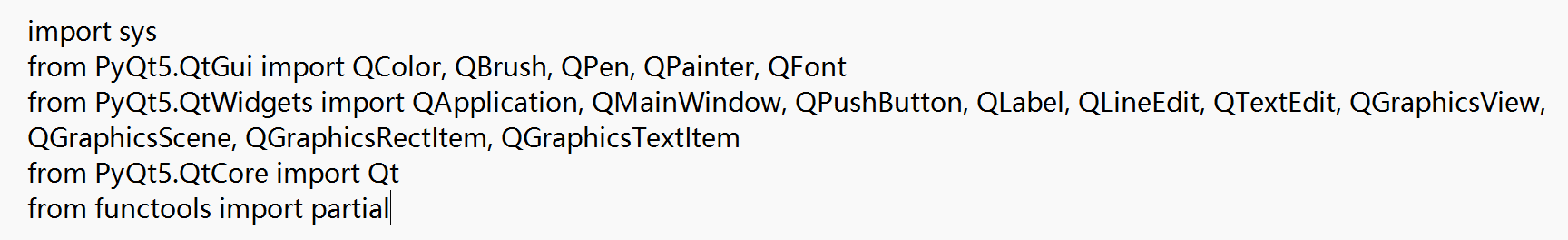
**Figure 1.10 pseudo code for maze solving and output**

#### 3.3.3 External Program

1. Library Usage

This program utilizes the PyQt5 library to visualize the input and output of the maze-solving process. The `PyQt5.QtGui` module provides various tools for graphical operations: `QFont` is used for setting and managing text fonts; `QColor` represents colors for drawing shapes and setting widget colors; `QBrush` handles the filling of shapes with color and texture; `QPen` defines the pen style for drawing, including color, line width, and line style; and `QPainter` is used for rendering shapes and text on widgets or graphic items. In the `PyQt5.QtWidgets` module, `QApplication` is the main class managing the application's control flow and settings; `QMainWindow` provides the main window framework; `QPushButton` is a button widget that triggers click events; `QLineEdit` serves as a single-line text editor for user input; `QLabel` displays text or images; `QTextEdit` is a multi-line text editor for displaying and editing multiple lines of text; and `QGraphicsTextItem` displays text within a `QGraphicsScene`.

The code demonstrating these functionalities is illustrated in Figure 1.11.

**Figure 1.11 Extension Library Call Code**

1. Stack Class Design

stack uses a list (`data`) to store its elements and an integer (`top`) to represent the index of the top element, which is initialized to -1.

The `is\_empty()` function determines whether the stack is empty by checking if `top` is less than 0. If `top` is less than 0, it indicates that the stack is empty, and the function returns `True`. Otherwise, it returns `False`.

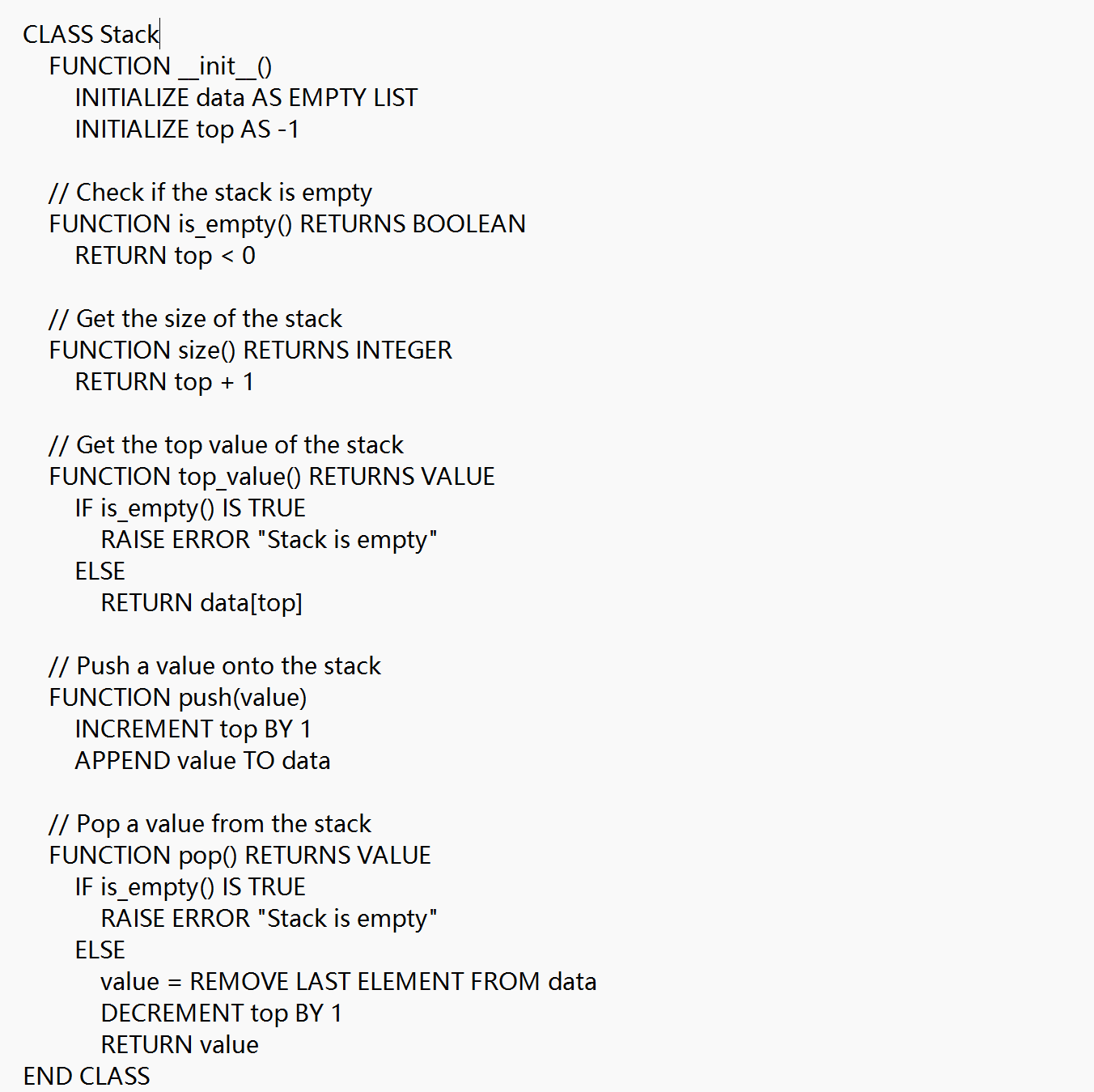
The `size()` function returns the number of elements in the stack. Since `top` represents the index of the last element, adding 1 to `top` gives the count of elements, or it can simply return the length of the `data` list.

The `top\_value()` function returns the current top element, which is the last element in the `data` list. It throws an exception if the stack is empty.

The `push(value)` function adds a new value to the top of the stack. It appends the value to the end of the `data` list and increments the `top` index by 1.

The `pop()` function removes and returns the top element of the stack. It deletes the last element from the `data` list and decrements the `top` index by 1. If the stack is empty, it throws an exception.

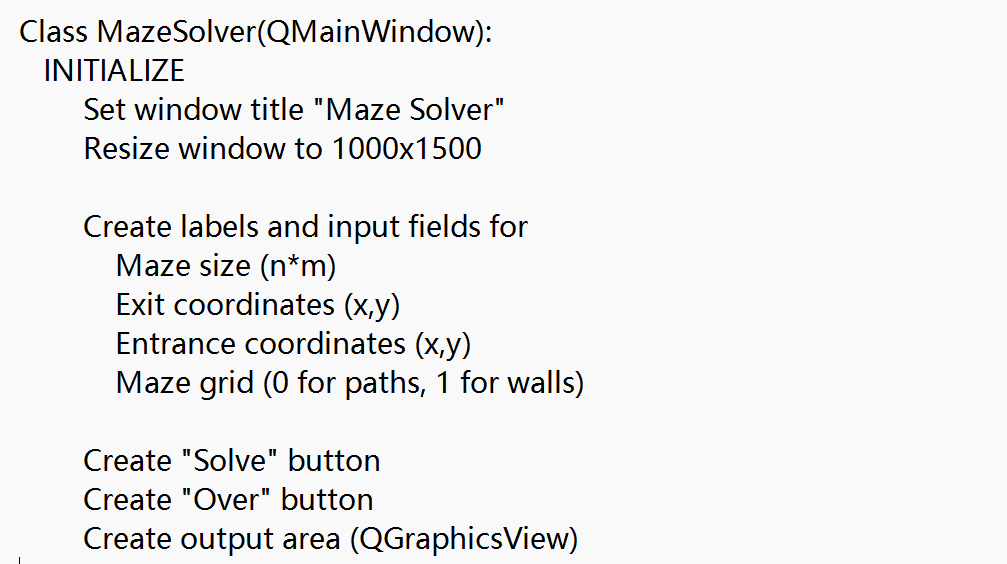
The pseudo code for these operations is illustrated in Figure 1.12.

**Figure 1.12 Pseudo code for Stack Class Design**

1. MazeSolver Class Design

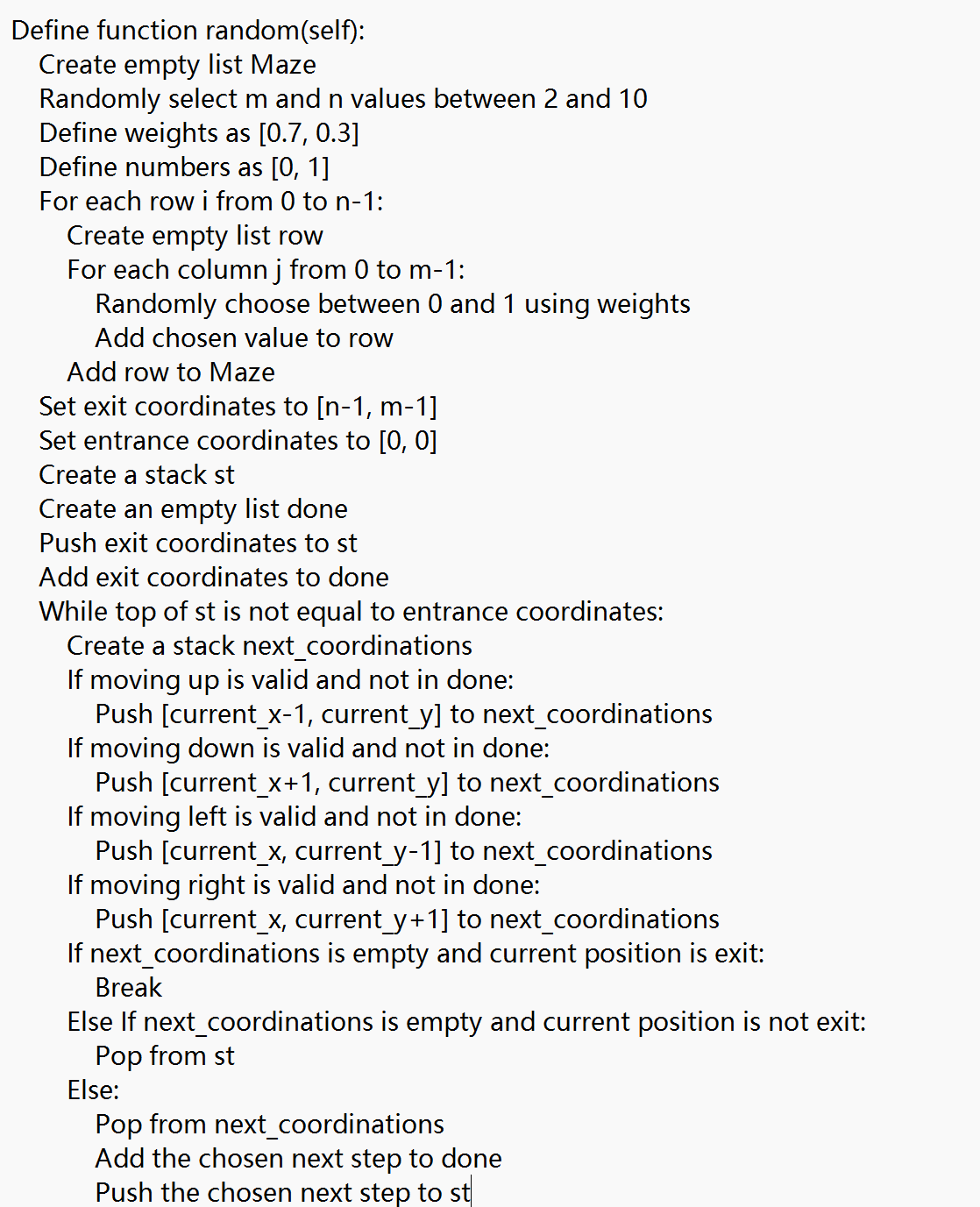
Initialize the class to display the initial interface by setting the window title to "Maze Solver" and the window size to 1000mm by 1500mm. Set up and arrange various controls on the interface: labels (QLabel) provide explanatory text, line edit boxes (QLineEdit) and text edit boxes (QTextEdit) allow users to input maze size, entrance, exit, and maze matrix, buttons (QPushButton) are used to trigger the `solve`, `over`, and `random` functions, and graphics view (QGraphicsView) and scene (QGraphicsScene) are used for drawing the maze and path.

The pseudo code for the initial interface is illustrated in Figure 1.13.



**Figure 1.13 Pseudo code for interface class initialization design**

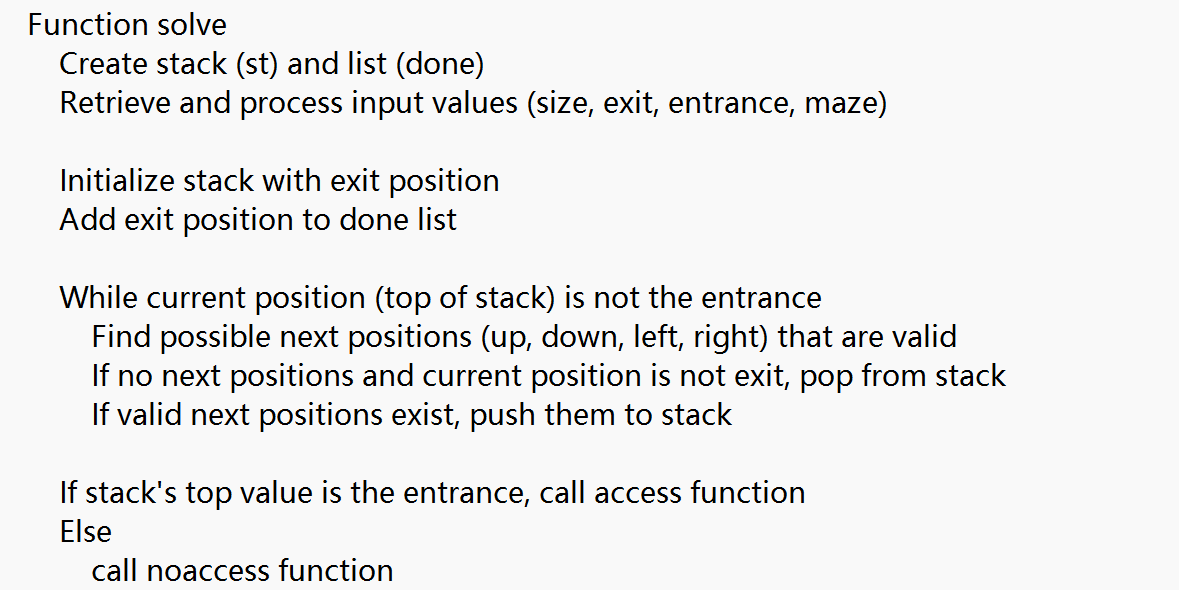
In the `Random` function, first randomly generate the maze size and maze content. Use a non-recursive algorithm to determine if the maze has a solution. If a path is found, call the `access()` method to draw the path; otherwise, call the `noaccess()` method to indicate that there is no solution, passing the parameter `Hello` to signal whether a solution was found.

The pseudo code for the `random` function in the `Mazesolver` class is illustrated in Figure 1.14.

**Figure 1.14 Pseudo code for Random Maze Generation and Solving in the Interface Class**

After the user inputs data into the text boxes and clicks the Solve button to solve the maze, the `Solve` function proceeds as follows: First, use the `text()` function to obtain the user's input. Then, convert the input into the appropriate format. Apply a non-recursive algorithm to determine whether the maze has a solution. If a path is found, call the `access()` method to draw the path; otherwise, call the `noaccess()` method to indicate no solution, passing the parameter `Hello` to signal whether a solution was found.

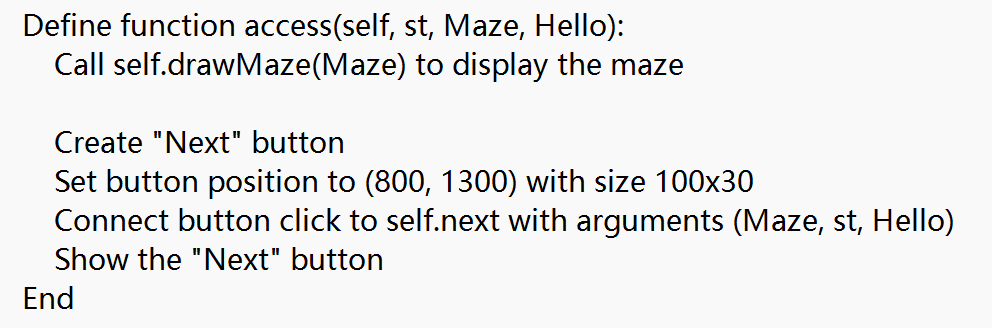
The pseudo code for the maze-solving interface class is illustrated in Figure 1.15.



**Figure 1.15 Interface class maze solving design pseudo code**

In the `access` function, first draw the maze using the `drawMaze()` function. Calculate the appropriate block size based on the canvas size and maze dimensions, painting blocks representing walls in black and the rest in white. Set up a button (QPushButton) to trigger the `next()` function. Each time the button is clicked, if the stack `st` representing the correct path is not empty, remove the top element and use the `drawPath()` function to paint the corresponding block red. If the stack is empty, display "Out of Maze!" at the end of the canvas to indicate that the maze has been successfully solved.

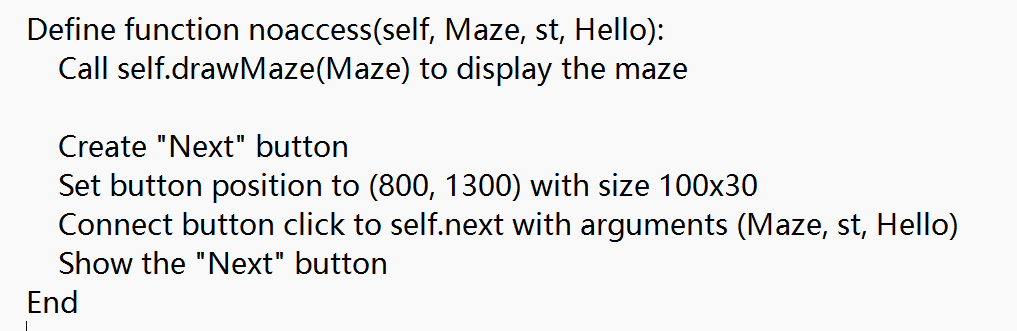
The pseudo code for the path visualization output design in the interface class is shown in Figure 1.16.



