GROUP C

NAME	**STUDENT NO**	**REGISTRATION NO**
TUSIIME MABLE	2300701494	23/U/1494
BAINGANA ABISHA	2300707328	23/U/07328/PS
AINAMANI JIM	2300723830	23/U/23830/EVE
TENDO JEMIMAH NAKAYIWA	2300717920	23/U/17920/PS
BANTROBUSA KAZIBWE FZ	2300707416	23/U/07416/EVE

TASK 1: DATATYPES IN C AND JAVA

a) Built-in datatypes in C

Int	• Integer datatype
Float	• floating-point type
Double	• Double-precision floating-point type
char	• Character type
void	• Represents the absence of type

Built-in datatypes in Java

byte	• 8-bit signed integer
short	• 16-bit signed integer
int	• 32-bit signed integer
long	• 64-bit signed integer
float	• 32-bit signed integer
double	• 64-bit signed integer
char	• 16-bit Unicode character
boolean	• Represents true or false

b) User-defined datatypes in C

struct	• collection of variables grouped under a single name
union	• only one variable can store value at a time
typedef	• Creating an alias of an existing data type
enum	• Consists of a set of named integer constants.

User-defined datatypes in Java

- Classes
- Interfaces
- Enums

c) Built-in Datatypes in Java that are not in C

- Byte
- Short
- Long
- Boolean

TASK 2: ARRAYS

a) <u>Axioms to implement array operations</u>

1. Access (Retrieve an element)

To get the element at index i in an array arr, if i is within bounds ($0 \leq i < \text{length(arr)}$), return arr[i]. Otherwise, it's undefined.

2. Insert (Add an element at a position)

To insert an element x at index i in arr, put x at i and move elements to the right to make room.

3. Delete (Remove an element at a position)

To delete an element at index i, remove the element and shift everything after i one step to the left.

4. Length (Size of the array)

The length of arr is the number of elements in it.

5. Append (Add an element to the end)

To add an element x to the end, place x at index length(arr).

6. Update (Replace an element)

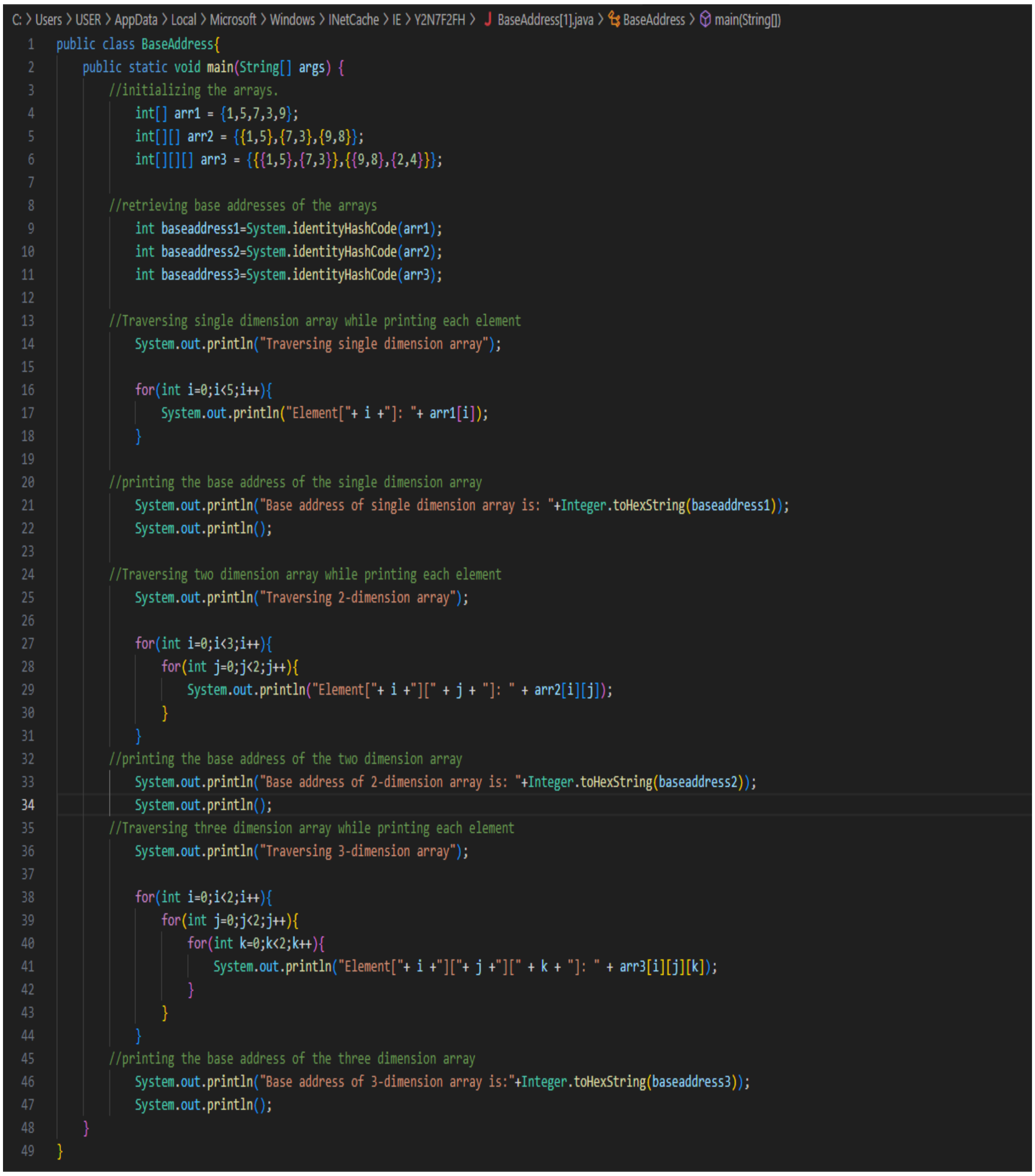
To change the element at index i to x, just replace arr[i] with x.

7. Find (Locate an element)

To find an element x, return the index of the first occurrence of x in arr. If x isn't there, return undefined.

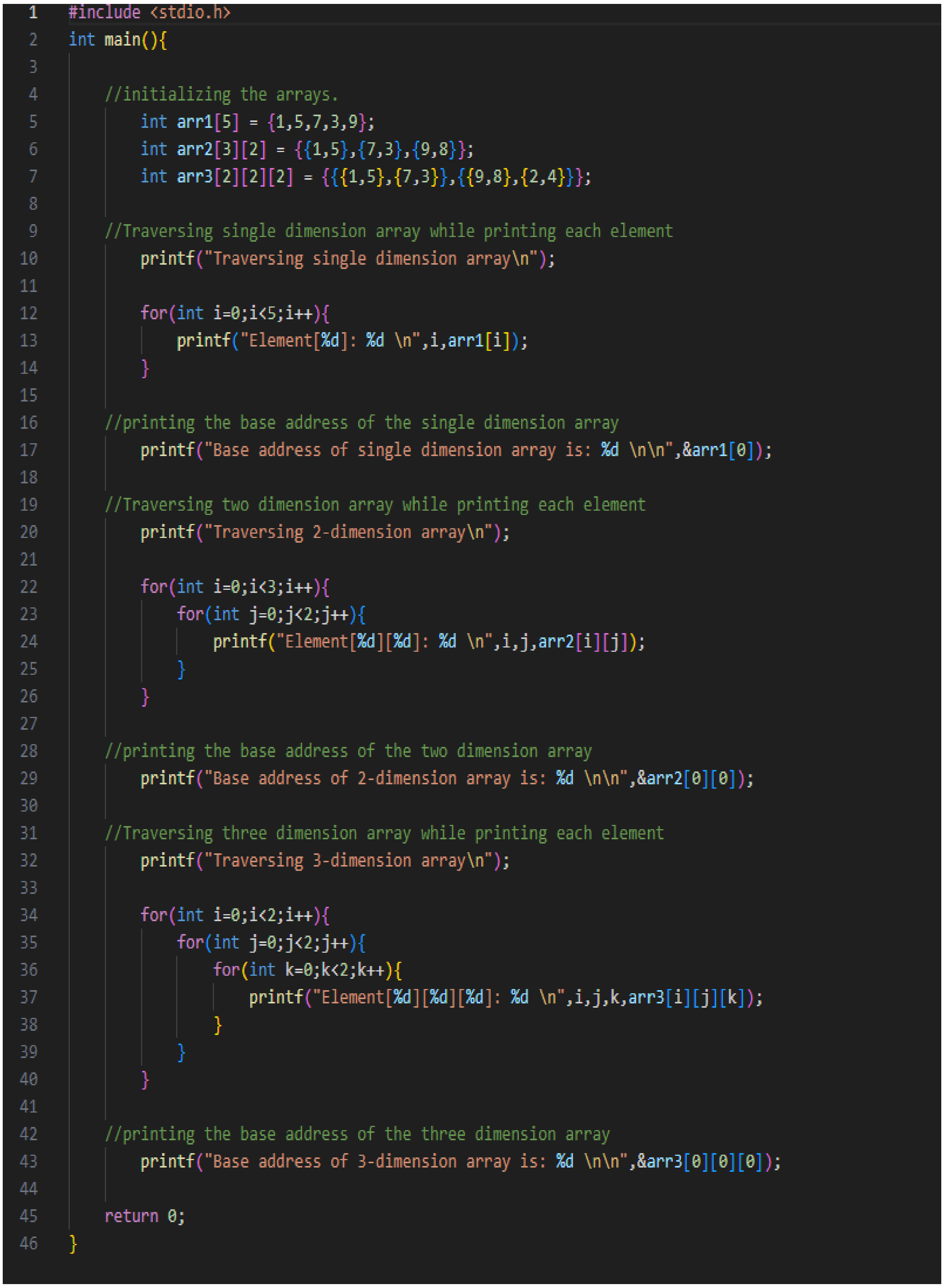
b) Traversal of a single dimension, 2-dimension, and 3-dimension arrays and determine the base address of each array.