**Figure 1.16 Interface Class Path Visualization Output Design Pseudo code**

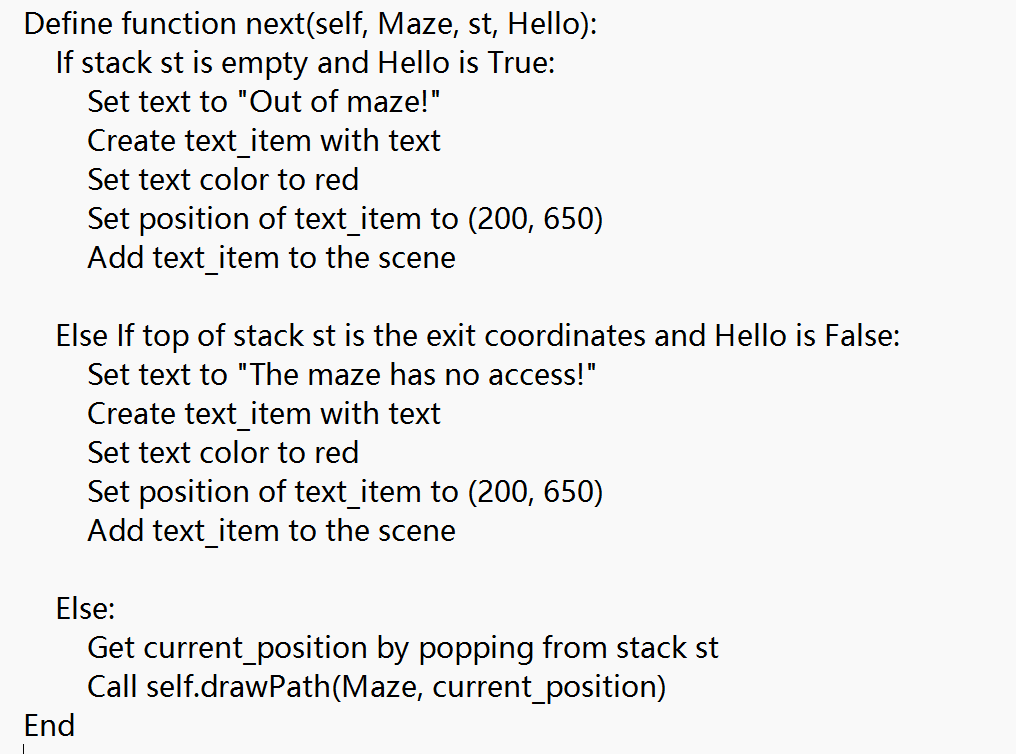
In the `noaccess` function, first use the `drawMaze()` function to draw the maze. Calculate the appropriate block size based on the canvas size and maze dimensions, painting blocks representing walls in black and the rest in white. Set up a button (QPushButton) to trigger the `next()` function. Display "The maze has no access!" at the end of the canvas.

The pseudo code for the no-path output design in the interface class is shown in Figure 1.17.

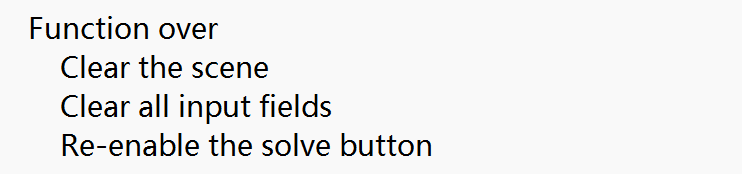
**Figure 1.17 Pseudo code for Interface Class No Path Output Design**

In the `next` function, determine whether there is a solution based on the `Hello` parameter. When `Hello` is `False`, it indicates that there is no solution, so directly print "The maze has no access!" at the end of the canvas. When `Hello` is `True`, it indicates that there is a solution. If the current coordinate has not yet reached the exit coordinate, paint the next path block in red. If the coordinate is the exit, print "Out of Maze!" at the end of the canvas to indicate that the maze has been successfully solved.

The pseudocode for triggering movement in the interface class is shown in Figure 1.18.

**Figure 1.18 Pseudo code for Triggering Movement in the Interface Class**

In the `over` function, clear the current canvas content and reset the input fields. Re-enable the solve button to allow the user to design and solve a new maze.

The pseudocode for clearing the screen in the interface class is shown in Figure 1.19.

**Figure 1.19 Pseudo code for interface class screen clearing design**

## Debugging Analysis

1. Difficulties and Solutions in the Process:

In the initial testing phase of the non-recursive maze-solving program, the output consistently included the coordinates of erroneous paths attempted at junctions. Upon debugging, it was discovered that during each iteration of the loop, the `next\_coordinations` list retained previously untraveled valid coordinates. Thus, when reaching a junction and attempting to follow an incorrect path, the `next\_coordinations` list still contained coordinates even though there were no viable paths left. This prevented the algorithm from removing the current coordinate from the st stack, causing it to continue searching with invalid coordinates. The issue was resolved by reinitializing the `next\_coordinations` stack within each loop iteration to ensure it only contained the coordinates for the current step. This approach ensured that invalid coordinates were correctly removed from the stack.

1. Algorithm Time and Space Complexity Analysis

The space complexity of the non-recursive algorithm is O(n\*m), primarily due to the storage of the maze, the stack, and the list of visited cells. The time complexity is O(n\*m) because the operations involved in finding the next coordinates within each iteration are constant-time operations. Therefore, in the worst case, where all coordinates are traversed, the algorithm’s time complexity is O(n\*m). The recursive algorithm also has a space complexity of O(n\*m), which accounts for storing all visited cells and the recursion stack. The time complexity is O(n\*m) as well, though it could be higher in practice because recursive calls might revisit the same cell multiple times.

1. Algorithm Discussion

Consider using a tree data structure to represent the maze, where all traversable coordinates are stored as nodes. The goal is to find the unique path from the root to a specified leaf node.

## User Manual

1. Non-recursive and Recursive Programs

Follow the terminal prompts to input data after the colon. Use "\*" to separate the maze's length and width, and use spaces to separate the maze matrix content. Use "," to connect the horizontal and vertical coordinates for the entrance and exit. After completing the input, press Enter to confirm and wait for the output.

1. Extended Program

Input data according to the window prompts in the specified text boxes. Use "\*" to connect the maze's length and width, and spaces to separate the maze matrix content. Use "," to connect the horizontal and vertical coordinates for the entrance and exit. After all information is entered, click the "Solve" button. If a solution exists, the maze will be displayed on the canvas below. Click the "Next" button in the bottom right corner to reveal the solution step by step. Finally, press the "Over" button at the bottom to clear the interface and prepare for the next maze input.

## Test Data And Test Results

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test data | Purpose | Correct Output | Non-recursive program output | Recursive program output | External program output | Current state |
| 0 0 1  1 0 1  1 0 0 | Check if the program can run normally | [[0, 0, 'right']->[0, 1, 'under']->[1, 1, 'under']->[2, 1, 'right']->[2, 2] | Figure  1.18.1 | Figure  1.18.2 | Figure  1.18.3 | Pass |
| 0 1 1 0 1  0 0 1 0 1  1 0 0 0 0 | Check if the program can handle interference from crossroads | [0, 0, 'under']->[1, 0, 'right']->[1, 1, 'under']->[2, 1, 'right']->[2, 2, 'right']->[2, 3, 'right']->[2, 4] | Figure  1.18.4 | Figure  1.18.5 | Figure  1.18.6 | Pass |
| 0 0 1 1 1  1 0 1 1 1  1 1 1 1 1  1 1 1 0 0 | Check if the program can handle situations where the correct path does not exist | The maze has no access! | Figure  1.18.7 | Figure  1.18.8 | Figure  1.18.9 | Pass |
| 0 0 1 1 1 0 1  1 0 0 0 0 0 0  0 1 0 1 0 1 1  1 1 1 0 0 0 0 | Check if the program can handle situations with complex interference | [0, 0, 'right']->[0, 1, 'under']->[1, 1, 'right']->[1, 2, 'right']->[1, 3, 'right']->[1, 4, 'under']->[2, 4, 'under']->[3, 4, 'right']->[3, 5, 'right']->[3, 6] | Figure  1.18.10 | Figure  1.18.11 | Figure  1.18.12 | Pass |
| 1 1 0 1 1 1 0 1 0  1 1 0 0 0 0 0 0 0  1 0 0 1 0 1 1 0 1  0 1 1 0 0 0 0 1 1  0 0 0 0 1 1 1 1 0  1 0 1 1 1 0 1 1 0 | Check if the program can handle maze situations with larger amounts of data | [0, 8, 'under']->[1, 8, 'left']->[1, 7, 'left']->[1, 6, 'left']->[1, 5, 'left']->[1, 4, 'under']->[2, 4, 'under']->[3, 4, 'left']->[3, 3, 'under']->[4, 3, 'left']->[4, 2, 'left']->[4, 1, 'left']->[4, 0] | Figure  1.18.13 | Figure  1.18.14 | Figure  1.18.15 | Pass |

**Figure 1.18 Test Case Table**



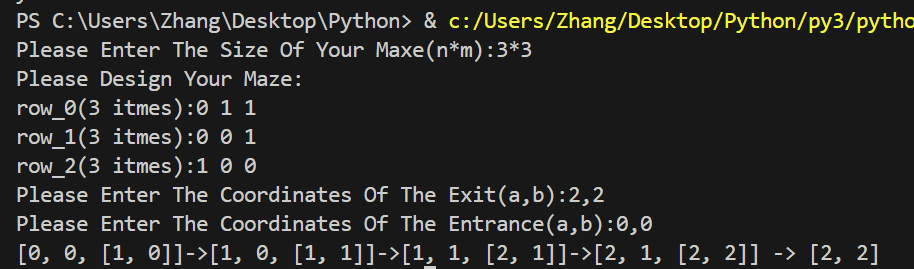
**Figure 1.18.1 Case 1 Non recursive Program** 

Figure 1.18.2 Case 1 Recursive Program Output

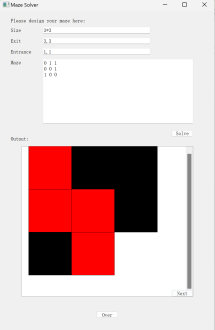
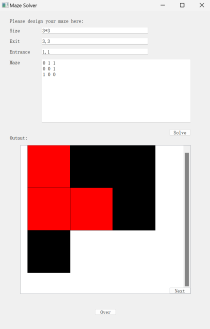
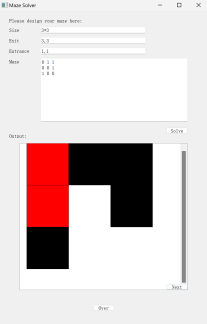
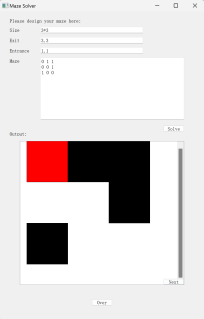
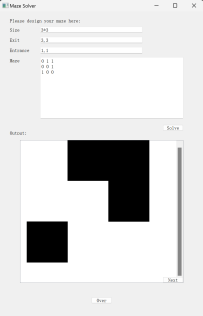


Figure 1.18.3 Case 1 Extended Program Output

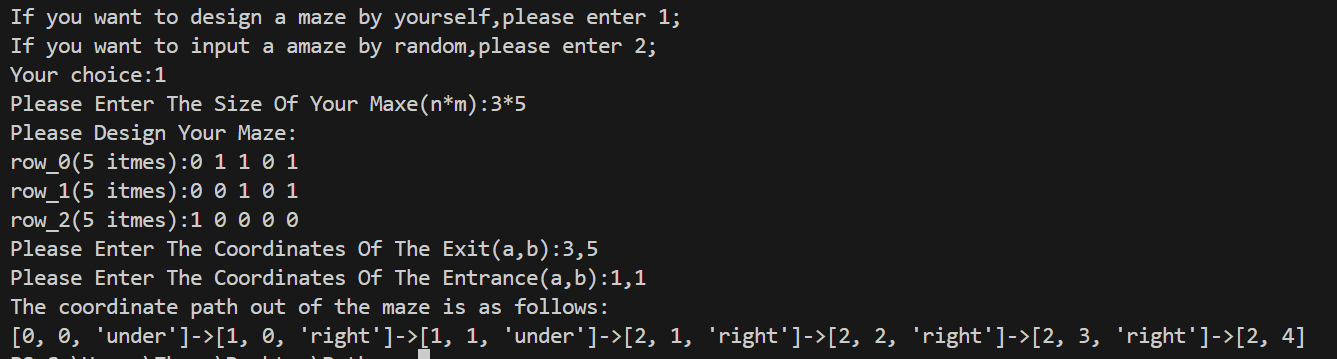


Figure 1.18.4 Case 2 Non recursive Program Output

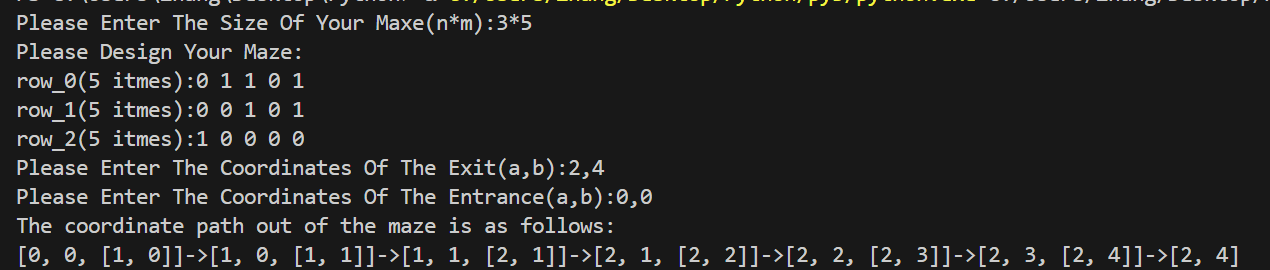


Figure 1.18.5 Case 2 Recursive Program Output

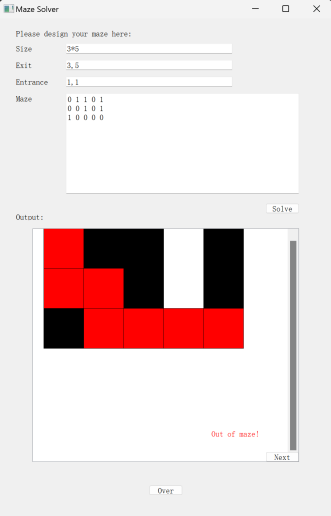


Figure 1.18.6 Case 2 Extended Program Output

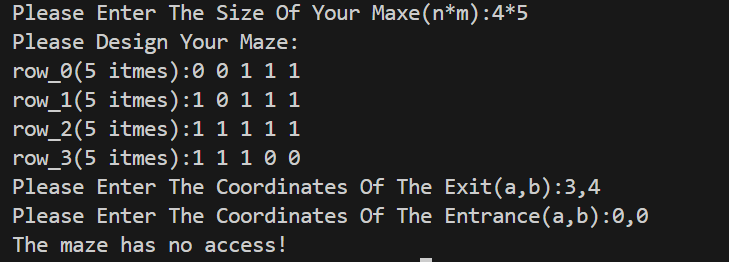


Figure 1.18.7 Case 3 Non recursive Program Output

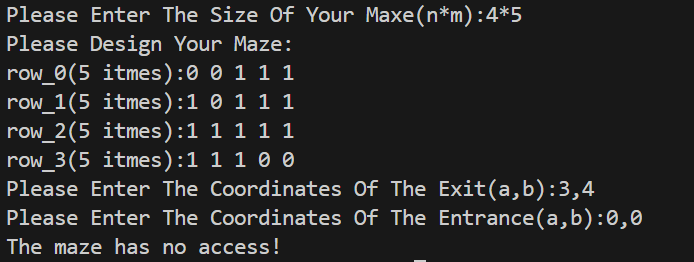


Figure 1.18.8 Case 3 Recursive Program Output



Figure 1.18.9 Case 3 Extended Program Output

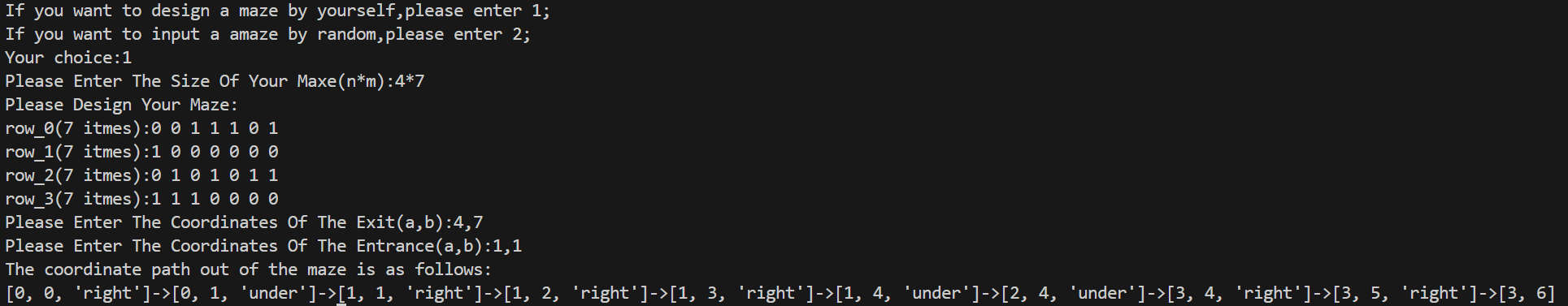


Figure 1.18.10 Case 4 Non recursive Program Output

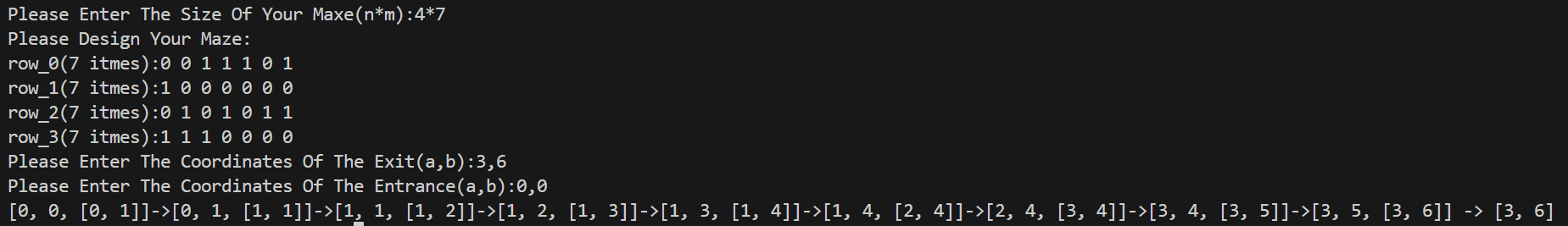


Figure 1.18.11 Case 4 Recursive Program Output

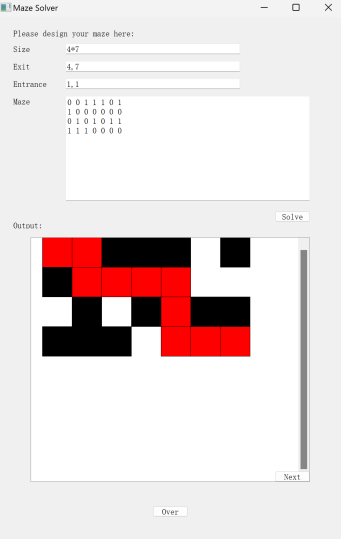


Figure 1.18.12 Case 4 Extended Program Output

Figure 1.18.13 Case 5 Non recursive Program Output

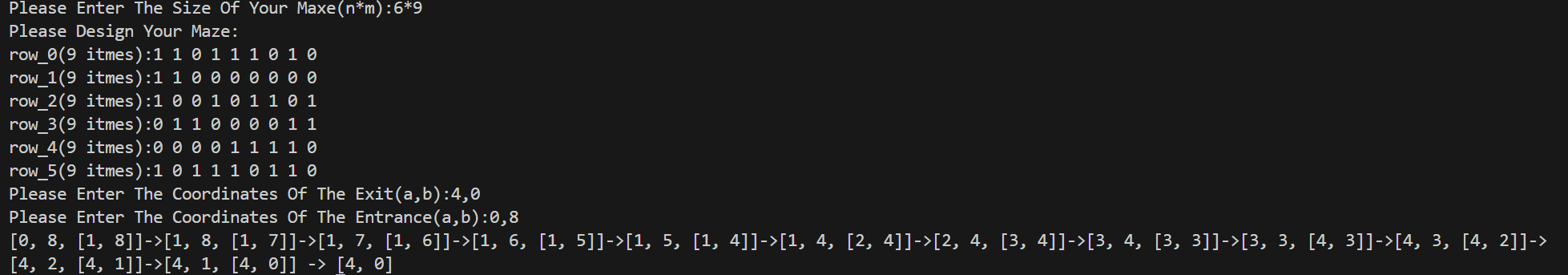


Figure 1.18.14 Case 5 Recursive Program Output

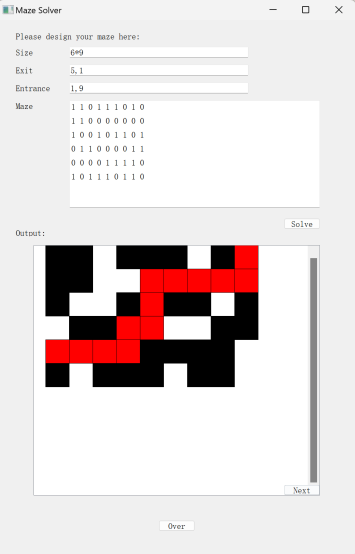


Figure 1.18.15 Case 5 Extended Program Output

## Source Program Inventory

1. Non-recursive Program

import random

#design the database-stack

class Stack():

    def \_\_init\_\_(self):

        self.data=[]

        self.top=-1

    def is\_empty(self):

        return self.top<0

    def size(self):

        return self.top+1

    def top\_value(self):

        if self.is\_empty():

            raise KeyError("Stack is empty")

        else:

            return self.data[self.top]

    def push(self,value):

        self.top+=1

        self.data.append(value)

    def pop(self):

        if self.is\_empty():

            raise KeyError("Stack is empty")

        else:

            value=self.data.pop()

            self.top-=1

            return value

#design a maze

print("If you want to design a maze by yourself,please enter 1;")

print("If you want to input a amaze by random,please enter 2;")

choice=int(input("Your choice:"))

if choice==1:

    n,m=map(int,input("Please Enter The Size Of Your Maxe(n\*m):").split("\*"))#n rows with each row m items

    print("Please Design Your Maze:")