-


C: > Users > USER > AppData > Local > Microsoft > Windows > INetCache > IE > Y2N7F2FH > BaseAddress[1].java > BaseAddress > main(String[])

```java
public class BaseAddress{
    public static void main(String[] args) {
        //initializing the arrays.
            int[] arr1 = {1,5,7,3,9};
            int[][] arr2 = {{1,5},{7,3},{9,8}};
            int[][][] arr3 = {{{1,5},{7,3}},{{9,8},{2,4}}};

        //retrieving base addresses of the arrays
            int baseaddress1=System.identityHashCode(arr1);
            int baseaddress2=System.identityHashCode(arr2);
            int baseaddress3=System.identityHashCode(arr3);

        //Traversing single dimension array while printing each element
            System.out.println("Traversing single dimension array");

            for(int i=0;i<5;i++){
                System.out.println("Element["+ i +"]: "+ arr1[i]);
            }

        //printing the base address of the single dimension array
            System.out.println("Base address of single dimension array is: "+Integer.toHexString(baseaddress1));
            System.out.println();

        //Traversing two dimension array while printing each element
            System.out.println("Traversing 2-dimension array");

            for(int i=0;i<3;i++){
                for(int j=0;j<2;j++){
                    System.out.println("Element["+ i +"][" + j + "]: " + arr2[i][j]);
                }
            }
        //printing the base address of the two dimension array
            System.out.println("Base address of 2-dimension array is: "+Integer.toHexString(baseaddress2));
            System.out.println();
        //Traversing three dimension array while printing each element
            System.out.println("Traversing 3-dimension array");

            for(int i=0;i<2;i++){
                for(int j=0;j<2;j++){
                    for(int k=0;k<2;k++){
                        System.out.println("Element["+ i +"]["+ j +"][" + k + "]: " + arr3[i][j][k]);
                    }
                }
            }
        //printing the base address of the three dimension array
            System.out.println("Base address of 3-dimension array is:"+Integer.toHexString(baseaddress3));
            System.out.println();
    }
}
```

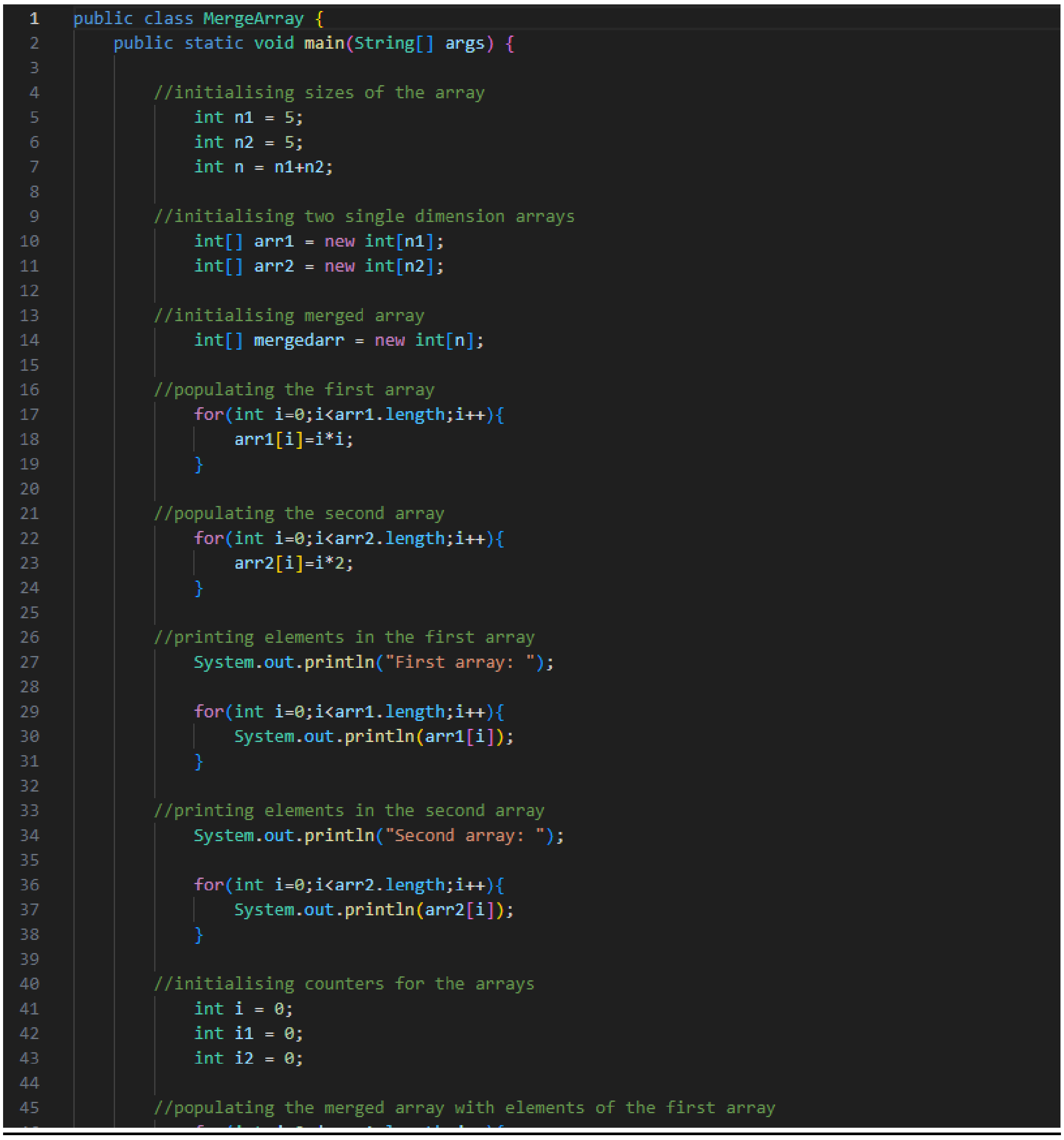
Figure 1: showing Java code for Base Address