    Maze=[]

    for i in range(n):

        row=input(f"row\_{i}({m} itmes):").split()

        Maze.append(row)

    a,b=map(int,input("Please Enter The Coordinates Of The Exit(a,b):").split(","))

    exit=[a-1,b-1]

    a,b=map(int,input("Please Enter The Coordinates Of The Entrance(a,b):").split(","))

    entr=[a-1,b-1]

elif choice==2:

    Maze=[]

    m=random.randint(2,10)

    n=random.randint(2,10)

    weights=[0.7,0.3]

    numbers=[0,1]

    for i in range(n):

        row=[]

        for j in range(m):

            row.append(str(random.choices(numbers,weights,k=1)[0]))

        Maze.append(row)

    exit=[n-1,m-1]

    entr=[0,0]

    print("The maze is as follows:")

    for i in range(n):

        for j in range(m):

            print(Maze[i][j],end="")

        print("\n",end="")

    print("The coordinates of the entrance is:",entr)

    print("The coordinates of the exit is:",exit)

else:

    print("Wrong Input!")

#find the way

st=Stack()#the right way

exit.append("exit")

st.push(exit)

done=[]#the place already passed

done.append([exit[0],exit[1]])

next\_coordinations=Stack()

while [st.top\_value()[0],st.top\_value()[1]]!=entr:

    next\_coordinations=[]

    #find the possibiltis of the next coordination first:1.still valid 2.is "0" 3.have not passed

    if st.top\_value()[0]-1>=0 and Maze[st.top\_value()[0]-1][st.top\_value()[1]]=="0" and [st.top\_value()[0]-1,st.top\_value()[1]] not in done:

        next\_coordinations.append([st.top\_value()[0]-1,st.top\_value()[1],"upper"])#upper

    if st.top\_value()[0]+1<n and Maze[st.top\_value()[0]+1][st.top\_value()[1]]=="0" and [st.top\_value()[0]+1,st.top\_value()[1]] not in done:

        next\_coordinations.append([st.top\_value()[0]+1,st.top\_value()[1],"under"])#under

    if st.top\_value()[1]-1>=0 and Maze[st.top\_value()[0]][st.top\_value()[1]-1]=="0" and [st.top\_value()[0],st.top\_value()[1]-1] not in done:

        next\_coordinations.append([st.top\_value()[0],st.top\_value()[1]-1,"left"])#left

    if st.top\_value()[1]+1<m and Maze[st.top\_value()[0]][st.top\_value()[1]+1]=="0" and [st.top\_value()[0],st.top\_value()[1]+1] not in done:

        next\_coordinations.append([st.top\_value()[0],st.top\_value()[1]+1,"right"])#right

    #consider a fork:if one comes to an end,we need to back and find another,until we back to the start,taht means there is really no way

    if len(next\_coordinations)==0 and [st.top\_value()[0],st.top\_value()[1]]==[exit[0],exit[1]]:

        break

    elif len(next\_coordinations)==0 and [st.top\_value()[0],st.top\_value()[1]]!=[exit[0],exit[1]]:

        k=st.pop()

    else:

        next=next\_coordinations.pop(-1)

        done.append([next[0],next[1]])

        st.push(next)

#output

if [st.top\_value()[0],st.top\_value()[1]]==entr:

    print("The coordinate path out of the maze is as follows:")

    for i in range(st.size()):

        now=st.pop()

        if [now[0],now[1]]!=[exit[0],exit[1]]:

            output=[]

            way=now[2]

            if way=="upper":

                output=[now[0],now[1],"under"]

            if way=="under":

                output=[now[0],now[1],"upper"]

            if way=="left":

                output=[now[0],now[1],"right"]

            if way=="right":

                output=[now[0],now[1],"left"]

            print(output,end="->")

        else:

            print([now[0],now[1]])

else:

    print("The maze has no access!")

（2）递归程序

import random

#design a maze

print("If you want to design a maze by yourself,please enter 1;")

print("If you want to input a amaze by random,please enter 2;")

choice=int(input("Your choice:"))

if choice==1:

    n,m=map(int,input("Please Enter The Size Of Your Maxe(n\*m):").split("\*"))#n rows with each row m items

    print("Please Design Your Maze:")

    Maze=[]

    for i in range(n):

        row=input(f"row\_{i}({m} itmes):").split()

        Maze.append(row)

    a,b=map(int,input("Please Enter The Coordinates Of The Exit(a,b):").split(","))

    exit=[a-1,b-1]

    a,b=map(int,input("Please Enter The Coordinates Of The Entrance(a,b):").split(","))

    entr=[a-1,b-1]

elif choice==2:

    Maze=[]

    m=random.randint(2,10)

    n=random.randint(2,10)

    weights=[0.7,0.3]

    numbers=[0,1]

    for i in range(n):

        row=[]

        for j in range(m):

            row.append(str(random.choices(numbers,weights,k=1)[0]))

        Maze.append(row)

    exit=[n-1,m-1]

    entr=[0,0]

    print("The maze is as follows:")

    for i in range(n):

        for j in range(m):

            print(Maze[i][j],end="")

        print("\n",end="")

    print("The coordinates of the entrance is:",entr)

    print("The coordinates of the exit is:",exit)

else:

    print("Wrong Input!")

#find the way

done=[]

bingo=False

def judge(coordination,Maze,done) :

    global bingo

    done.append(coordination)

    if coordination==entr:

        return True

    else:

        next\_coordinations=[]

        if coordination[0]-1>=0 and Maze[coordination[0]-1][coordination[1]]=="0" and [coordination[0]-1,coordination[1]] not in done:

            next\_coordinations.append([coordination[0]-1,coordination[1]])#upper

        if coordination[0]+1<n and Maze[coordination[0]+1][coordination[1]]=="0" and [coordination[0]+1,coordination[1]] not in done:

            next\_coordinations.append([coordination[0]+1,coordination[1]])#under

        if coordination[1]-1>=0 and Maze[coordination[0]][coordination[1]-1]=="0" and [coordination[0],coordination[1]-1] not in done:

            next\_coordinations.append([coordination[0],coordination[1]-1])#left

        if coordination[1]+1<m and Maze[coordination[0]][coordination[1]+1]=="0" and [coordination[0],coordination[1]+1] not in done:

            next\_coordinations.append([coordination[0],coordination[1]+1])#right

        if len(next\_coordinations)!=0:

            for i in next\_coordinations:

                bingo=judge(i,Maze,done)

                if bingo:

                    if coordination==exit:

                        i.append(coordination)

                        print(i,"->",coordination)

                        break

                    else:

                        i.append(coordination)

                        print(i,end="->")

                        break

        return bingo

if judge(exit,Maze,done)==False:

    print("The maze has no access!")

（3）拓展程序

import sys

from PyQt5.QtGui import QColor, QBrush, QPen, QPainter, QFont

from PyQt5.QtWidgets import QApplication, QMainWindow, QPushButton, QLabel, QLineEdit, QTextEdit, QGraphicsView, QGraphicsScene, QGraphicsRectItem, QGraphicsTextItem

from PyQt5.QtCore import Qt

from functools import partial

import random

class Stack():

    def \_\_init\_\_(self):

        self.data = []

        self.top = -1

    def is\_empty(self):

        return self.top < 0

    def size(self):

        return self.top + 1

    def top\_value(self):

        if self.is\_empty():

            raise KeyError("Stack is empty")

        else:

            return self.data[self.top]

    def push(self, value):

        self.top += 1

        self.data.append(value)

    def pop(self):

        if self.is\_empty():

            raise KeyError("Stack is empty")

        else:

            value = self.data.pop()

            self.top -= 1

            return value

class MazeSolver(QMainWindow):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.setWindowTitle("Maze Solver")

        self.resize(1000, 1500)

        # Create labels and line edits

        self.label\_0 = QLabel("Please design your maze here:", self)

        self.label\_0.setGeometry(50, 30, 500, 30)

        self.label\_1 = QLabel("Size", self)

        self.label\_1.setGeometry(50, 80, 200, 20)

        self.edit\_1 = QLineEdit(self)

        self.edit\_1.setPlaceholderText("n\*m")

        self.edit\_1.setGeometry(200, 75, 500, 30)

        self.label\_2 = QLabel("Exit", self)

        self.label\_2.setGeometry(50, 130, 200, 20)

        self.edit\_2 = QLineEdit(self)

        self.edit\_2.setPlaceholderText("x,y")

        self.edit\_2.setGeometry(200, 125, 500, 30)

        self.label\_3 = QLabel("Entrance", self)

        self.label\_3.setGeometry(50, 180, 200, 20)

        self.edit\_3 = QLineEdit(self)

        self.edit\_3.setPlaceholderText("x,y")

        self.edit\_3.setGeometry(200, 175, 500, 30)

        self.label\_4 = QLabel("Maze", self)

        self.label\_4.setGeometry(50, 230, 500, 20)

        self.edit\_4 = QTextEdit(self)

        self.edit\_4.setPlaceholderText("0 for paths and 1 for walls")

        self.edit\_4.setGeometry(200, 225, 700, 300)

        self.edit\_4.setFixedWidth(700)

        self.edit\_4.setFixedHeight(300)

        # Create a button

        self.solve\_button = QPushButton("Solve", self)

        self.solve\_button.setGeometry(800, 555, 100, 30)

        self.solve\_button.clicked.connect(self.solve)

        # Create "Over" button

        self.over\_button = QPushButton("Over", self)

        self.over\_button.setGeometry(450, 1400, 100, 30)

        self.over\_button.clicked.connect(self.over)

        # Create "Random" button

        self.random\_button = QPushButton("Random", self)

        self.random\_button.setGeometry(600, 30, 100, 30)

        self.random\_button.clicked.connect(self.random)

        # Create the drawing area

        self.label\_5 = QLabel("Output:", self)

        self.label\_5.setGeometry(50, 585, 200, 20)

        self.scene = QGraphicsScene(self)

        self.scene.setSceneRect(0, 0, 700, 700)

        self.view = QGraphicsView(self.scene, self)

        self.view.setGeometry(100, 630, 800, 700)

    def random(self):

        Maze=[]

        m=random.randint(2,10)

        n=random.randint(2,10)

        weights=[0.7,0.3]

        numbers=[0,1]

        for i in range(n):

            row=[]

            for j in range(m):

                row.append(str(random.choices(numbers,weights,k=1)[0]))

            Maze.append(row)

        exit=[n-1,m-1]

        entr=[0,0]

        st = Stack()

        done = []

        st.push(exit)

        done.append(exit)

        while st.top\_value() != entr:

            next\_coordinations = Stack()

            if st.top\_value()[0] - 1 >= 0 and Maze[st.top\_value()[0] - 1][st.top\_value()[1]] == '0' and [st.top\_value()[0] - 1, st.top\_value()[1]] not in done:

                next\_coordinations.push([st.top\_value()[0] - 1, st.top\_value()[1]])  # upper

            if st.top\_value()[0] + 1 < n and Maze[st.top\_value()[0] + 1][st.top\_value()[1]] == '0' and [st.top\_value()[0] + 1, st.top\_value()[1]] not in done:

                next\_coordinations.push([st.top\_value()[0] + 1, st.top\_value()[1]])  # under

            if st.top\_value()[1] - 1 >= 0 and Maze[st.top\_value()[0]][st.top\_value()[1] - 1] == '0' and [st.top\_value()[0], st.top\_value()[1] - 1] not in done:

                next\_coordinations.push([st.top\_value()[0], st.top\_value()[1] - 1])  # left

            if st.top\_value()[1] + 1 < m and Maze[st.top\_value()[0]][st.top\_value()[1] + 1] == '0' and [st.top\_value()[0], st.top\_value()[1] + 1] not in done:

                next\_coordinations.push([st.top\_value()[0], st.top\_value()[1] + 1])  # right

            if next\_coordinations.is\_empty() and st.top\_value() == exit:

                break

            elif next\_coordinations.is\_empty() and st.top\_value() != exit:

                st.pop()

            else:

                next = next\_coordinations.pop()

                done.append(next)

                st.push(next)

        if st.top\_value() == entr:

            hello=True

            self.access(Maze,st,hello)

        else:

            hello=False

            self.noaccess(Maze,st,hello)

    def solve(self):

        st = Stack()

        done = []

        input\_1 = self.edit\_1.text()

        input\_2 = self.edit\_2.text()

        input\_3 = self.edit\_3.text()

        input\_4 = self.edit\_4.toPlainText().strip()

        n, m = map(int, input\_1.split("\*"))

        x, y = map(int, input\_2.split(","))

        exit = [x - 1, y - 1]

        x, y = map(int, input\_3.split(","))

        entr = [x - 1, y - 1]

        Maze = [i.split() for i in input\_4.split('\n')]

        st.push(exit)

        done.append(exit)

        while st.top\_value() != entr:

            next\_coordinations = Stack()

            if st.top\_value()[0] - 1 >= 0 and Maze[st.top\_value()[0] - 1][st.top\_value()[1]] == '0' and [st.top\_value()[0] - 1, st.top\_value()[1]] not in done:

                next\_coordinations.push([st.top\_value()[0] - 1, st.top\_value()[1]])  # upper

            if st.top\_value()[0] + 1 < n and Maze[st.top\_value()[0] + 1][st.top\_value()[1]] == '0' and [st.top\_value()[0] + 1, st.top\_value()[1]] not in done:

                next\_coordinations.push([st.top\_value()[0] + 1, st.top\_value()[1]])  # under

            if st.top\_value()[1] - 1 >= 0 and Maze[st.top\_value()[0]][st.top\_value()[1] - 1] == '0' and [st.top\_value()[0], st.top\_value()[1] - 1] not in done:

                next\_coordinations.push([st.top\_value()[0], st.top\_value()[1] - 1])  # left

            if st.top\_value()[1] + 1 < m and Maze[st.top\_value()[0]][st.top\_value()[1] + 1] == '0' and [st.top\_value()[0], st.top\_value()[1] + 1] not in done:

                next\_coordinations.push([st.top\_value()[0], st.top\_value()[1] + 1])  # right

            if next\_coordinations.is\_empty() and st.top\_value() == exit:

                break

            elif next\_coordinations.is\_empty() and st.top\_value() != exit:

                st.pop()

            else:

                next = next\_coordinations.pop()

                done.append(next)

                st.push(next)

        if st.top\_value() == entr:

            hello=True

            self.access(Maze,st,hello)

        else:

            hello=False

            self.noaccess(Maze,st,hello)

    def noaccess(self,Maze,st,Hello):

        self.drawMaze(Maze)

        self.next\_button = QPushButton("Next", self)

        self.next\_button.setGeometry(800, 1300, 100, 30)

        self.next\_button.clicked.connect(partial(self.next,Maze, st,Hello))

        self.next\_button.show()

    def access(self, st, Maze,Hello):

        self.drawMaze(Maze)

        self.next\_button = QPushButton("Next", self)

        self.next\_button.setGeometry(800, 1300, 100, 30)

        self.next\_button.clicked.connect(partial(self.next,Maze, st,Hello))

        self.next\_button.show()

    def next(self,Maze, st,Hello):

        if st.is\_empty() and Hello ==True:

            text = "Out of maze!"

            text\_item = QGraphicsTextItem(text)

            text\_item.setDefaultTextColor(QColor(255, 0, 0))

            text\_item.setPos(200,650.0)

            self.scene.addItem(text\_item)

        elif st.top\_value()==[len(Maze)-1,len(Maze[0])-1] and Hello==False:

            text = "The maze has no access!"

            text\_item = QGraphicsTextItem(text)

            text\_item.setDefaultTextColor(QColor(255, 0, 0))

            text\_item.setPos(200,650.0)

            self.scene.addItem(text\_item)

        else:

            current\_position = st.pop()

            self.drawPath(Maze,current\_position)

    def drawMaze(self, Maze):

        cell\_size = 600//max(len(Maze),len(Maze[0]))

        wall\_color = QColor(0, 0, 0)

        for i in range(len(Maze)):

            for j in range(len(Maze[i])):

                if Maze[i][j] == '1':

                    rect\_item = QGraphicsRectItem(j \* cell\_size, i \* cell\_size, cell\_size, cell\_size)

                    rect\_item.setBrush(wall\_color)

                    self.scene.addItem(rect\_item)

        self.scene.update()

    def drawPath(self, Maze,position):

        cell\_size = 600//max(len(Maze),len(Maze[0]))

        path\_color = QColor(255, 0, 0)

        rect\_item = QGraphicsRectItem(position[1] \* cell\_size, position[0] \* cell\_size, cell\_size, cell\_size)

        rect\_item.setBrush(path\_color)

        self.scene.addItem(rect\_item)

    def over(self):

        self.scene.clear()

        self.edit\_1.clear()

        self.edit\_2.clear()

        self.edit\_3.clear()

        self.edit\_4.clear()

        self.solve\_button.setEnabled(True)

if \_\_name\_\_ == "\_\_main\_\_":

    app = QApplication(sys.argv)

    maze\_solver = MazeSolver()

    maze\_solver.show()

    sys.exit(app.exec\_())

# Application of Binary Tree -– Expression Evaluation

## 1.Course Design Topics and Requirements

【**Problem Description**】

There is a natural correspondence between an expression and a binary tree. Suppose an arithmetic expression can contain variables (a~z), constants (0~9), parentheses, and binary operators (+,-,\*,/).

【**Basic Requirements**】

(1) Enter the arithmetic expression in the infix form and check the validity;

(2) Firstly, use the stack to change the infix form of the expression into the suffix form and output it. Secondly, construct the corresponding binary tree according to the suffix form, and print the construction process of the binary tree.

(3) Perform pre-order traversal, in-order traversal and post-order traversal on the constructed binary tree. Print the traversal sequence.

(4) Assign values to each variable, evaluate the expression based on the constructed

binary tree, and print the calculation process.

(5) Test several different data sets to accomplish the above requirements.

【**Extended Requirements**】

Construct a new compound expression (E1) P (E2), where P is the operator and E1 and E2 represent two valid infix arithmetic expressions, respectively. Construct a corresponding binary tree for the compound expression, print the binary tree and evaluate it.

## 2.Requirement Analysis

【Basic functional requirements】

(1) Enter an arithmetic expression in infix form.

(2) Output the suffix form of its arithmetic expression. Output the binary tree construction process. Output the sequence of pre order traversal, mid order traversal, and post order traversal of a binary tree.

(3) Assign values to variables and evaluate expressions.

【User interface requirements】

(1) Users need to be prompted with information to input the format of infix arithmetic expressions.

(2) When outputting, it is necessary to provide users with prompt information, indicating the introduction of the output content.

## 3.Design

### 3.1 Design Concept

#### 3.1.1 Data Structure Design

(1) Stack: According to the requirements, a stack is used to convert infix arithmetic expressions to postfix form and output the result. The stack ensures that operators are applied in the correct priority order and handles parentheses correctly. It temporarily stores operators and parentheses, helping the algorithm to process the expression accurately and ultimately generate the postfix expression.

(2) Binary Tree: According to the requirements, a binary tree is constructed from the postfix expression and used to compute the expression. The binary tree represents the expression structure, where each node can be an operator or an operand. This structure facilitates the computation and visualization of the expression.

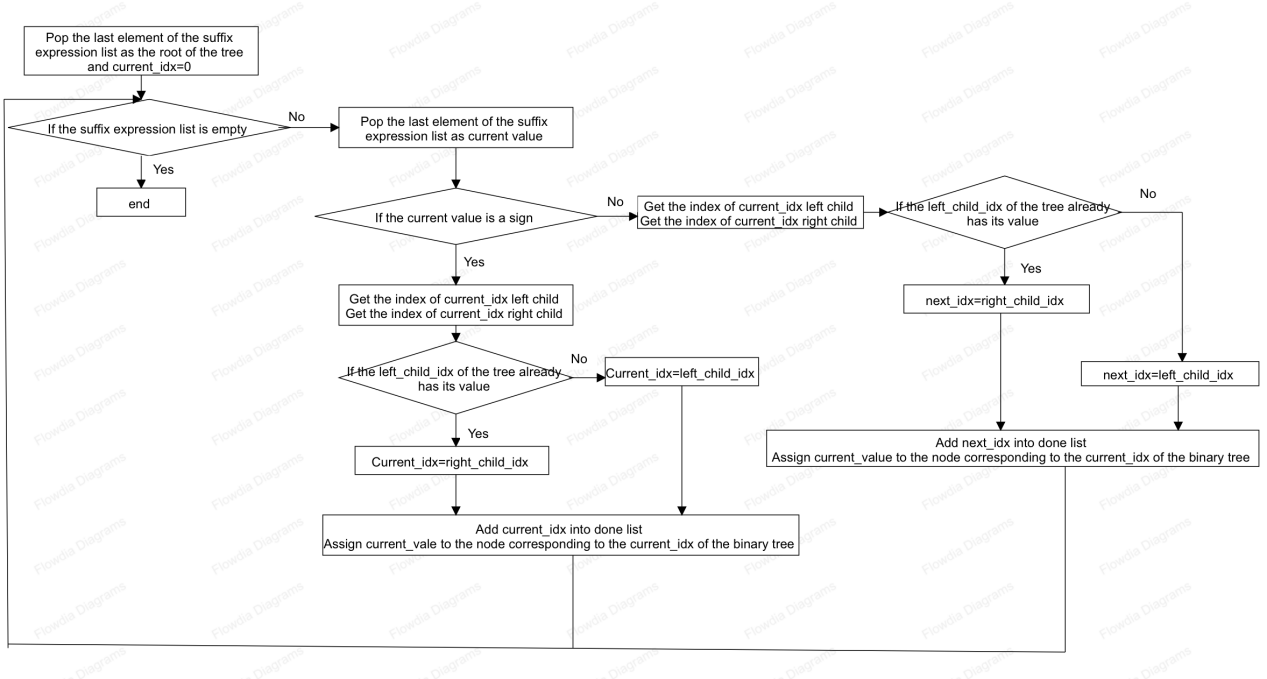
(3) List: According to the requirements, lists are used to store the input infix expressions and the converted postfix expressions. Based on the specified range of elements in the arithmetic expressions, it is necessary to check whether elements fall within the range and their respective categories. Multiple lists, stored as strings, contain elements within each category range for lookup purposes. During the binary tree conversion, lists store the indices of assigned binary tree nodes to determine if a coordinate can still be assigned a value.

(4) Dictionary: According to algorithm requirements, a dictionary stores symbols and their corresponding weights to better determine the computation priority of each symbol. During the final assignment and computation, the dictionary stores unknowns and their corresponding values, allowing for direct lookups of unknown values and establishing a more effective correspondence.

#### 3.1.2 Algorithm Design

(1) Constructing a Binary Tree

The postfix expression is characterized by operators being at the end, numbers at the beginning, and truncated numbers. Based on this characteristic, elements of the postfix expression are sequentially extracted from the end to the beginning and inserted into the binary tree from top to bottom. The process is managed through a `while` loop. Outside the loop, remove and retrieve the last element from the postfix expression list and place it at index 0 of the binary tree as the root. Inside the loop, remove and retrieve the last element from the postfix expression list as the current value. Based on the type of the current value, determine its corresponding index and insert the value into the binary tree at this index. Add the current index to the `done` list to indicate that it has been assigned a value. Continue this process until the postfix expression list is completely emptied, at which point the length is 0, and the loop ends, resulting in the completed binary tree.

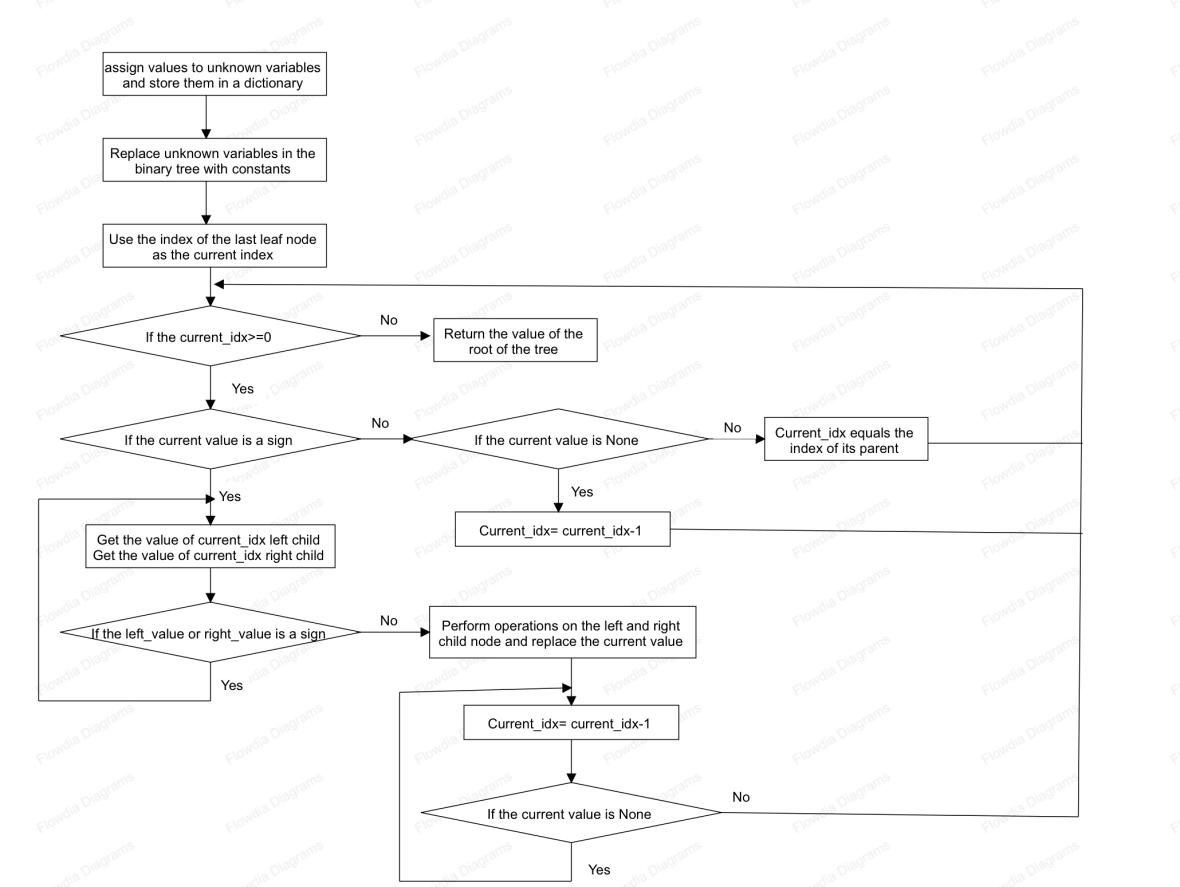
The design flowchart for constructing the binary tree is shown in Figure 2.1.

**Figure 2.1 Design Flowchart for Constructing a Binary Tree**

1. Calculating the Binary Tree

Based on the characteristics of the binary tree constructed from the postfix expression, calculate the binary tree from bottom to top and from right to left. Start by obtaining the index of the last leaf node as the current index. Use a `while` loop for the overall process. Inside the loop, determine whether to perform calculations and confirm the next index based on the type of the value at the current index in the binary tree. When the value is not an operator, it can be a number, an unknown variable, or `None`. For the first two cases, directly return the parent node's index as the current index. For the last case (`None`), return the previous index as the current index. When the value is an operator, retrieve the values of its left and right children, perform the corresponding calculation using a sub-calculation function, and assign the result to the current index. Then return the previous index as the current index. The loop ends when the current index is less than 0, with the value at the root node being the final result of the calculation.

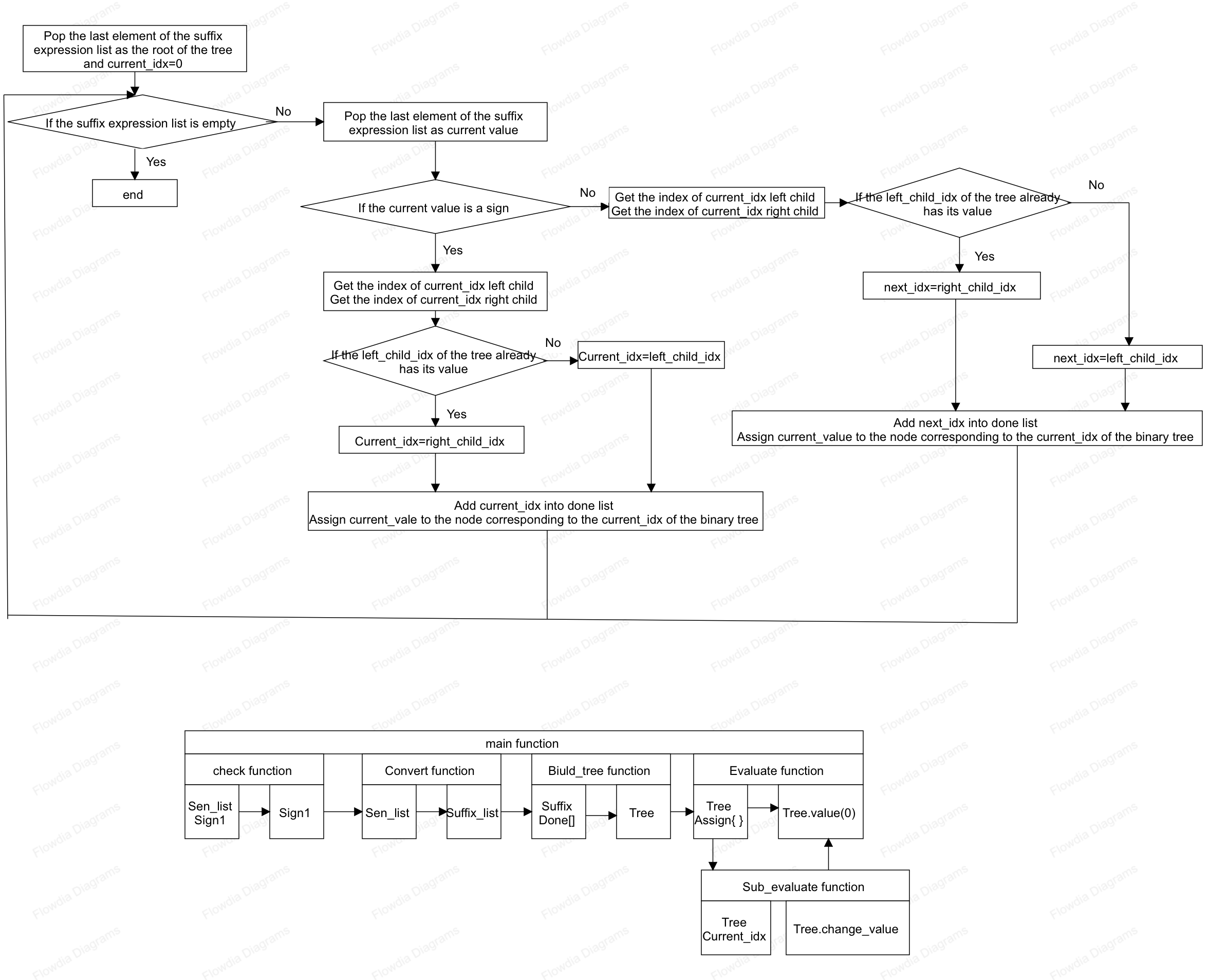
The design flowchart for calculating the binary tree is shown in Figure 2.2.



**Figure 2.2 Design Flowchart for Evaluating a Binary Tree**

### 3.2 Design Presentation

The main function call relationships are illustrated in Figure 2.3.

**Figure 2.3 Function Call Relationship Diagram**

(1) check Function Module: This module takes a list of infix arithmetic expressions as input. It initializes a boolean variable `sign1` to `False` and uses an exhaustive method to determine whether the infix expressions are valid. If they are not valid, `sign1` is set to `True`. The function ultimately returns the value of `sign1`.

(2) Convert Function Module: This module takes a list of infix arithmetic expressions as input, rearranges and processes it, and outputs a list of postfix expressions, `suffix\_list`.

(3) Build\_tree Function Module: This module takes a list of postfix expressions, `suffix`, as input and generates a temporary list, `done`, to record the indices that have been assigned. After applying the algorithm, it produces a binary tree, `tree`.

(4) Evaluate Function Module: This module takes a binary tree as input and creates a temporary dictionary, `assign`, to store the values assigned to unknowns. After applying the algorithm, it returns the value at the root node of the tree. Within this function, the `sub\_evaluate` function is also used to compute and update the values of the operator nodes.

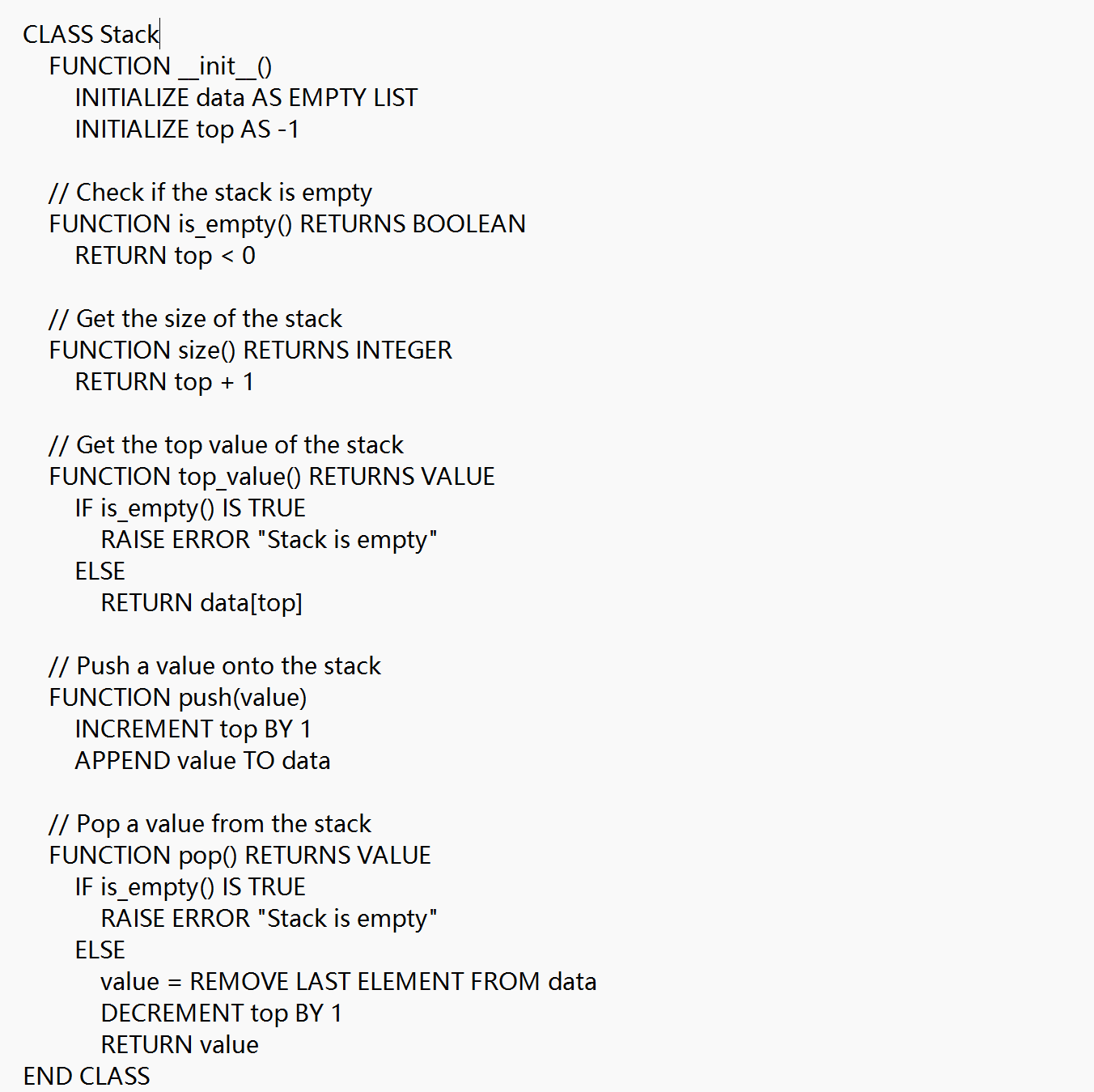
### 3.3 Detailed Design

#### 3.3.1 Basic Functionality Program

(1) Stack Class Design

A list `data` is used to store the elements in the stack, and an integer `top` represents the index of the top element, initialized to -1. The `Is\_empty()` function checks if the stack is empty by determining if `top` is less than 0. If `top` is less than 0, it indicates that the top element does not exist and the stack is empty; otherwise, the stack is not empty and the function returns a boolean value. The `Size()` function returns the number of elements in the stack. `top` represents the index of the top element, which is the index of the last element in the list `data`. Adding 1 to this index gives the length of the list, or the function can directly return the length of `data`. The `Top\_value()` function returns the current top element, which is the last element in the list `data`. The `Push(value)` function adds `value` to the stack as the new top element, which involves appending `value` to the end of the list `data` and incrementing the `top` value by one. The `Pop()` function removes and returns the top element from the stack, which involves deleting the last element from the list `data` and decrementing the `top` value by one.