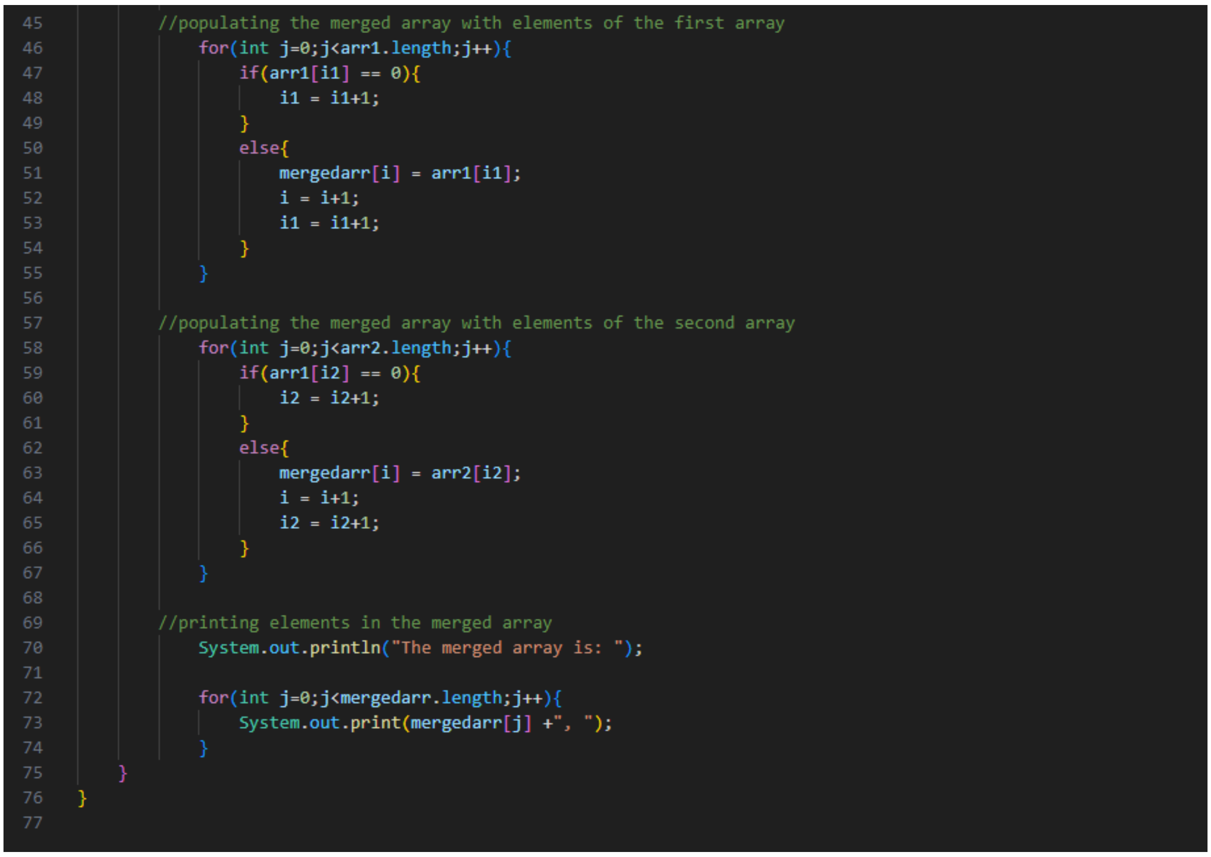
```c
#include <stdio.h>
int main(){

    //initializing the arrays.
        int arr1[5] = {1,5,7,3,9};
        int arr2[3][2] = {{1,5},{7,3},{9,8}};
        int arr3[2][2][2] = {{{1,5},{7,3}},{{9,8},{2,4}}};

    //Traversing single dimension array while printing each element
        printf("Traversing single dimension array\n");

        for(int i=0;i<5;i++){
            printf("Element[%d]: %d \n",i,arr1[i]);
        }

    //printing the base address of the single dimension array
        printf("Base address of single dimension array is: %d \n\n",&arr1[0]);

    //Traversing two dimension array while printing each element
        printf("Traversing 2-dimension array\n");

        for(int i=0;i<3;i++){
            for(int j=0;j<2;j++){
                printf("Element[%d][%d]: %d \n",i,j,arr2[i][j]);
            }
        }

    //printing the base address of the two dimension array
        printf("Base address of 2-dimension array is: %d \n\n",&arr2[0][0]);

    //Traversing three dimension array while printing each element
        printf("Traversing 3-dimension array\n");

        for(int i=0;i<2;i++){
            for(int j=0;j<2;j++){
                for(int k=0;k<2;k++){
                    printf("Element[%d][%d][%d]: %d \n",i,j,k,arr3[i][j][k]);
                }
            }
        }

    //printing the base address of the three dimension array
        printf("Base address of 3-dimension array is: %d \n\n",&arr3[0][0][0]);

    return 0;
}
```

Figure 2: showing C code for Base Address

c) Create two single-dimension arrays and merge them as a single array.

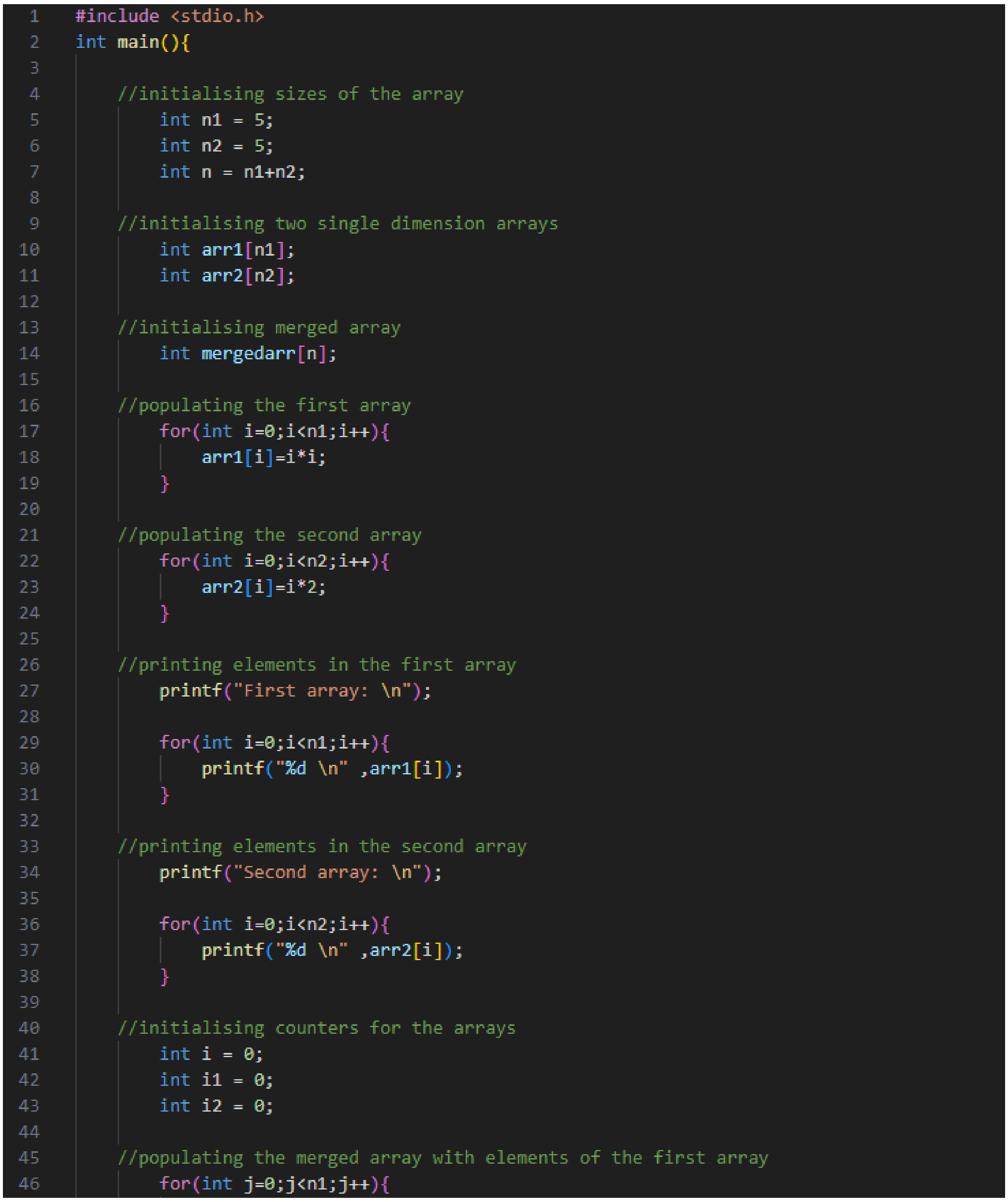


```java
public class MergeArray {
    public static void main(String[] args) {

        //initialising sizes of the array
            int n1 = 5;
            int n2 = 5;
            int n = n1+n2;

        //initialising two single dimension arrays
            int[] arr1 = new int[n1];
            int[] arr2 = new int[n2];

        //initialising merged array
            int[] mergedarr = new int[n];

        //populating the first array
            for(int i=0;i<arr1.length;i++){
                arr1[i]=i*i;
            }

        //populating the second array
            for(int i=0;i<arr2.length;i++){
                arr2[i]=i*2;
            }

        //printing elements in the first array
            System.out.println("First array: ");

            for(int i=0;i<arr1.length;i++){
                System.out.println(arr1[i]);
            }

        //printing elements in the second array
            System.out.println("Second array: ");

            for(int i=0;i<arr2.length;i++){
                System.out.println(arr2[i]);
            }

        //initialising counters for the arrays
            int i = 0;
            int i1 = 0;
            int i2 = 0;

        //populating the merged array with elements of the first array
```