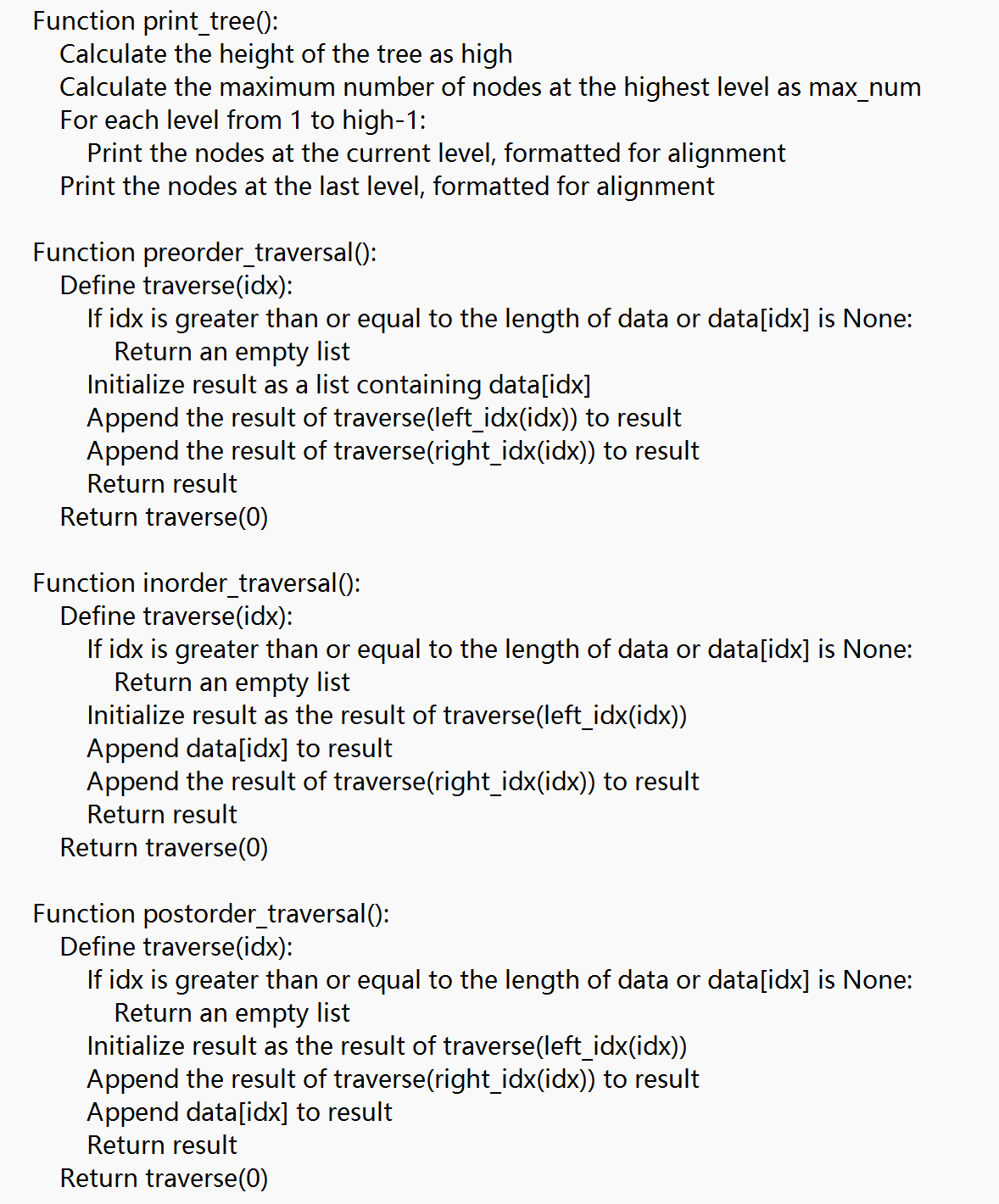
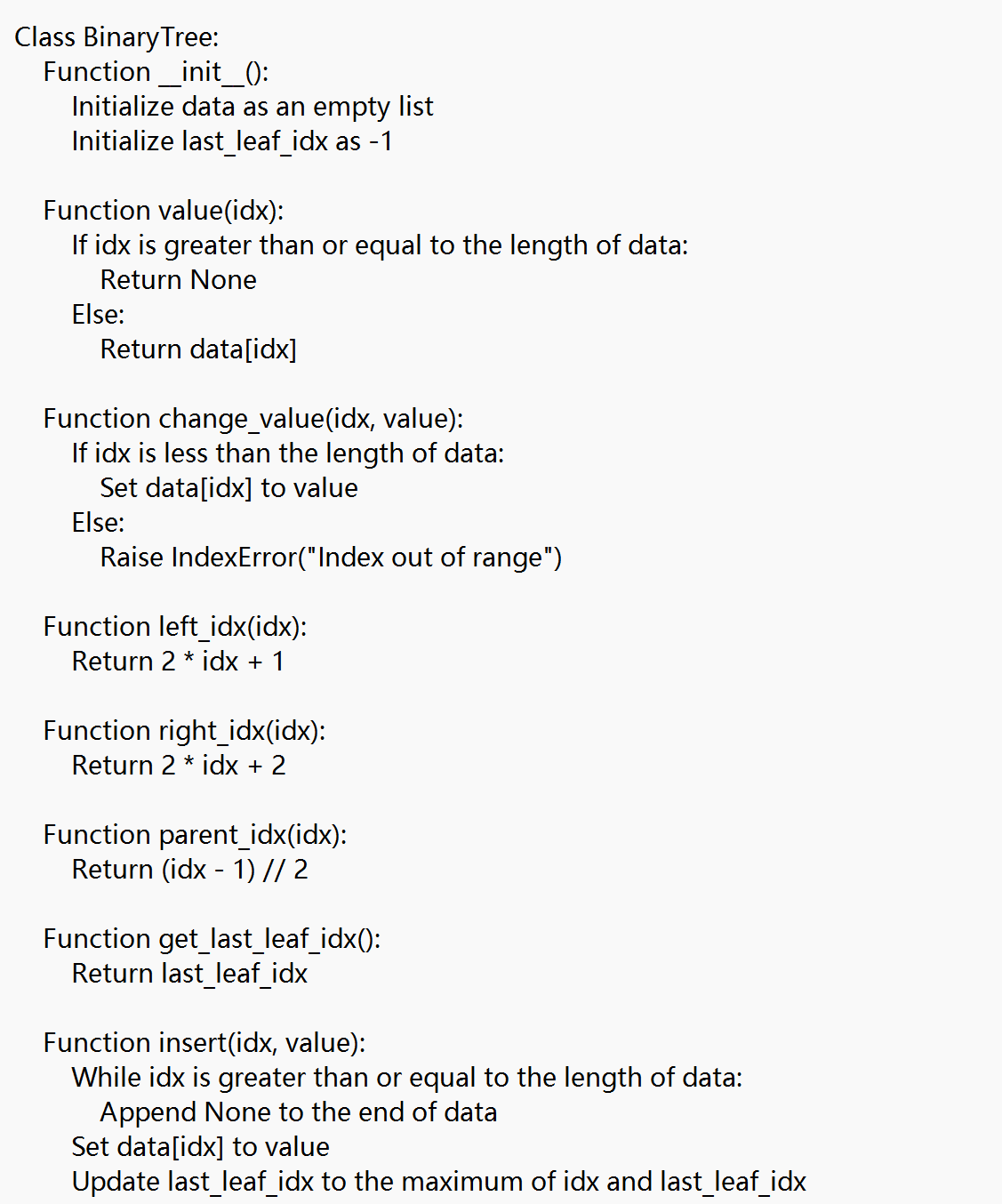
The pseudocode is shown in Figure 2.4.

**Figure 2.4 Pseudocode for Stack Class Design**

(2) Binary Tree Class Design

Initialize the binary tree using a list to store the tree structure, with the index of the last leaf node initialized to -1. The `value(idx)` function retrieves the value corresponding to the binary tree at index `idx`, returning the value directly from the list at that index. The `change\_value(idx, value)` function assigns the value `value` to the binary tree at index `idx`, enforcing the assignment directly in the list. The `left\_idx(idx)` and `right\_idx(idx)` functions calculate the indices of the left and right children of the current index `idx`, respectively. The `print\_tree()` function prints the binary tree by computing the height of the tree and the number of nodes at each level, printing each node level by level and handling alignment for readability. The `preorder\_traversal()` function performs a preorder traversal of the binary tree using a recursive algorithm, starting from the root node and visiting the left subtree followed by the right subtree. The `inorder\_traversal()` function performs an inorder traversal of the binary tree using a recursive algorithm, first visiting the left subtree, then the current node, and finally the right subtree. The `postorder\_traversal()` function performs a postorder traversal of the binary tree using a recursive algorithm, first visiting the left and right subtrees and then the current node.

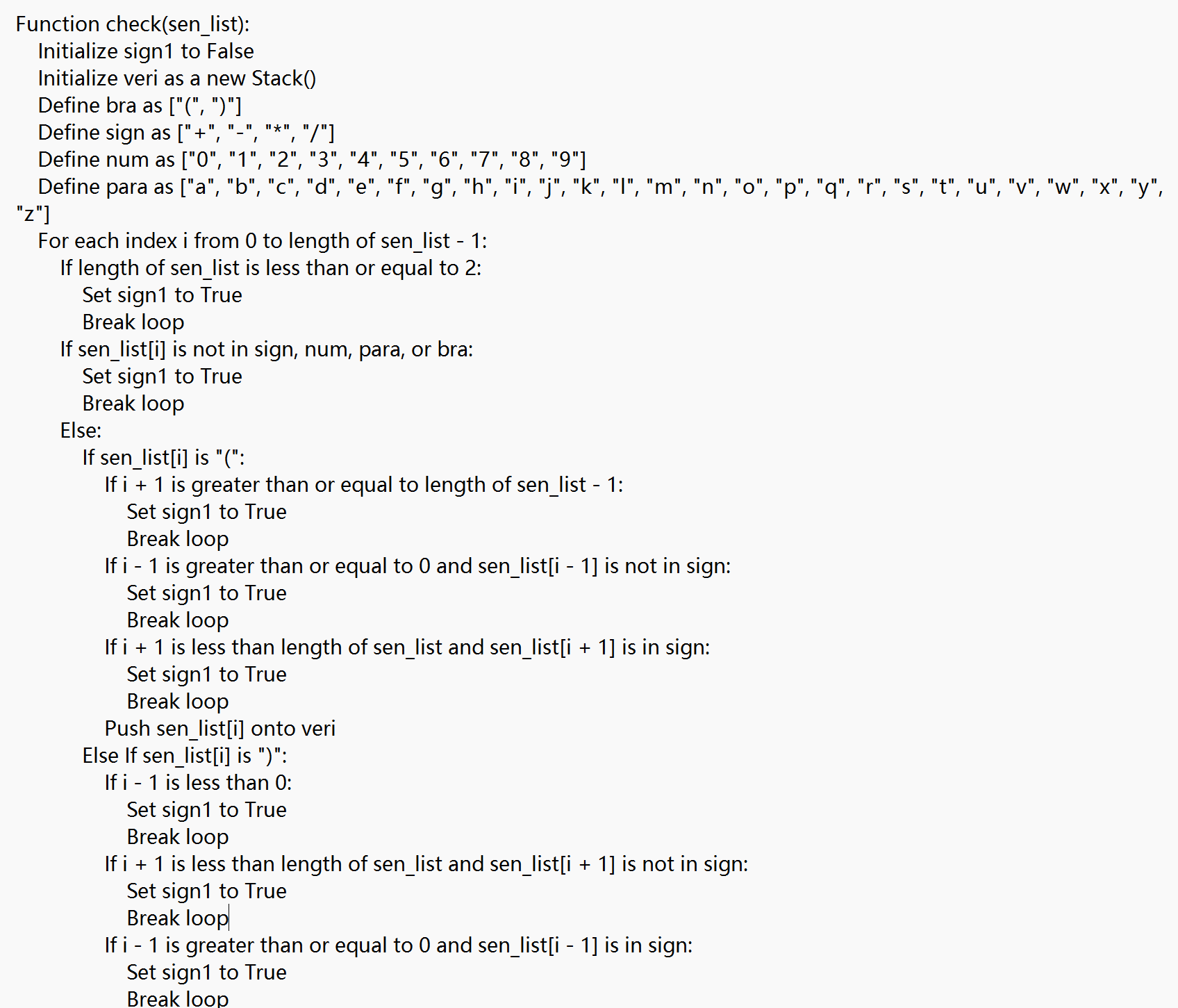
The pseudocode is shown in Figure 2.5.

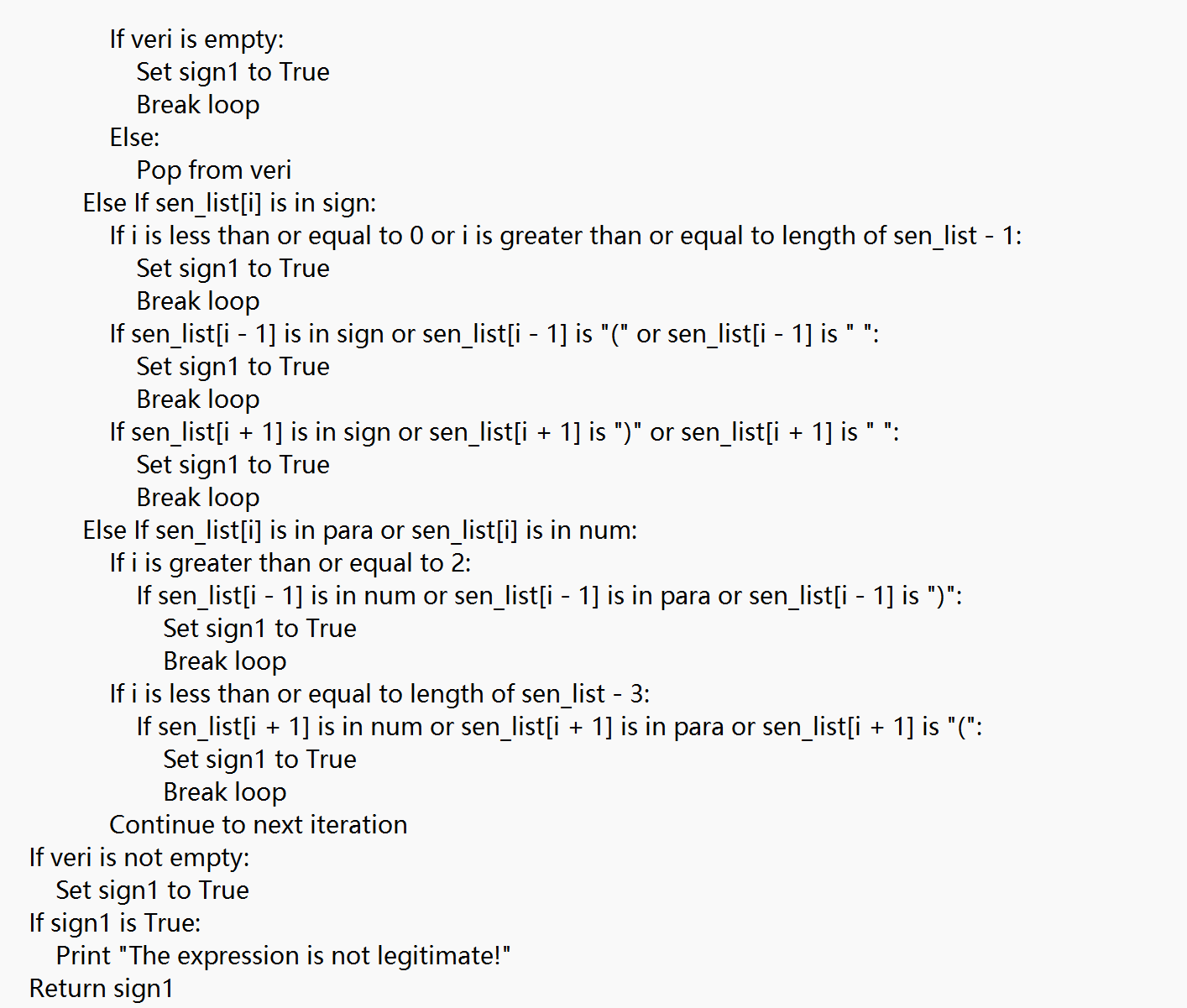
**Figure 2.5 Pseudocode for Binary Tree Class Design**

(3) Legality Verification Design

The `check` function is used to validate the syntax correctness of an arithmetic expression. It checks whether parentheses are matched, characters are valid, and operators are correctly placed. First, initialize a boolean flag `sign1` to indicate whether the expression is invalid, and create an instance of the `Stack` class to track opening parentheses and ensure they are paired. Lists `bra`, `sign`, `num`, and `para` define valid parentheses, operators, numbers, and variables, respectively. The function iterates through each character in `sen\_list`. If the expression length is less than or equal to 2, it is considered invalid because such an expression is too short to form a valid expression.

Check if the current character belongs to valid operators, numbers, variables, or parentheses; if not, the expression is invalid. Verify that left parentheses are not at the end of the expression, that they have valid characters before them, and that they are not immediately followed by operators. Push valid left parentheses onto the stack for tracking. If right parentheses are the first character, check that there are valid characters following them and operators preceding them. Attempting to close parentheses when the stack is empty indicates unmatched parentheses. Ensure that operators do not appear at the beginning or end of the expression, and that they are not directly preceded or followed by other operators or parentheses; operators should be surrounded by valid characters. Ensure correct placement of variables and numbers—they cannot directly follow other numbers, variables, or right parentheses and must be followed by valid characters. Finally, check if there are any unmatched parentheses remaining in the stack; if so, this indicates incomplete matching of parentheses and renders the expression invalid. If `sign1` is set to `True`, the expression is invalid, and an error message is printed. Return `sign1`, where `True` indicates an invalid expression and `False` indicates a valid one.

The pseudocode is shown in Figure 2.6.