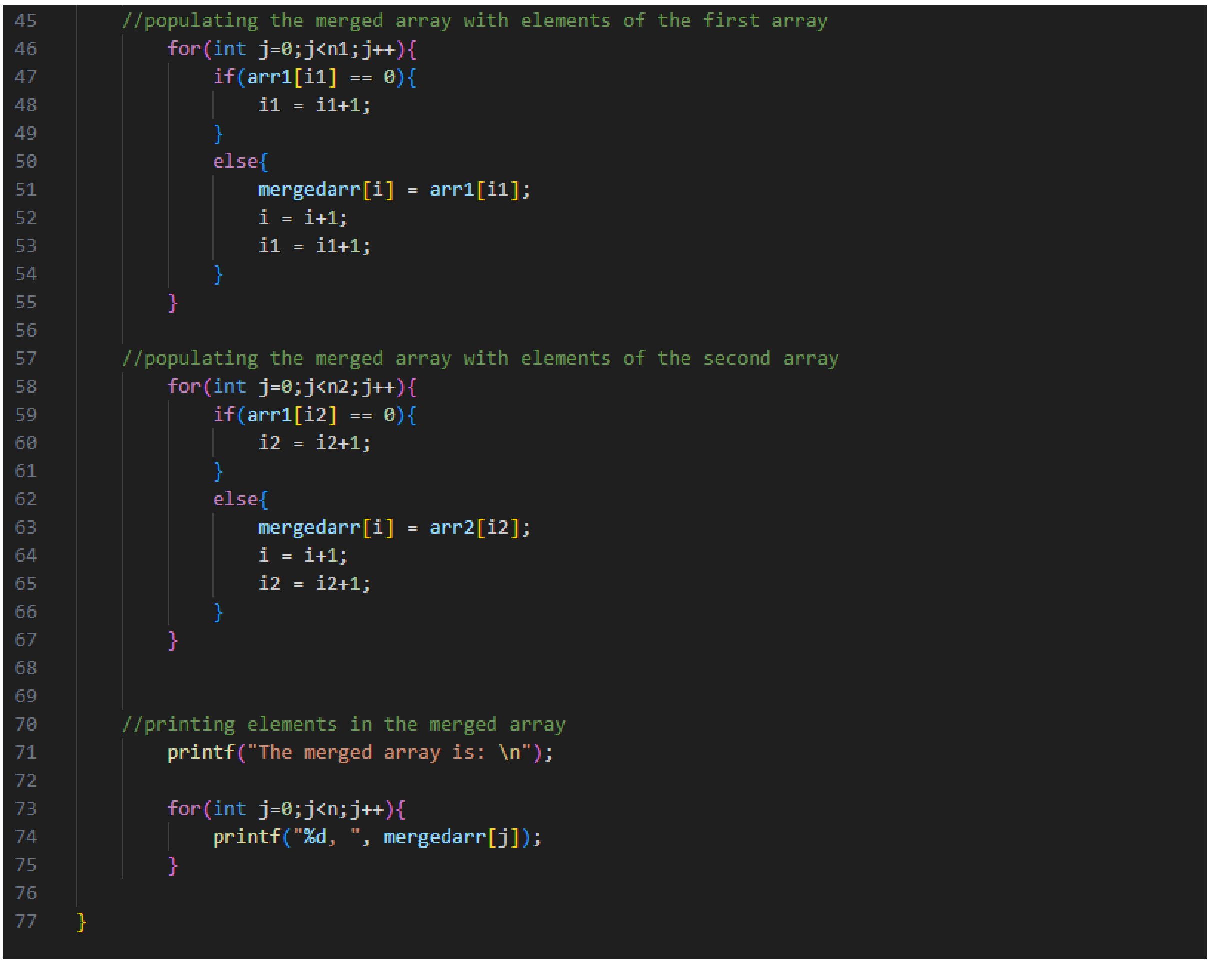


```java
        //populating the merged array with elements of the first array
            for(int j=0;j<arr1.length;j++){
                if(arr1[i1] == 0){
                    i1 = i1+1;
                }
                else{
                    mergedarr[i] = arr1[i1];
                    i = i+1;
                    i1 = i1+1;
                }
            }

        //populating the merged array with elements of the second array
            for(int j=0;j<arr2.length;j++){
                if(arr1[i2] == 0){
                    i2 = i2+1;
                }
                else{
                    mergedarr[i] = arr2[i2];
                    i = i+1;
                    i2 = i2+1;
                }
            }

        //printing elements in the merged array
            System.out.println("The merged array is: ");

            for(int j=0;j<mergedarr.length;j++){
                System.out.print(mergedarr[j] +", ");
            }
    }
}
```

Figure 3: showing Java code for merging two single-dimension arrays



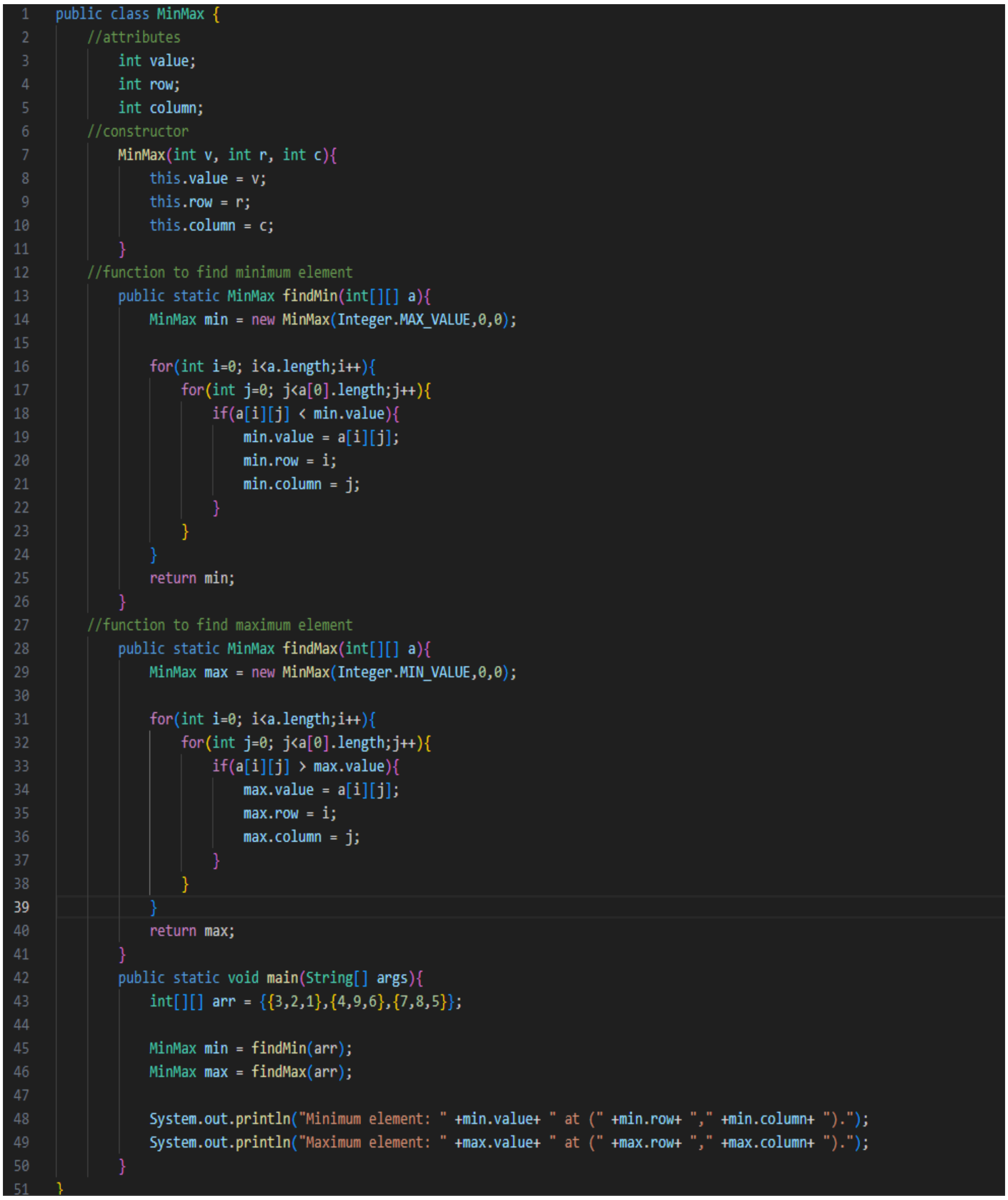
```c
#include <stdio.h>
int main(){

    //initialising sizes of the array
        int n1 = 5;
        int n2 = 5;
        int n = n1+n2;

    //initialising two single dimension arrays
        int arr1[n1];
        int arr2[n2];

    //initialising merged array
        int mergedarr[n];

    //populating the first array
        for(int i=0;i<n1;i++){
            arr1[i]=i*i;
        }

    //populating the second array
        for(int i=0;i<n2;i++){
            arr2[i]=i*2;
        }

    //printing elements in the first array
        printf("First array: \n");

        for(int i=0;i<n1;i++){
            printf("%d \n" ,arr1[i]);
        }

    //printing elements in the second array
        printf("Second array: \n");

        for(int i=0;i<n2;i++){
            printf("%d \n" ,arr2[i]);
        }

    //initialising counters for the arrays
        int i = 0;
        int i1 = 0;
        int i2 = 0;

    //populating the merged array with elements of the first array
        for(int j=0;j<n1;j++){
```



```c
    //populating the merged array with elements of the first array
        for(int j=0;j<n1;j++){
            if(arr1[i1] == 0){
                i1 = i1+1;
            }
            else{
                mergedarr[i] = arr1[i1];
                i = i+1;
                i1 = i1+1;
            }
        }

    //populating the merged array with elements of the second array
        for(int j=0;j<n2;j++){
            if(arr1[i2] == 0){
                i2 = i2+1;
            }
            else{
                mergedarr[i] = arr2[i2];
                i = i+1;
                i2 = i2+1;
            }
        }


    //printing elements in the merged array
        printf("The merged array is: \n");

        for(int j=0;j<n;j++){
            printf("%d, ", mergedarr[j]);
        }

}
```