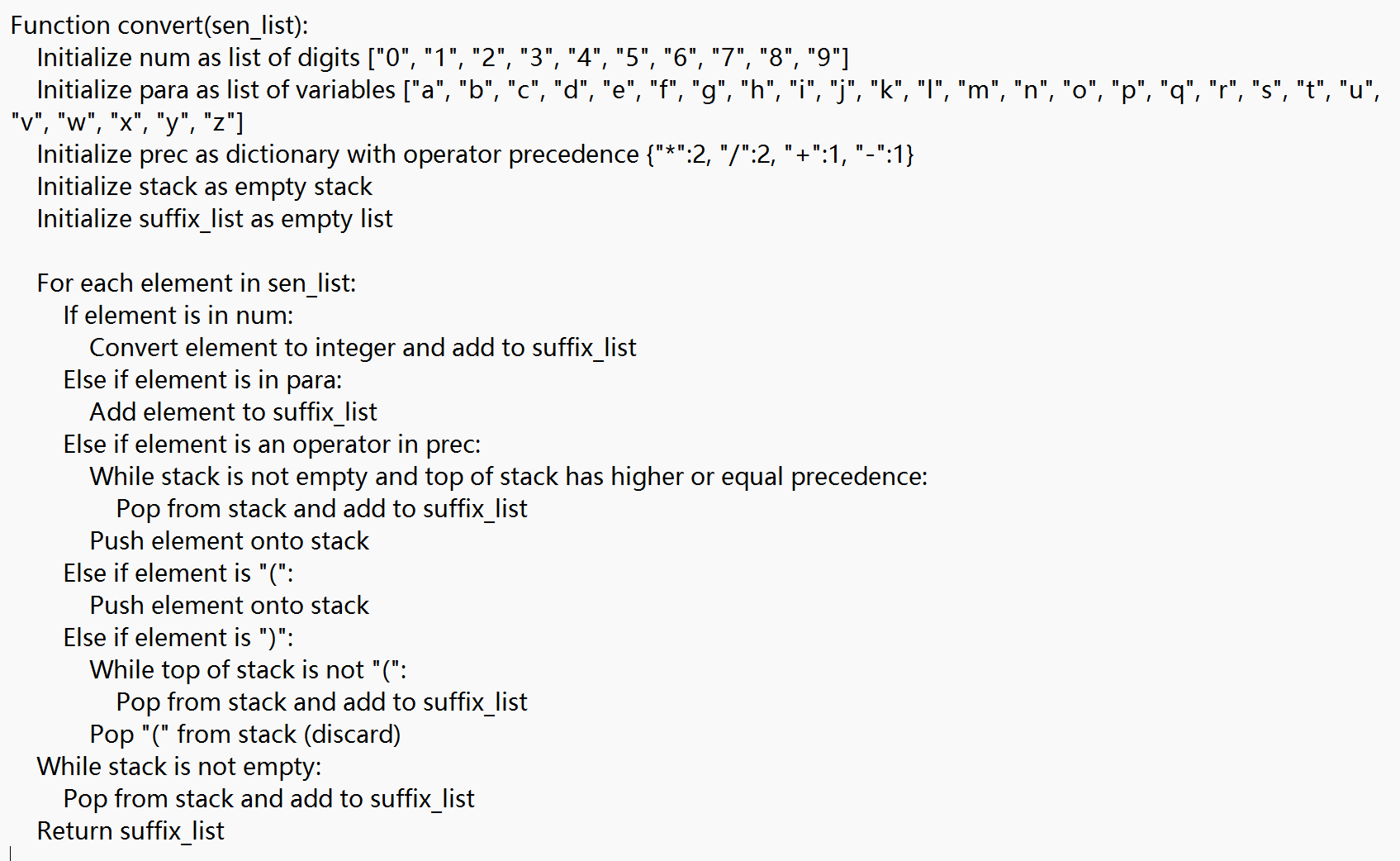
**Figure 2.6 Pseudocode for Legality Verification Design**

(4) Infix to Postfix Expression Conversion Design

Lists `bra`, `num`, and `para` define valid parentheses, numbers, and variables, respectively. A dictionary `prec` defines the operator precedence, where multiplication `\*` and division `/` have a precedence of 2, and addition `+` and subtraction `-` have a precedence of 1. The stack `s` is used to store operators and parentheses, and `suffix\_list` is used to store the converted postfix expression.

Traverse the input list. If the current element is a number, convert it to an integer and add it to `suffix\_list`. If the current element is a variable, directly add it to `suffix\_list`. If the current element is an operator, consider its preceding and succeeding symbols, as well as parentheses and precedence. First, check if the stack `s` is empty. If it is empty, this indicates no waiting operators, so push the current operator onto the stack. If the top element of the stack is a left parenthesis, push the current operator onto the stack as well. If the top element of the stack is an operator `j`, compare the precedence of the current operator with `j`. If the current operator has higher precedence, push it onto the stack. If the current operator does not have higher precedence, pop operators from the top of the stack (operators with higher precedence) and add them to `suffix\_list`, then push the current operator onto the stack. If the current element is a left parenthesis, push it onto the stack. If the current element is a right parenthesis, pop elements from the stack until a left parenthesis is encountered, and add these elements to `suffix\_list`. After all elements have been processed, pop the remaining operators from the stack and add them to `suffix\_list`.

The pseudocode is shown in Figure 2.7.

**Figure 2.7 Pseudocode for Infix to Postfix Expression Conversion Design**

(5) Binary Tree Construction Design

The list `sign` defines the operators, the list `done` is used to store assigned indices, and `current\_idx` is the index for the node currently being inserted, initialized to 0.

Start by popping the last value from the `suffix` list to serve as the root node of the tree and insert it into the tree. Then, enter a `while` loop to process each element of the postfix expression. When the current element is an operator, calculate the indices for the left and right child nodes, `left\_child\_idx` and `right\_child\_idx`. Check the `done` list to determine if the left and right child nodes have already been used. If the left child node has not been used, set `current\_idx` to the index of the left child node; if the left child node has been used, set `current\_idx` to the index of the right child node. Insert the operator into the tree, print the insertion process, and mark the current node index as used.

When the current value is an operand, similarly calculate the indices for the left and right child nodes. Determine if these child nodes have been used; if the left child node has not been used, set it as the index for the new node; if the left child node has been used, look for the right child node. If the right child node is also used, backtrack to the parent node and continue searching for an appropriate insertion position. Insert the operand into the tree, print the insertion process, and mark the new node index as used. Finally, return the constructed `tree` object.

The pseudocode is shown in Figure 2.8.

**Figure 2.8 Pseudocode for Binary Tree Construction Design**

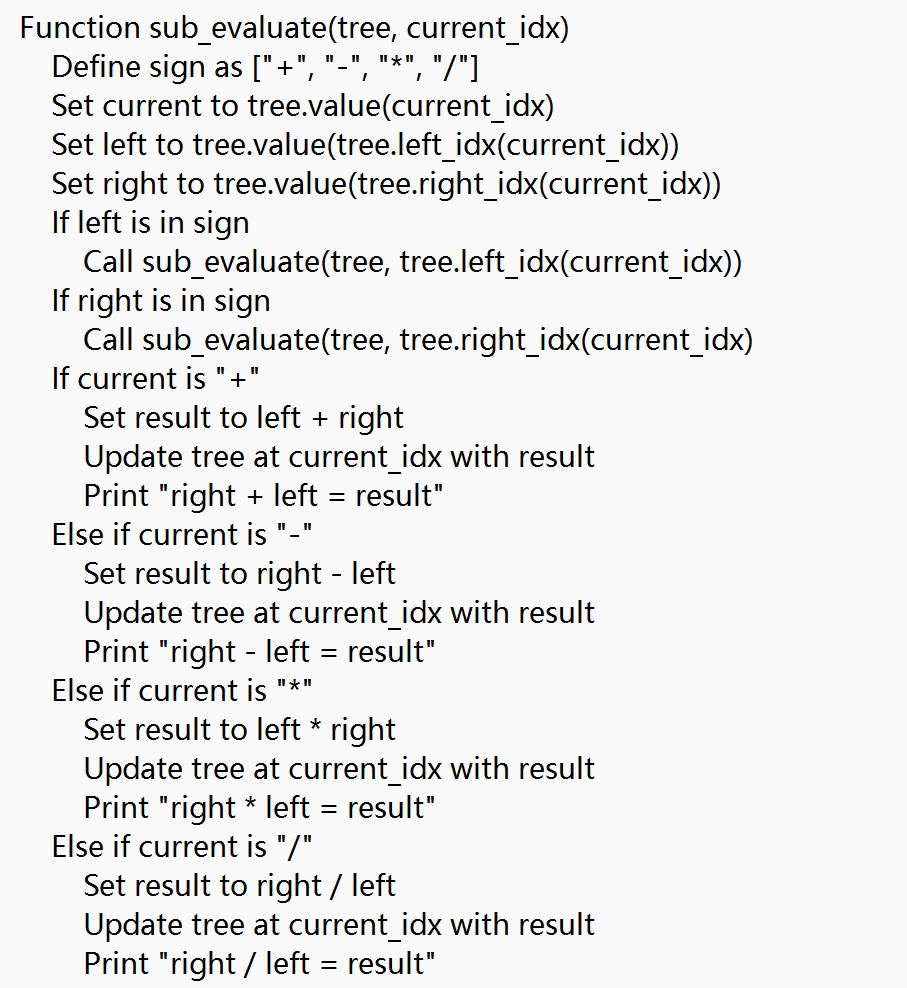
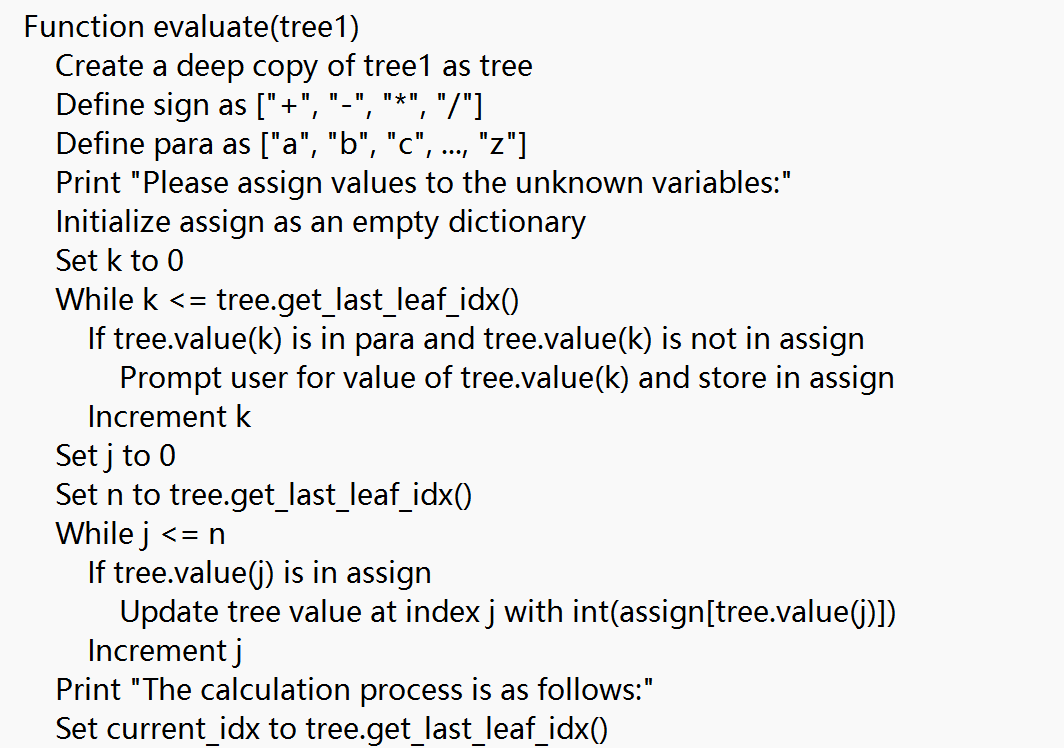
(6) Calculating the Binary Tree

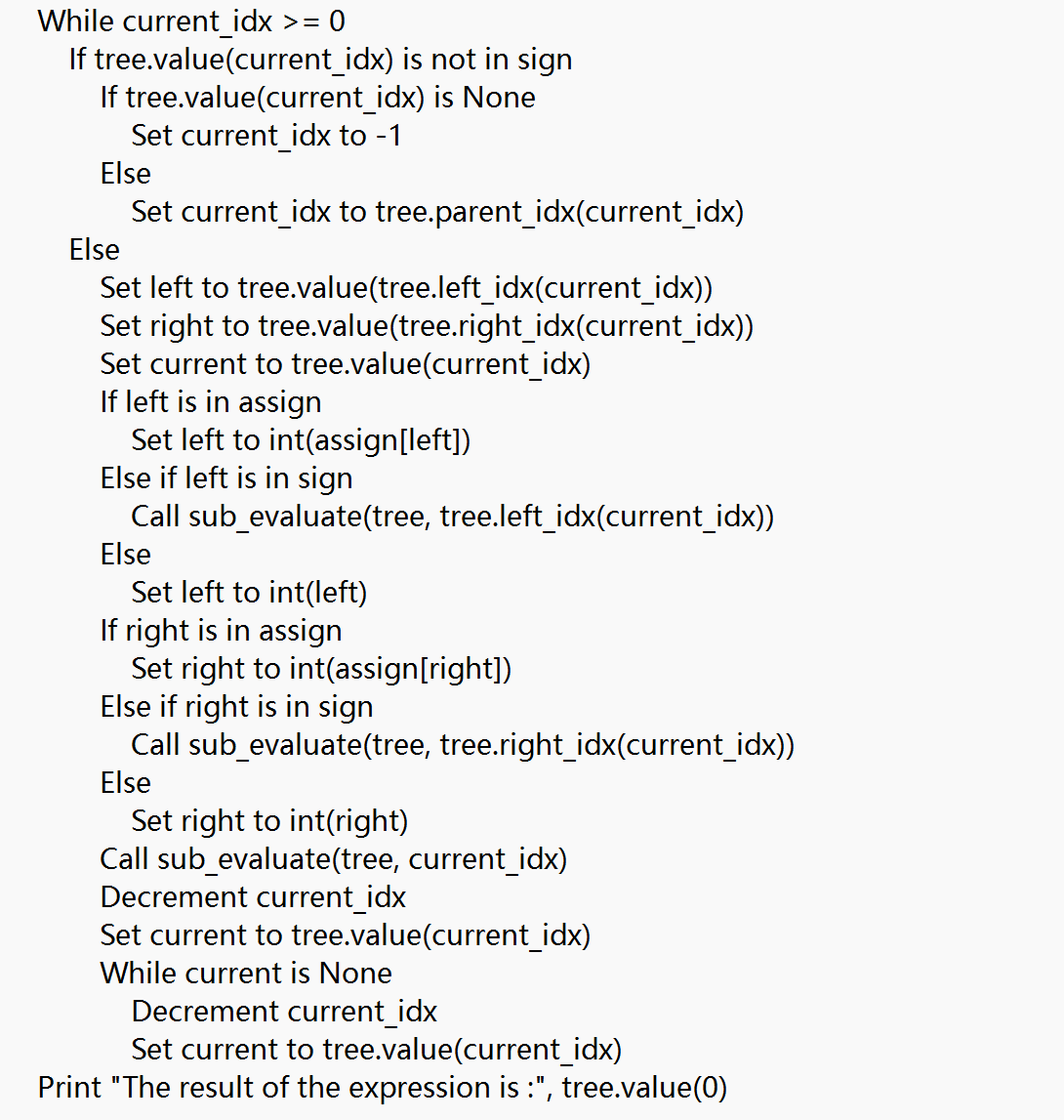
Since the main function repeatedly calls this function to assign values to unknowns in the binary tree, `copy.deepcopy` is used to copy the input tree `tree1` to ensure that the original tree remains unchanged. Traverse the binary tree to find unassigned unknowns, prompt the user to assign values to each unknown variable, and use the `assign` dictionary to store variables and their corresponding values. Then, traverse all nodes in the tree, updating the tree with the user-inputted variable values.

The `Evaluate` function computes the binary tree based on the characteristics of the postfix expression and the binary tree. The calculation proceeds from the leaf nodes up to the root node. First, locate the index of the last leaf node as `current\_idx`. Traverse the tree upwards from this node. If the current node is not an operator, check if its value is `None` or already assigned. If it has not been assigned, decrement the index until an operator is found. When an operator node is encountered, compute the values of its left and right child nodes. If the child nodes contain variables or operators, recursively call `sub\_evaluate` to calculate their values, then perform the calculation and update the current node's value. Continue processing each node until the calculations are complete.

In the `sub\_evaluate` function, `current\_idx` is the index of the current node passed from the `evaluate` function. Read the current node's value `current`, as well as the values of its left and right child nodes `left` and `right`. If either `left` or `right` is an operator, recursively call `sub\_evaluate` to compute their values. Otherwise, perform the corresponding operation based on the operator at `current`, update the current node's value accordingly.

The pseudocode is shown in Figure 2.9.





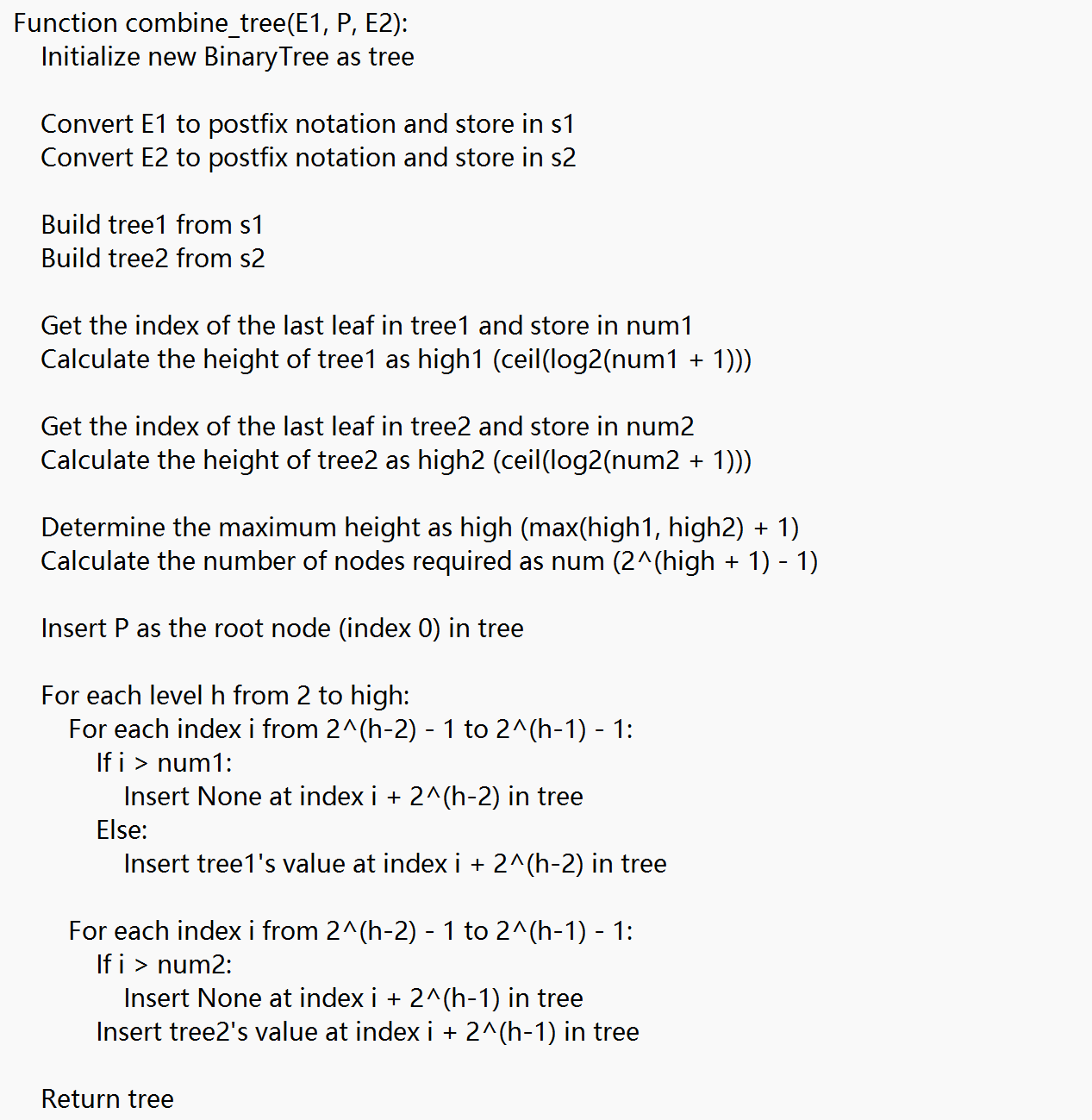
**Figure 2.9 Pseudocode for Binary Tree Calculation Design**

#### 3.3.2 External Program

In the extended program, first convert the two sub-expressions E1 and E2 from infix arithmetic expressions to postfix expressions. Then, construct binary trees from these postfix expressions. After that, create a new binary tree by adding a root node P and connecting the root nodes of the binary trees of E1 and E2 as its left and right children, respectively. Considering that subtraction and division involve the order of operands, where the previous `evaluate` function applied operations to the right child before the left child, E1 will be the right subtree and E2 the left subtree. Finally, use the `evaluate` function to compute the result of the combined binary tree.

In the `Combine` function, create a new empty binary tree `tree` as the combined binary tree. Use the `convert` function to convert expressions E1 and E2 to postfix notation, storing the results in `s1` and `s2`. Construct `tree1` and `tree2` from postfix notations `s1` and `s2`, respectively. Compute the heights `high1` and `high2` and the number of nodes `num1` and `num2` of these two trees. The height of the combined binary tree `high` is the maximum of `high1` and `high2` plus one, and `num` is the total number of nodes in the new binary tree. Insert the operator P into the root position of the new tree. Starting from the second level, traverse each level of the new tree, inserting nodes from `tree1` and `tree2` according to the tree's height and node index. For each level, first insert nodes from `tree1` into the right subtree of the new tree, then insert nodes from `tree2` into the left subtree. If there are not enough nodes in `tree1` or `tree2`, insert `None` in the corresponding positions. Finally, return the new combined binary tree `tree`.

Pseudocode for Merging Binary Trees is shown in Figure 2.10.

 **Figure 2.10 Pseudocode for Merging Binary Trees**

## Debugging Analysis

(1)Difficulties and Solutions: During the validation of infix expressions, errors often occurred due to overlooked scenarios. After multiple attempts and refinements, the process was improved. Initially, while calculating the binary tree, cases with multiple parentheses and sub-expressions with equal precedence were not considered, resulting in errors about invalid operations. Debugging revealed that errors occurred when moving to the root node. It was discovered that repeatedly returning to the parent node index caused only one subtree to be calculated, missing nodes at the same height. The issue was ultimately resolved by adjusting the index of symbol nodes and `None` nodes to locate the next node, enabling correct binary tree calculations.

(2)Time and Space Complexity Analysis:The `check` function has a time complexity of O(n), where n is the length of the expression, as it traverses the expression once to check parentheses pairing, operators, and operand positions. Its space complexity is O(n) for storing operand and parentheses states in a stack. The `convert` function also has a time complexity of O(n) for traversing the expression once, managing operators and parentheses with a stack, and generating the postfix expression. Its space complexity is O(n) for storing the postfix expression and stack operators. The `build\_tree` function has a time complexity of O(n), where n is the length of the postfix expression, as it traverses the postfix expression once to construct the binary tree. Its space complexity is O(n) for storing tree nodes and the postfix expression. The `sub\_evaluate` function has a time complexity of O(n), where n is the number of tree nodes, due to recursive calculations and node traversal. Its space complexity is O(h), where h is the tree's height, due to stack depth in recursion. The `evaluate` function has a time complexity of O(n), where n is the number of tree nodes, as it replaces leaf node values and performs calculations. Its space complexity is O(n) for storing tree nodes and value dictionaries.

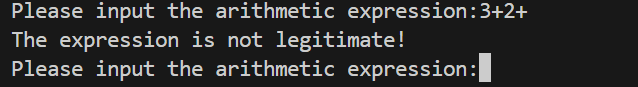
(3)Algorithm Discussion: Algorithms involving `while` loops can be transformed into recursive functions. For example, calculating a binary tree can be achieved by recursively evaluating from the root node, continually processing child nodes to obtain the final result.

## 5.User Manual

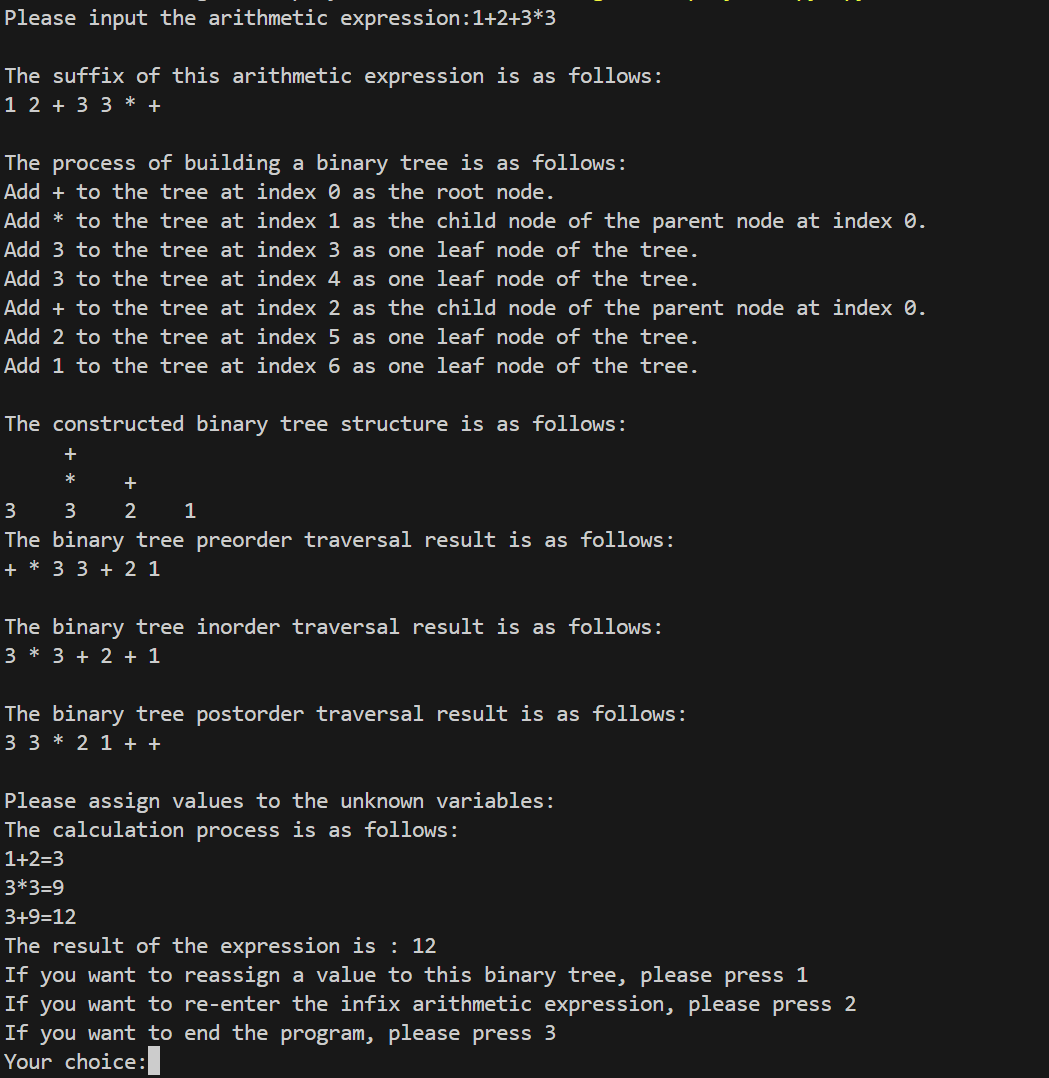
Enter the infix expression as prompted and press Enter to output the postfix expression and other required results. Then, assign values to any unknown variables as instructed and press Enter to perform the calculation. Finally, follow the prompts to either reassign values, input another expression, or exit the program.

## 6.Test Data And Test Results

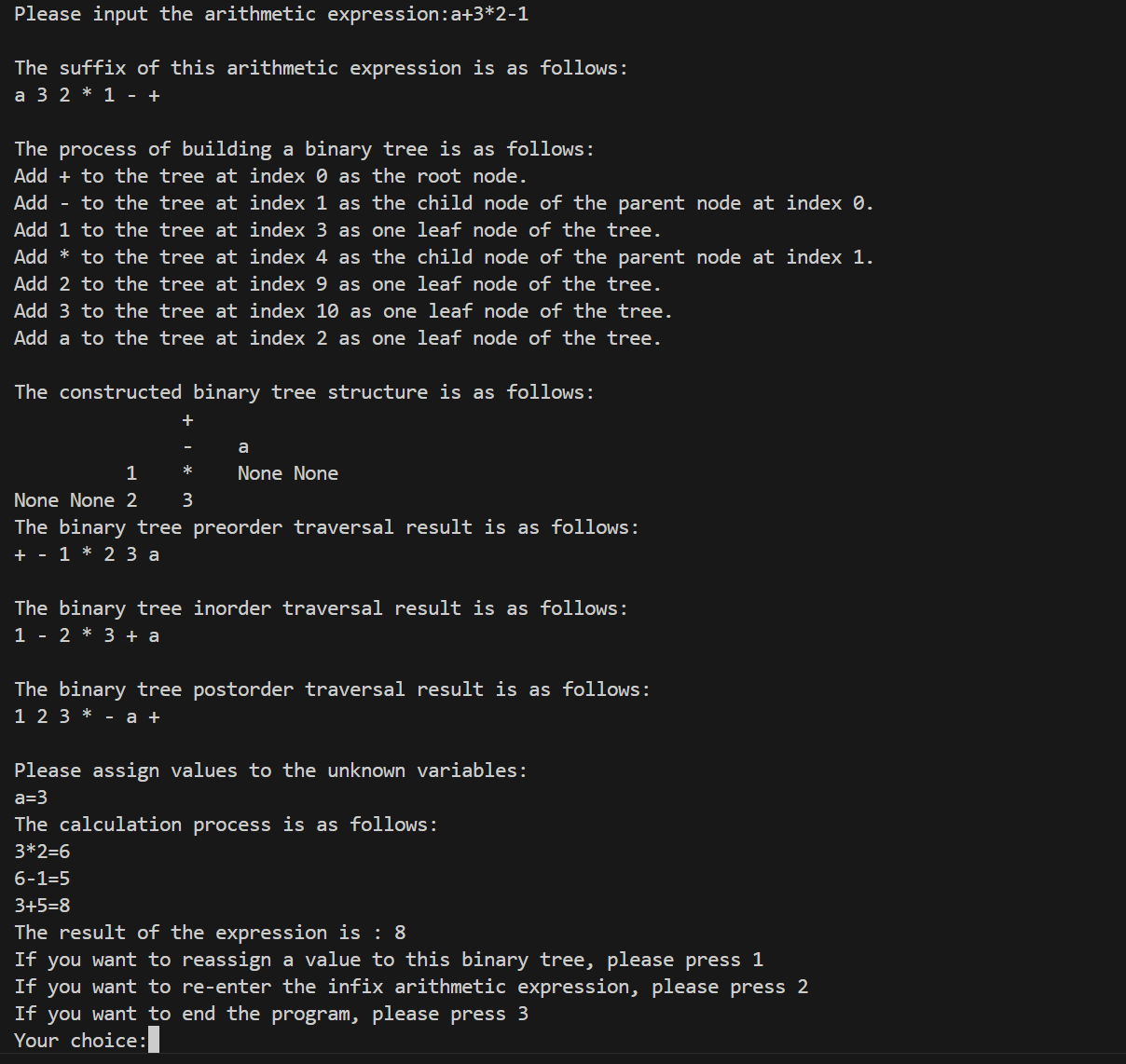
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test data | Purpose | Correct Output | Program Output | Current state |
| 3+2+ | If the program can detect invalid input in an infix expression. | The expression is not legitimate! | Figure 2.11.1 | Pass |
| 8+3+2-1\*1 | If the program can handle simple infix expressions. | 12 | Figure 2.11.2 | Pass |
| a+3\*2-1 | If the program can correctly perform value assignments and calculations. | 8 | Figure2.11.3 | Pass |
| 2\*(a-b)+8/2 | If the program can handle more complex infix expressions. | 12.0 | Figure 2.11.4 | Pass |
| (1+3\*a)+(2-b/3) | If the extended program can correctly handle the calculation of two fractions. | 7.0 | Figure 1.11.5 | Pass |

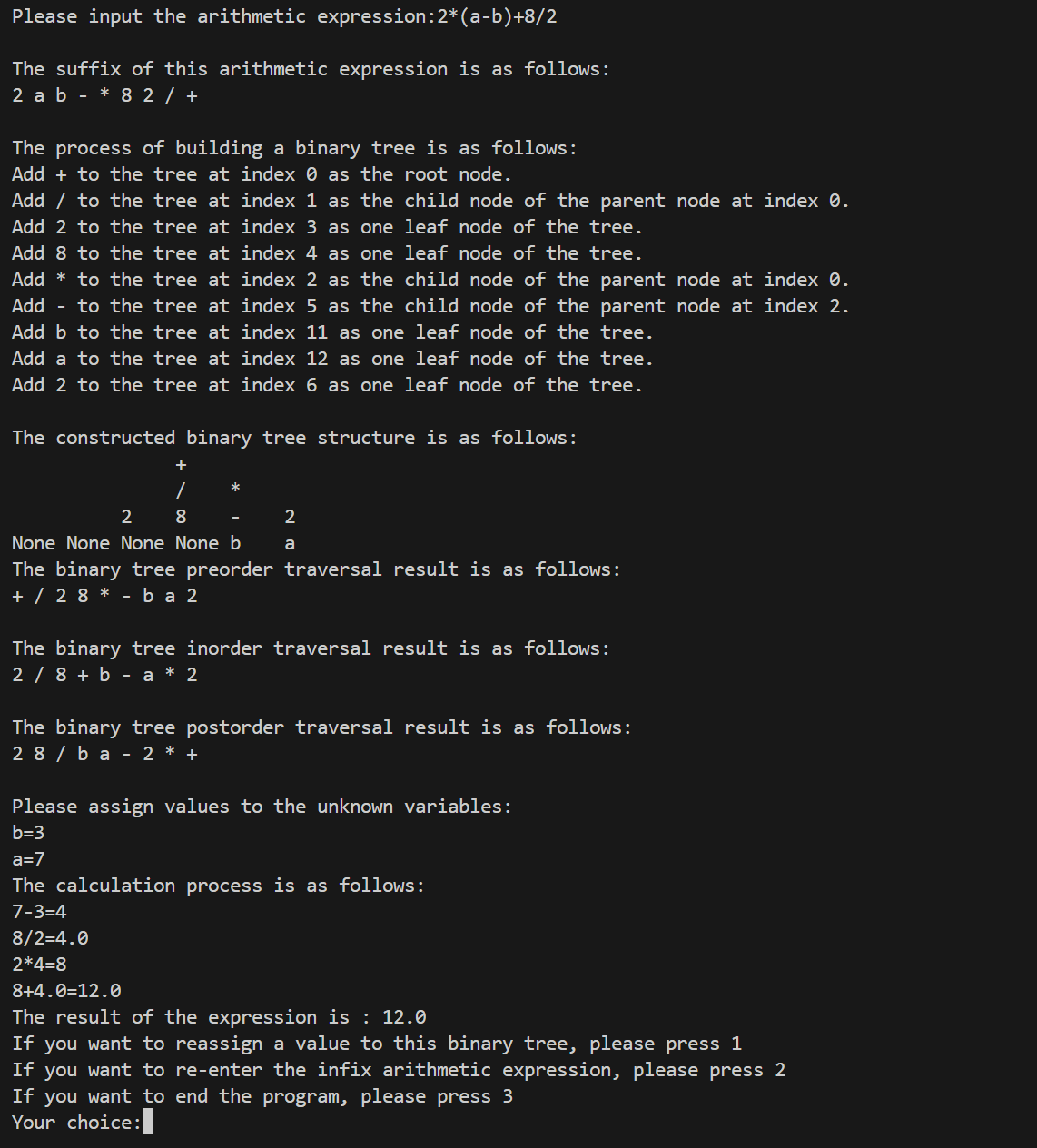


**Figure 2.11.1 Use Case 1 Output**

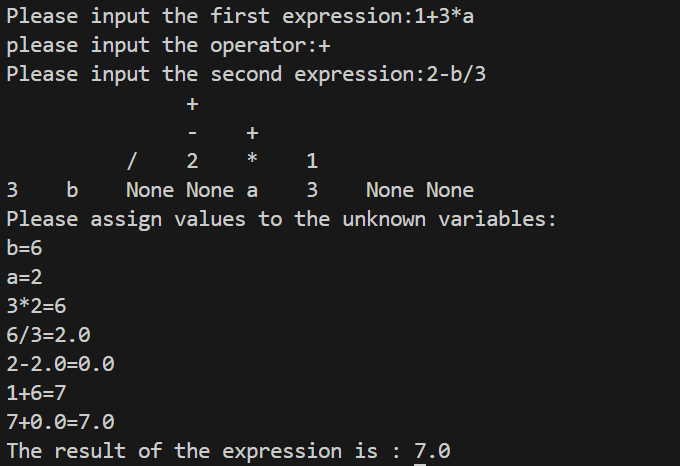


**Figure 2.11.2 Use Case Output**

**Figure 2.11.3 Use Case 3 Output**



**Figure 2.11.4 Use Case 4 Output**



**Figure 2.11.5 Use Case 5 Output**

## 7.Source Program Inventory

(1)Basic Functionality Program

import math

import copy

class Stack():

    def \_\_init\_\_(self):

        self.data=[]

        self.top=-1

    def is\_empty(self):

        return self.top<0

    def size(self):

        return self.top+1

    def top\_value(self):

        if self.is\_empty():

            raise KeyError("Stack is empty")

        else:

            return self.data[self.top]

    def push(self,value):

        self.top+=1

        self.data.append(value)

    def pop(self):

        if self.is\_empty():

            raise KeyError("Stack is empty")

        else:

            value=self.data.pop()

            self.top-=1

            return value

class BinaryTree:

    def \_\_init\_\_(self):

        self.data = []

        self.last\_leaf\_idx = -1

    def value(self, idx):

        if idx >= len(self.data):

            return None

        return self.data[idx]

    def change\_value(self, idx, value):

        if idx < len(self.data):

            self.data[idx] = value

        else:

            raise IndexError("Index out of range")

    def left\_idx(self, idx):

        return 2 \* idx + 1

    def right\_idx(self, idx):

        return 2 \* idx + 2

    def parent\_idx(self, idx):

        return (idx - 1) // 2

    def get\_last\_leaf\_idx(self):

        return self.last\_leaf\_idx

    def insert(self, idx, value):

        while idx >= len(self.data):

            self.data.append(None)

        self.data[idx] = value

        self.last\_leaf\_idx = max(self.last\_leaf\_idx, idx)

    def print\_tree(self):

        n=self.last\_leaf\_idx

        high=math.ceil(math.log((n+1),2))

        max\_num=2\*\*(high-1)

        for i in range(1,high):

            num=2\*\*(i-1)

            for k in range((max\_num-num)//2):

                print("     ",end="")

            for j in range(2\*\*(i-1)-1,2\*\*i-1):

                if self.data[j]==None:

                    print(self.data[j],end=" ")

                else:

                    print(self.data[j],end="    ")

            print("\n",end="")

        for i in range(2\*\*(high-1)-1,n+1):

            if self.data[i]==None:

                print(self.data[i],end=" ")

            else:

                print(self.data[i],end="    ")

    def preorder\_traversal(self):

        def traverse(idx):

            if idx >= len(self.data) or self.data[idx] is None:

                return []

            result = [self.data[idx]]

            result += traverse(self.left\_idx(idx))

            result += traverse(self.right\_idx(idx))

            return result

        return traverse(0)

    def inorder\_traversal(self):

        def traverse(idx):

            if idx >= len(self.data) or self.data[idx] is None:

                return []

            result = traverse(self.left\_idx(idx))

            result.append(self.data[idx])

            result += traverse(self.right\_idx(idx))

            return result

        return traverse(0)

    def postorder\_traversal(self):

        def traverse(idx):

            if idx >= len(self.data) or self.data[idx] is None:

                return []

            result = traverse(self.left\_idx(idx))

            result += traverse(self.right\_idx(idx))

            result.append(self.data[idx])

            return result

        return traverse(0)