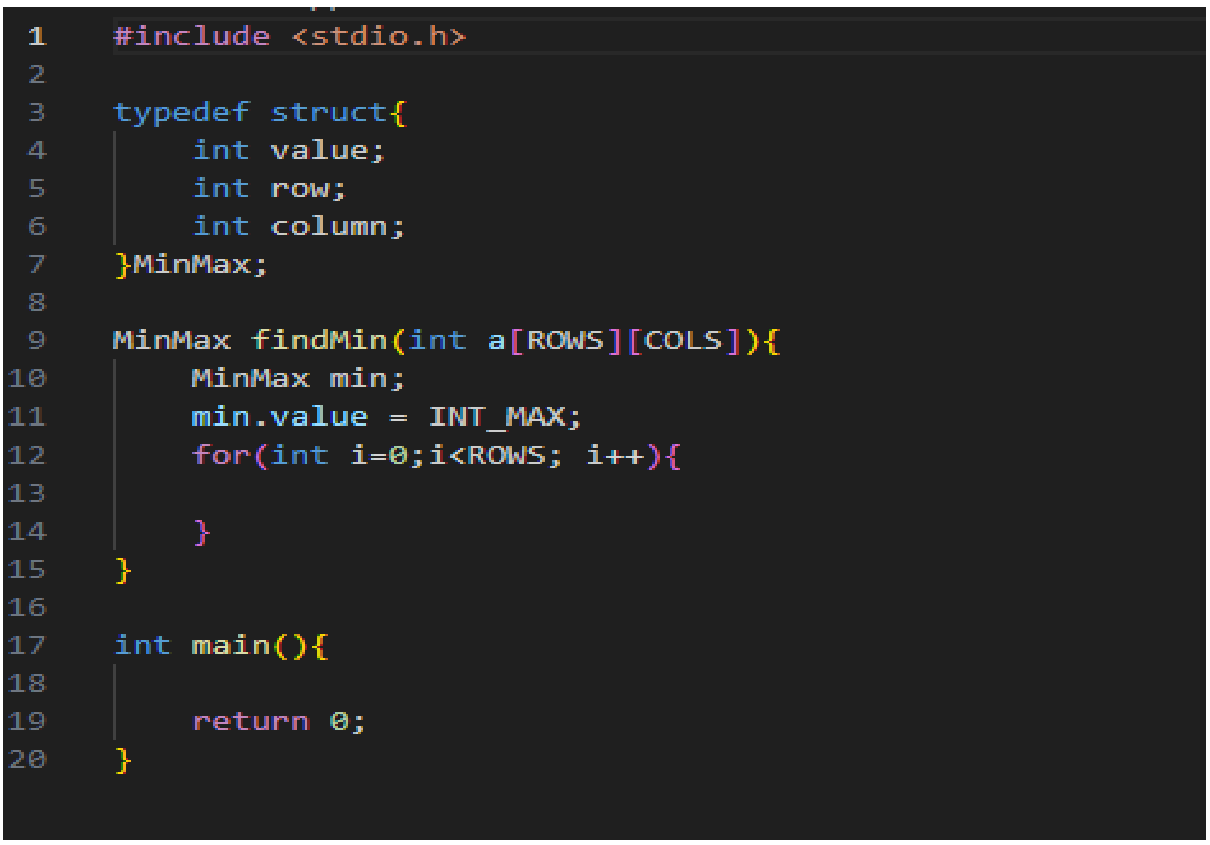
Figure 4: showing C code for merging two single-dimension arrays

d) Create a two-dimensional array and design two separate functions that return the minimum and maximum element with their locations.



```java
public class MinMax {
    //attributes
        int value;
        int row;
        int column;
    //constructor
        MinMax(int v, int r, int c){
            this.value = v;
            this.row = r;
            this.column = c;
        }
    //function to find minimum element
        public static MinMax findMin(int[][] a){
            MinMax min = new MinMax(Integer.MAX_VALUE,0,0);

            for(int i=0; i<a.length;i++){
                for(int j=0; j<a[0].length;j++){
                    if(a[i][j] < min.value){
                        min.value = a[i][j];
                        min.row = i;
                        min.column = j;
                    }
                }
            }
            return min;
        }
    //function to find maximum element
        public static MinMax findMax(int[][] a){
            MinMax max = new MinMax(Integer.MIN_VALUE,0,0);

            for(int i=0; i<a.length;i++){
                for(int j=0; j<a[0].length;j++){
                    if(a[i][j] > max.value){
                        max.value = a[i][j];
                        max.row = i;
                        max.column = j;
                    }
                }
            }
            return max;
        }
        public static void main(String[] args){
            int[][] arr = {{3,2,1},{4,9,6},{7,8,5}};

            MinMax min = findMin(arr);
            MinMax max = findMax(arr);

            System.out.println("Minimum element: " +min.value+ " at (" +min.row+ "," +min.column+ ").");
            System.out.println("Maximum element: " +max.value+ " at (" +max.row+ "," +max.column+ ").");
        }
}
```

Figure 5: showing Java code for returning a maximum and minimum elements with their locations

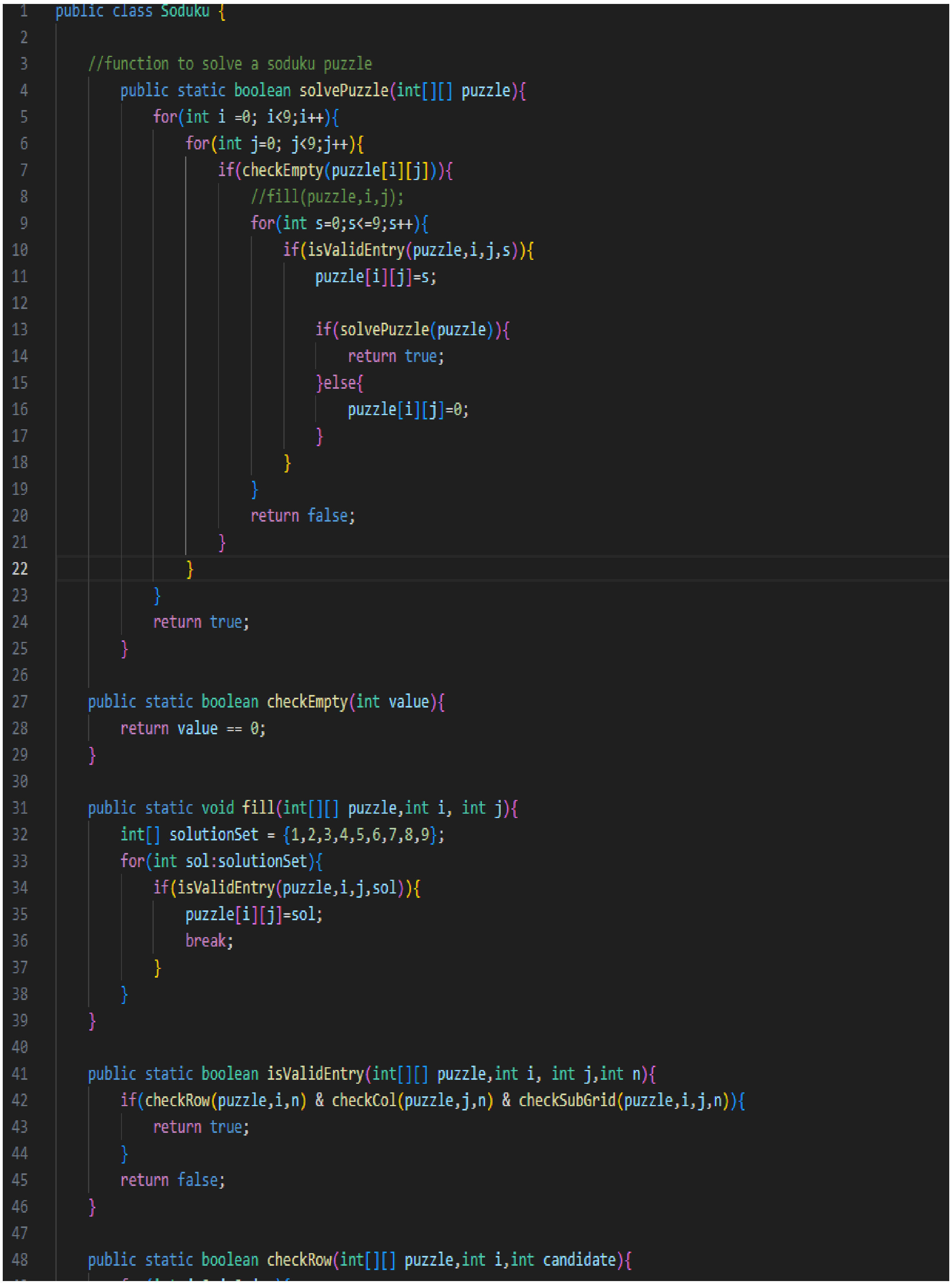


```c
#include <stdio.h>

typedef struct{
    int value;
    int row;
    int column;
}MinMax;

MinMax findMin(int a[ROWS][COLS]){
    MinMax min;
    min.value = INT_MAX;
    for(int i=0;i<ROWS; i++){

    }
}

int main(){

    return 0;
}
```

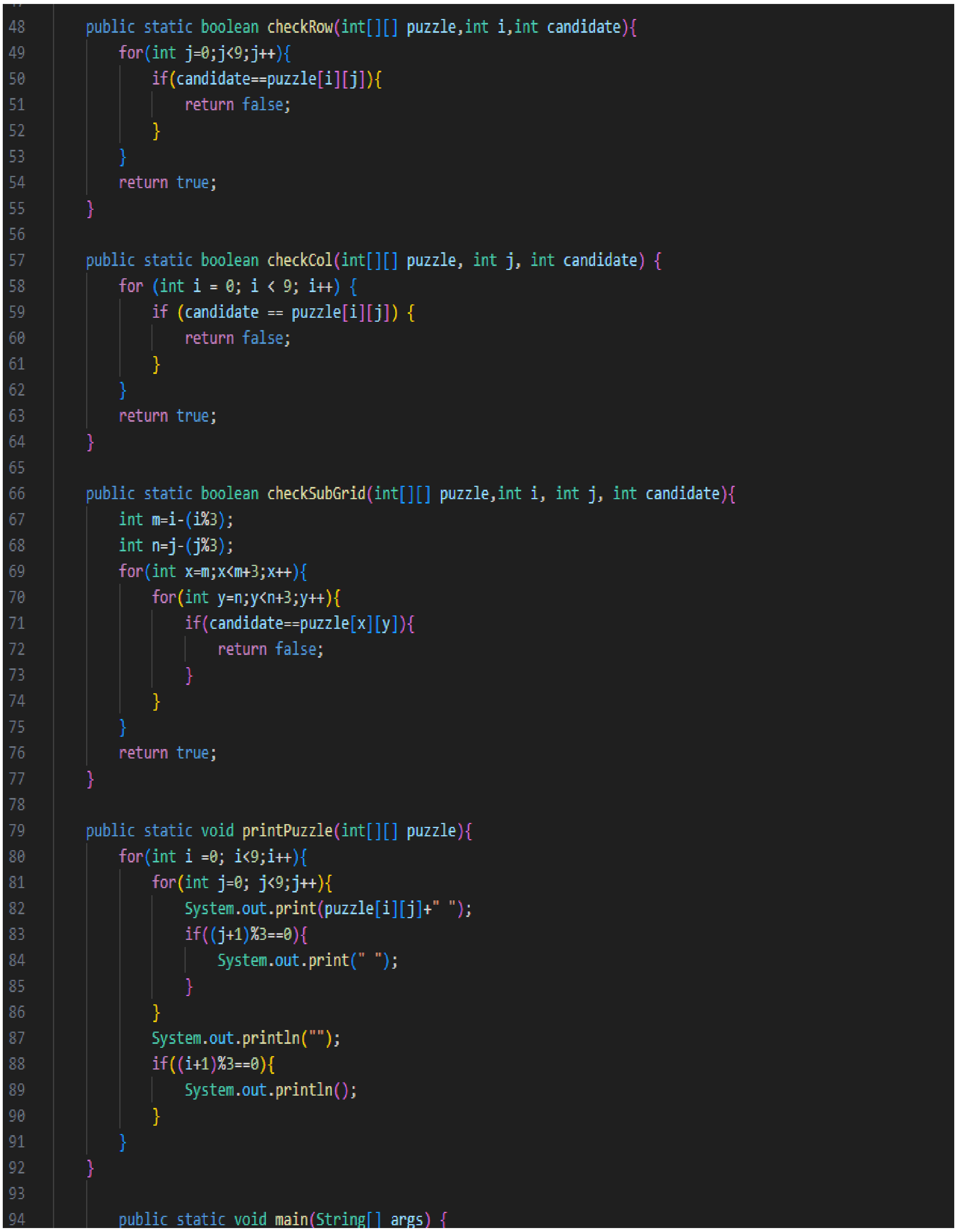
Figure 6: showing C code for returning the maximum and minimum elements with their positions

e) **Use a 2D array to represent a 9x9 Sudoku grid. Some cells will be pre-filled with numbers, while others will be empty (represented by 0).**