#Legitimacy verification

#check if the parentheses are complete

def check(sen\_list):

    sign1=False

    veri=Stack()

    bra=["(",")"]

    sign=["+","-","\*","/"]

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

    para=["a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"]

    for i in range(len(sen\_list)):

        if len(sen\_list)<=2:

            sign1=True

            break

        elif sen\_list[i] not in sign and sen\_list[i] not in num and sen\_list[i] not in para and sen\_list[i] not in bra:

            sign1=True

            break

        else:

            if sen\_list[i]=="(":

                if i+1>=len(sen\_list)-1:

                    sign1=True

                    break

                if i-1>=0 and  sen\_list[i-1] not in sign:

                    sign1=True

                    break

                if i+1<len(sen\_list) and sen\_list[i+1] in sign:

                    sign1=True

                    break

                veri.push(sen\_list[i])

            elif sen\_list[i]==")":

                if i-1<0:

                    sign1=True

                    break

                if i+1<len(sen\_list) and sen\_list[i+1] not in sign:

                    sign1=True

                    break

                if i-1>=0 and sen\_list[i-1] in sign:

                    sign1=True

                    break

                if veri.is\_empty():

                    sign1=True

                    break

                else:

                    veri.pop()

            elif sen\_list[i] in sign:

                if i<=0 or i>=len(sen\_list)-1:

                    sign1=True

                    break

                if sen\_list[i-1] in sign or sen\_list[i-1]=="(" or sen\_list[i-1]==" ":

                    sign1=True

                    break

                if sen\_list[i+1] in sign or sen\_list[i+1]==")" or sen\_list[i+1]==" ":

                    sign1=True

                    break

            elif sen\_list[i] in para or sen\_list[i] in num:

                if i>=2:

                    if sen\_list[i-1] in num or sen\_list[i-1] in para or sen\_list[i-1]==")":

                        sign1=True

                        break

                if i<=len(sen\_list)-3:

                    if sen\_list[i+1] in num or sen\_list[i+1] in para or sen\_list[i+1]=="(":

                        sign1=True

                        break

                else:

                    continue

    if veri.is\_empty()==False:

        sign1=True

    if sign1==True:

        print("The expression is not legitimate!")

    return sign1

#convert the expression to the postfix expression

def convert(sen\_list):

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

    para=["a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"]

    prec={"\*":2,"/":2,"+":1,"-":1}

    s=Stack()

    suffix\_list=[]

    for i in range(len(sen\_list)):

        if sen\_list[i] in num:#if the current element is a number,turn it into an int form element here,which is convenient for subsequent calculations

            val=int(sen\_list[i])

            suffix\_list.append(val)

            continue

        elif sen\_list[i] in para:

            suffix\_list.append(sen\_list[i])

            continue

        else:

            if sen\_list[i] in prec:

                if s.is\_empty()==True:

                    s.push(sen\_list[i])

                else:

                    j=s.top\_value()

                    if j=="(":

                        s.push(sen\_list[i])

                    else:

                        if prec[sen\_list[i]]>prec[j]:

                            s.push(sen\_list[i])

                        elif prec[sen\_list[i]]<=prec[j]:

                            s.pop()

                            suffix\_list.append(j)

                            s.push(sen\_list[i])

            elif sen\_list[i]=="(":

                 s.push(sen\_list[i])

            elif sen\_list[i]==")":

                j=s.pop()

                while j!="(":

                    suffix\_list.append(j)

                    j=s.pop()

    while not s.is\_empty():

        j=s.pop()

        suffix\_list.append(j)

    return suffix\_list

def build\_tree(suffix):

    print("\nThe process of building a binary tree is as follows:")

    sign=["+","-","\*","/",]

    tree = BinaryTree()

    done=[]

    current\_idx=0

    current\_value=suffix.pop(-1)

    tree.insert(current\_idx,current\_value)

    print(f"Add {current\_value} to the tree at index {current\_idx} as the root node. ")

    done.append(current\_idx)

    while len(suffix)>0:

        current\_value=suffix.pop(-1)

        if current\_value in sign:

            left\_child\_idx=tree.left\_idx(current\_idx)

            right\_child\_idx=tree.right\_idx(current\_idx)

            if left\_child\_idx not in done:

                current\_idx=left\_child\_idx

            else:

                if right\_child\_idx not in done:

                    current\_idx=right\_child\_idx

            done.append(current\_idx)

            tree.insert(current\_idx,current\_value)

            print(f"Add {current\_value} to the tree at index {current\_idx} as the child node of the parent node at index {tree.parent\_idx(current\_idx)}.")

        else:

            left\_child\_idx=tree.left\_idx(current\_idx)

            right\_child\_idx=tree.right\_idx(current\_idx)

            if left\_child\_idx not in done:

                next\_idx=left\_child\_idx

            else:

                if right\_child\_idx not in done:

                    next\_idx=right\_child\_idx

                    current\_idx=tree.parent\_idx(current\_idx)

                else:

                    while right\_child\_idx in done:

                        next\_idx=tree.parent\_idx(next\_idx)

                        right\_child\_idx=tree.right\_idx(next\_idx)

                    next\_idx=right\_child\_idx

            done.append(next\_idx)

            tree.insert(next\_idx,current\_value)

            print(f"Add {current\_value} to the tree at index {next\_idx} as one leaf node of the tree.")

    return tree

def sub\_evaluate(tree,current\_idx):

    sign=["+","-","\*","/",]

    current=tree.value(current\_idx)

    left=tree.value(tree.left\_idx(current\_idx))

    right=tree.value(tree.right\_idx(current\_idx))

    if left in sign:

        sub\_evaluate(tree,tree.left\_idx(current\_idx))

    if right in sign:

        sub\_evaluate(tree,tree.right\_idx(current\_idx))

    if current=="+":

        tree.change\_value(current\_idx,left+right)

        print(f"{right}+{left}={left+right}")

    elif current=="-":

        tree.change\_value(current\_idx,right-left)

        print(f"{right}-{left}={right-left}")

    elif current=="\*":

        tree.change\_value(current\_idx,left\*right)

        print(f"{right}\*{left}={left\*right}")

    elif current=="/":

        tree.change\_value(current\_idx,right/left)

        print(f"{right}/{left}={right/left}")

def evaluate(tree1):

    tree = copy.deepcopy(tree1)

    sign=["+","-","\*","/",]

    para=["a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"]

    print("\nPlease assign values to the unknown variables:")

    k=0

    assign={}

    while k<tree.get\_last\_leaf\_idx()+1:

        if tree.value(k) in para and tree.value(k) not in assign.keys():

            info=input(f"{tree.value(k)}=")

            assign[tree.value(k)]=info

        k+=1

    j=0

    n=tree.get\_last\_leaf\_idx()

    while j<=n:

        if tree.value(j) in assign.keys():

            tree.change\_value(j,int(assign[tree.value(j)]))

        j+=1

    #find the index of the last leaf

    print("The calculation process is as follows:")

    current\_idx=tree.get\_last\_leaf\_idx()

    while current\_idx>=0:

        if tree.value(current\_idx) not in sign:#not a operator

            if tree.value(current\_idx)==None:

                current\_idx-=1

            else:

                current\_idx=tree.parent\_idx(current\_idx)#find the next one

        else:#a operator

            left=tree.value(tree.left\_idx(current\_idx))

            right=tree.value(tree.right\_idx(current\_idx))

            current=tree.value(current\_idx)

            if left in assign.keys():

                left=int(assign[left])

            elif left in sign:

                sub\_evaluate(tree,tree.left\_idx(current\_idx))

            else:

                left=int(left)

            if right in assign.keys():

                right=int(assign[right])

            elif right in sign:

                sub\_evaluate(tree,tree.right\_idx(current\_idx))

            else:

                right=int(right)

            sub\_evaluate(tree,current\_idx)

            current\_idx=current\_idx-1

            current=tree.value(current\_idx)

            while current==None:

                current\_idx=current\_idx-1#find the next one

                current=tree.value(current\_idx)#find the next one

    print("The result of the expression is :",tree.value(0))

if \_\_name\_\_=="\_\_main\_\_":

    get=2

    while get==2:

        sen=str(input("Please input the arithmetic expression:"))

        sen\_list=[]

        for i in sen:

            sen\_list.append(str(i))

        if not check(sen\_list):

            suffix = convert(sen\_list)

            print("\nThe suffix of this arithmetic expression is as follows:")

            print(" ".join(map(str, suffix)))

            tree = build\_tree(suffix)

            print("\nThe constructed binary tree structure is as follows:")

            tree.print\_tree()

            print("\nThe binary tree preorder traversal result is as follows:")

            preorder\_result = tree.preorder\_traversal()

            print(" ".join(map(str, preorder\_result)))

            print("\nThe binary tree inorder traversal result is as follows:")

            inorder\_result = tree.inorder\_traversal()

            print(" ".join(map(str, inorder\_result)))

            print("\nThe binary tree postorder traversal result is as follows:")

            postorder\_result = tree.postorder\_traversal()

            print(" ".join(map(str, postorder\_result)))

            evaluate(tree)

            print("If you want to reassign a value to this binary tree, please press 1\nIf you want to re-enter the infix arithmetic expression, please press 2\nIf you want to end the program, please press 3")

            choose=int(input("Your choice:"))

            while choose==1:

                evaluate(tree)

            get=choose

(2)External Program

import math

import copy

class Stack():

    def \_\_init\_\_(self):

        self.data=[]

        self.top=-1

    def is\_empty(self):

        return self.top<0

    def size(self):

        return self.top+1

    def top\_value(self):

        if self.is\_empty():

            raise KeyError("Stack is empty")

        else:

            return self.data[self.top]

    def push(self,value):

        self.top+=1

        self.data.append(value)

    def pop(self):

        if self.is\_empty():

            raise KeyError("Stack is empty")

        else:

            value=self.data.pop()

            self.top-=1

            return value

class BinaryTree:

    def \_\_init\_\_(self):

        self.data = []

        self.last\_leaf\_idx = -1

    def value(self, idx):

        if idx >= len(self.data):

            return None

        return self.data[idx]

    def change\_value(self, idx, value):

        if idx < len(self.data):

            self.data[idx] = value

        else:

            raise IndexError("Index out of range")

    def left\_idx(self, idx):

        return 2 \* idx + 1

    def right\_idx(self, idx):

        return 2 \* idx + 2

    def parent\_idx(self, idx):

        return (idx - 1) // 2

    def get\_last\_leaf\_idx(self):

        return self.last\_leaf\_idx

    def insert(self, idx, value):

        while idx >= len(self.data):

            self.data.append(None)

        self.data[idx] = value

        self.last\_leaf\_idx = max(self.last\_leaf\_idx, idx)

    def print\_tree(self):

        n=self.last\_leaf\_idx

        high=math.ceil(math.log((n+1),2))

        max\_num=2\*\*(high-1)

        for i in range(1,high):

            num=2\*\*(i-1)

            for k in range((max\_num-num)//2):

                print("     ",end="")

            for j in range(2\*\*(i-1)-1,2\*\*i-1):

                if self.data[j]==None:

                    print(self.data[j],end=" ")

                else:

                    print(self.data[j],end="    ")

            print("\n",end="")

        for i in range(2\*\*(high-1)-1,n+1):

            if self.data[i]==None:

                print(self.data[i],end=" ")

            else:

                print(self.data[i],end="    ")

def check(sen\_list):

    sign1=False

    veri=Stack()

    bra=["(",")"]

    sign=["+","-","\*","/",]

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

    para=["a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"]

    for i in range(len(sen\_list)):

        if sen\_list[i] not in sign and sen\_list[i] not in num and sen\_list[i] not in para and sen\_list[i] not in bra:

            sign1=True

            break

        else:

            if sen\_list[i]=="(":

                if i-1>=0 and  sen\_list[i-1] not in sign:

                    sign1=True

                    break

                if i+1<len(sen\_list) and sen\_list[i+1] in sign:

                    sign1=True

                    break

                veri.push(sen\_list[i])

            elif sen\_list[i]==")":

                if i+1<len(sen\_list) and sen\_list[i+1] not in sign:

                    sign1=True

                    break

                if i-1>=0 and sen\_list[i-1] in sign:

                    sign1=True

                    break

                if veri.is\_empty():

                    sign1=True

                    break

                else:

                    veri.pop()

            elif sen\_list[i] in sign :

                if sen\_list[i-1] in sign or sen\_list[i-1]=="(":

                    sign1=True

                    break

                if sen\_list[i+1] in sign or sen\_list[i+1]==")":

                    sign1=True

                    break

    if veri.is\_empty()==False:

        sign1=True

    if sign1==True:

        print("The expression is not legitimate!")

    return sign1

def convert(sen\_list):

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

    para=["a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"]

    prec={"\*":2,"/":2,"+":1,"-":1}

    s=Stack()

    suffix\_list=[]

    for i in range(len(sen\_list)):

        if sen\_list[i] in num:#if the current element is a number,turn it into an int form element here,which is convenient for subsequent calculations

            val=int(sen\_list[i])

            suffix\_list.append(val)

            continue

        elif sen\_list[i] in para:

            suffix\_list.append(sen\_list[i])

            continue

        else:

            if sen\_list[i] in prec:

                if s.is\_empty()==True:

                    s.push(sen\_list[i])

                else:

                    j=s.top\_value()

                    if j=="(":

                        s.push(sen\_list[i])

                    else:

                        if prec[sen\_list[i]]>prec[j]:

                            s.push(sen\_list[i])

                        elif prec[sen\_list[i]]<=prec[j]:

                            s.pop()

                            suffix\_list.append(j)

                            s.push(sen\_list[i])

            elif sen\_list[i]=="(":

                 s.push(sen\_list[i])

            elif sen\_list[i]==")":

                j=s.pop()

                while j!="(":

                    suffix\_list.append(j)

                    j=s.pop()

    while not s.is\_empty():

        j=s.pop()

        suffix\_list.append(j)

    return suffix\_list

def build\_tree(suffix):

    sign=["+","-","\*","/",]

    tree = BinaryTree()

    done=[]

    current\_idx=0

    current\_value=suffix.pop(-1)

    tree.insert(current\_idx,current\_value)

    done.append(current\_idx)

    while len(suffix)>0:

        current\_value=suffix.pop(-1)

        if current\_value in sign:

            left\_child\_idx=tree.left\_idx(current\_idx)

            right\_child\_idx=tree.right\_idx(current\_idx)

            if left\_child\_idx not in done:

                current\_idx=left\_child\_idx

            else:

                if right\_child\_idx not in done:

                    current\_idx=right\_child\_idx

            done.append(current\_idx)

            tree.insert(current\_idx,current\_value)

        else:

            left\_child\_idx=tree.left\_idx(current\_idx)

            right\_child\_idx=tree.right\_idx(current\_idx)

            if left\_child\_idx not in done:

                next\_idx=left\_child\_idx

            else:

                if right\_child\_idx not in done:

                    next\_idx=right\_child\_idx

                else:

                    while right\_child\_idx in done:

                        next\_idx=tree.parent\_idx(next\_idx)

                        right\_child\_idx=tree.right\_idx(next\_idx)

                    next\_idx=right\_child\_idx

            done.append(next\_idx)

            tree.insert(next\_idx,current\_value)

    return tree

def combine\_tree(E1,P,E2):#put E2 at the left

    tree=BinaryTree()

    s1=convert(E1)

    s2=convert(E2)

    tree1=build\_tree(s1)

    tree2=build\_tree(s2)

    num1=tree1.get\_last\_leaf\_idx()

    high1=math.ceil(math.log((num1+1),2))

    num2=tree2.get\_last\_leaf\_idx()

    high2=math.ceil(math.log((num2+1),2))

    high=max(high1,high2)+1

    num=2\*\*(high+1)-1

    tree.insert(0,P)#the root node(high=1)

    for h in range(2,high+1):

        for i in range(2\*\*(h-2)-1,2\*\*(h-1)-1):

            if i>num1:

                tree.insert(i+2\*\*(h-2),None)

            else:

                tree.insert(i+2\*\*(h-2),tree1.value(i))

        for i in range(2\*\*(h-2)-1,2\*\*(h-1)-1):

            if i>num2:

                tree.insert(i+2\*\*(h-1),None)

            tree.insert(i+2\*\*(h-1),tree2.value(i))

    return tree

def sub\_evaluate(tree,current\_idx):

    sign=["+","-","\*","/",]

    current=tree.value(current\_idx)

    left=tree.value(tree.left\_idx(current\_idx))

    right=tree.value(tree.right\_idx(current\_idx))

    if left in sign:

        sub\_evaluate(tree,tree.left\_idx(current\_idx))

    if right in sign:

        sub\_evaluate(tree,tree.right\_idx(current\_idx))

    if current=="+":

        tree.change\_value(current\_idx,left+right)

        print(f"{right}+{left}={left+right}")

    elif current=="-":

        tree.change\_value(current\_idx,right-left)

        print(f"{right}-{left}={right-left}")

    elif current=="\*":

        tree.change\_value(current\_idx,left\*right)

        print(f"{right}\*{left}={left\*right}")

    elif current=="/":

        tree.change\_value(current\_idx,right/left)

        print(f"{right}/{left}={right/left}")

def evaluate(tree1):

    tree = copy.deepcopy(tree1)

    sign=["+","-","\*","/",]

    para=["a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"]

    print("\nPlease assign values to the unknown variables:")

    k=0

    assign={}

    while k<tree.get\_last\_leaf\_idx()+1:

        if tree.value(k) in para and tree.value(k) not in assign.keys():

            info=input(f"{tree.value(k)}=")

            assign[tree.value(k)]=info

        k+=1

    j=0

    n=tree.get\_last\_leaf\_idx()

    while j<=n:

        if tree.value(j) in assign.keys():

            tree.change\_value(j,int(assign[tree.value(j)]))

        j+=1

    #find the index of the last leaf

    current\_idx=tree.get\_last\_leaf\_idx()

    while current\_idx>=0:

        if tree.value(current\_idx) not in sign:#not a operator

            if tree.value(current\_idx)==None:

                current\_idx-=1

            else:

                current\_idx=tree.parent\_idx(current\_idx)#find the next one

        else:#a operator

            left=tree.value(tree.left\_idx(current\_idx))

            right=tree.value(tree.right\_idx(current\_idx))

            current=tree.value(current\_idx)

            if left in assign.keys():

                left=int(assign[left])

            elif left in sign:

                sub\_evaluate(tree,tree.left\_idx(current\_idx))

            else:

                left=int(left)

            if right in assign.keys():

                right=int(assign[right])

            elif right in sign:

                sub\_evaluate(tree,tree.right\_idx(current\_idx))

            else:

                right=int(right)

            sub\_evaluate(tree,current\_idx)

            current\_idx=current\_idx-1

            current=tree.value(current\_idx)

            while current==None:

                current\_idx=current\_idx-1#find the next one

                current=tree.value(current\_idx)#find the next one

    print("The result of the expression is :",tree.value(0))

if \_\_name\_\_=="\_\_main\_\_":

    e1=str(input("Please input the first expression:"))

    E1=[]

    for i in e1:

        E1.append(i)

    P=str(input("please input the operator:"))

    e2=str(input("Please input the second expression:"))

    E2=[]

    for i in e2:

        E2.append(i)

    if check(E1)==False and check(E2)==False:

        tree=combine\_tree(E2,P,E1)

        tree.print\_tree()

        evaluate(tree)