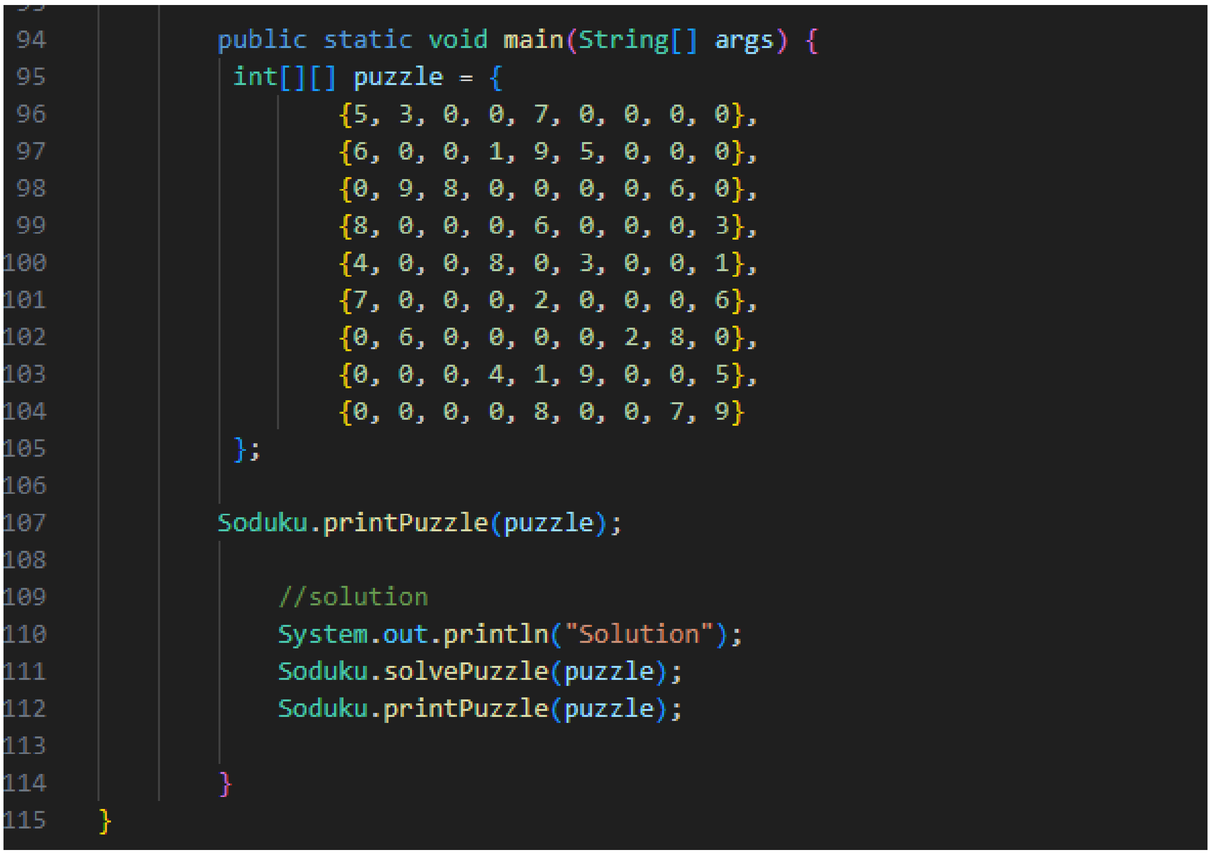
f) **Write a function to check whether a number can be placed in a specific cell without violating Sudoku rules (each number must be unique in its row, column, and 3x3 sub-grid).**



```java
public class Soduku {

    //function to solve a soduku puzzle
        public static boolean solvePuzzle(int[][] puzzle){
            for(int i =0; i<9;i++){
                for(int j=0; j<9;j++){
                    if(checkEmpty(puzzle[i][j])){
                        //fill(puzzle,i,j);
                        for(int s=0;s<=9;s++){
                            if(isValidEntry(puzzle,i,j,s)){
                                puzzle[i][j]=s;

                                if(solvePuzzle(puzzle)){
                                    return true;
                                }else{
                                    puzzle[i][j]=0;
                                }
                            }
                        }
                        return false;
                    }
                }
            }
            return true;
        }

    public static boolean checkEmpty(int value){
        return value == 0;
    }

    public static void fill(int[][] puzzle,int i, int j){
        int[] solutionSet = {1,2,3,4,5,6,7,8,9};
        for(int sol:solutionSet){
            if(isValidEntry(puzzle,i,j,sol)){
                puzzle[i][j]=sol;
                break;
            }
        }
    }

    public static boolean isValidEntry(int[][] puzzle,int i, int j,int n){
        if(checkRow(puzzle,i,n) & checkCol(puzzle,j,n) & checkSubGrid(puzzle,i,j,n)){
            return true;
        }
        return false;
    }

    public static boolean checkRow(int[][] puzzle,int i,int candidate){
```

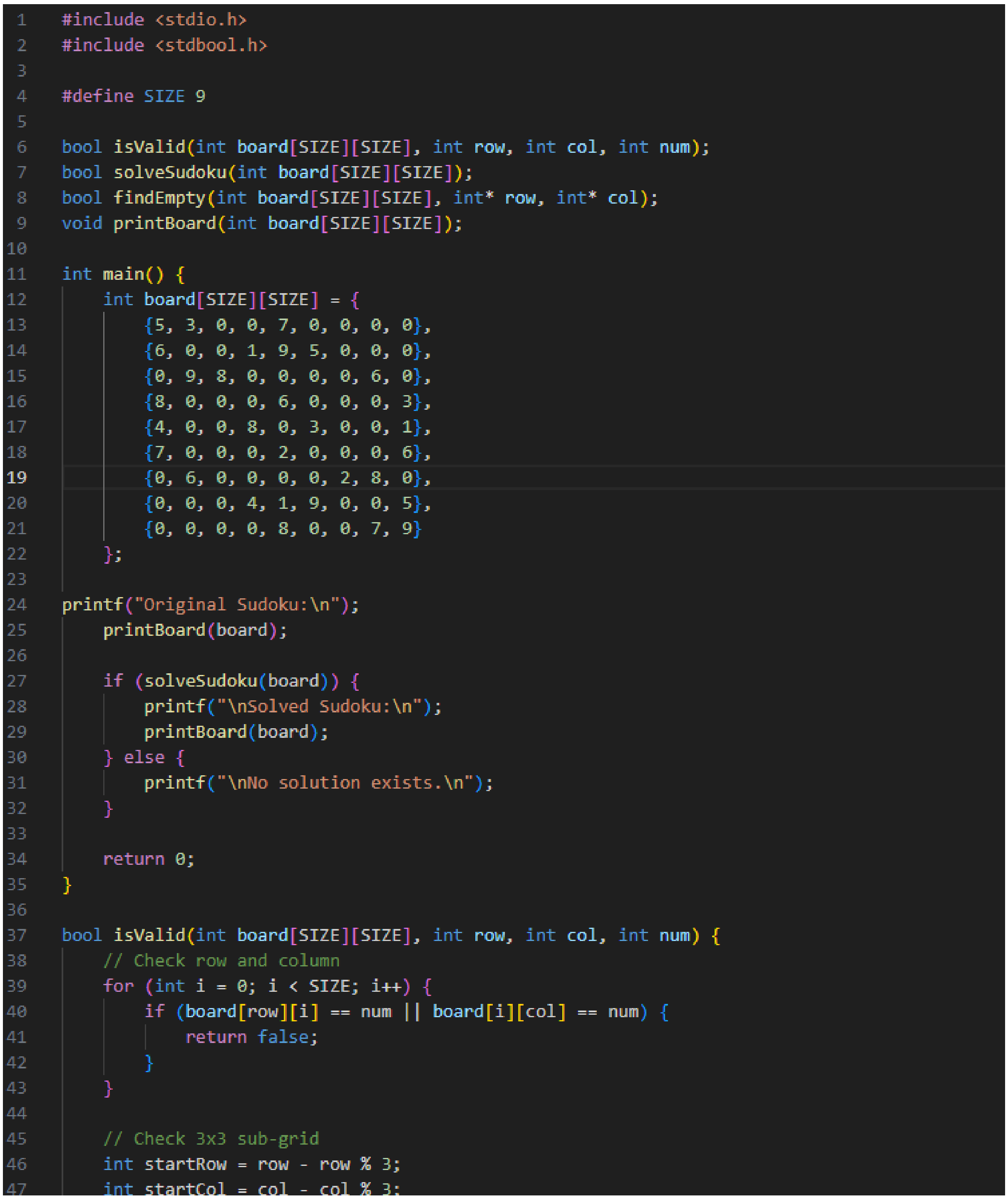


```
    public static boolean checkRow(int[][] puzzle,int i,int candidate){
        for(int j=0;j<9;j++){
            if(candidate==puzzle[i][j]){
                return false;
            }
        }
        return true;
    }

    public static boolean checkCol(int[][] puzzle, int j, int candidate) {
        for (int i = 0; i < 9; i++) {
            if (candidate == puzzle[i][j]) {
                return false;
            }
        }
        return true;
    }

    public static boolean checkSubGrid(int[][] puzzle,int i, int j, int candidate){
        int m=i-(i%3);
        int n=j-(j%3);
        for(int x=m;x<m+3;x++){
            for(int y=n;y<n+3;y++){
                if(candidate==puzzle[x][y]){
                    return false;
                }
            }
        }
        return true;
    }

    public static void printPuzzle(int[][] puzzle){
        for(int i =0; i<9;i++){
            for(int j=0; j<9;j++){
                System.out.print(puzzle[i][j]+" ");
                if((j+1)%3==0){
                    System.out.print(" ");
                }
            }
            System.out.println("");
            if((i+1)%3==0){
                System.out.println();
            }
        }
    }

        public static void main(String[] args) {
```

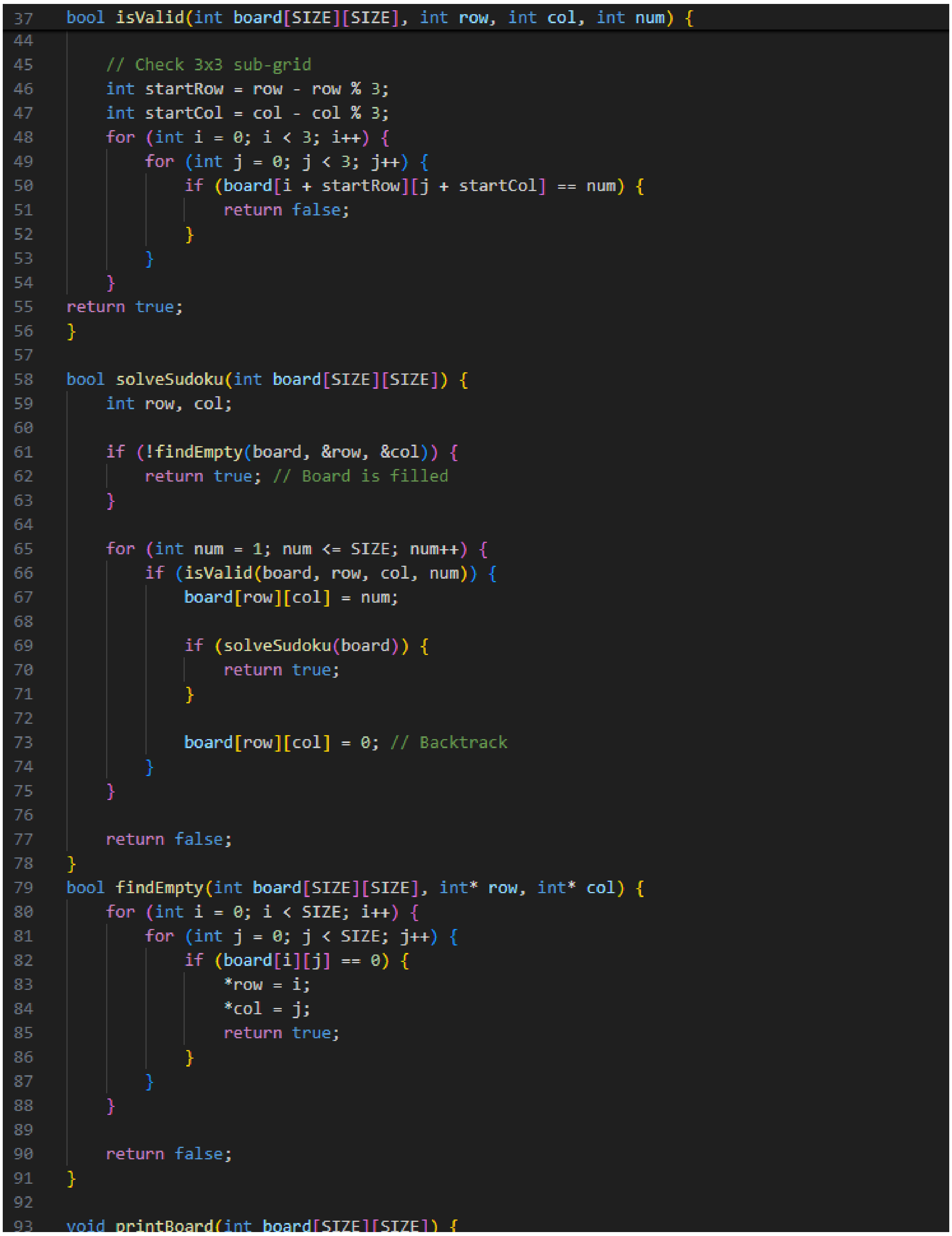


```java
        public static void main(String[] args) {
         int[][] puzzle = {
                {5, 3, 0, 0, 7, 0, 0, 0, 0},
                {6, 0, 0, 1, 9, 5, 0, 0, 0},
                {0, 9, 8, 0, 0, 0, 0, 6, 0},
                {8, 0, 0, 0, 6, 0, 0, 0, 3},
                {4, 0, 0, 8, 0, 3, 0, 0, 1},
                {7, 0, 0, 0, 2, 0, 0, 0, 6},
                {0, 6, 0, 0, 0, 0, 2, 8, 0},
                {0, 0, 0, 4, 1, 9, 0, 0, 5},
                {0, 0, 0, 0, 8, 0, 0, 7, 9}
         };

        Soduku.printPuzzle(puzzle);

            //solution
            System.out.println("Solution");
            Soduku.solvePuzzle(puzzle);
            Soduku.printPuzzle(puzzle);

        }
}
```

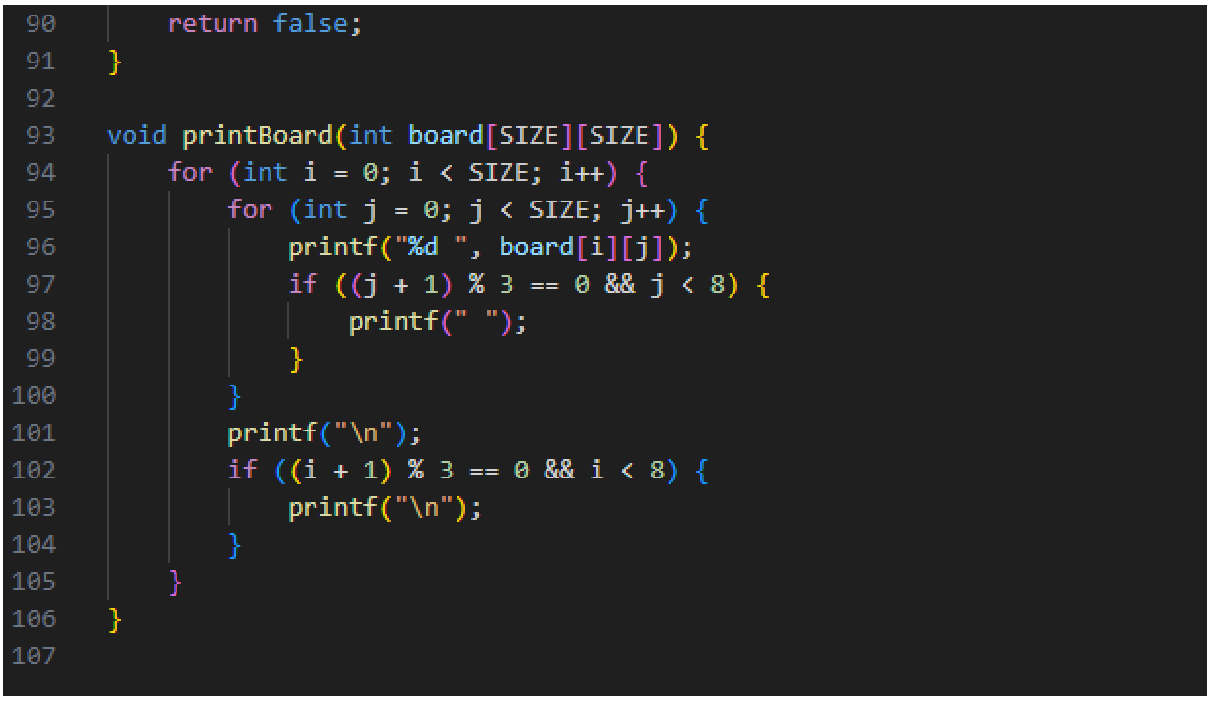
Figure 7: showing Java code solving a 9x9 Sudoku grid



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

#define SIZE 9

bool isValid(int board[SIZE][SIZE], int row, int col, int num);
bool solveSudoku(int board[SIZE][SIZE]);
bool findEmpty(int board[SIZE][SIZE], int* row, int* col);
void printBoard(int board[SIZE][SIZE]);

int main() {
    int board[SIZE][SIZE] = {
        {5, 3, 0, 0, 7, 0, 0, 0, 0},
        {6, 0, 0, 1, 9, 5, 0, 0, 0},
        {0, 9, 8, 0, 0, 0, 0, 6, 0},
        {8, 0, 0, 0, 6, 0, 0, 0, 3},
        {4, 0, 0, 8, 0, 3, 0, 0, 1},
        {7, 0, 0, 0, 2, 0, 0, 0, 6},
        {0, 6, 0, 0, 0, 0, 2, 8, 0},
        {0, 0, 0, 4, 1, 9, 0, 0, 5},
        {0, 0, 0, 0, 8, 0, 0, 7, 9}
    };

printf("Original Sudoku:\n");
    printBoard(board);

    if (solveSudoku(board)) {
        printf("\nSolved Sudoku:\n");
        printBoard(board);
    } else {
        printf("\nNo solution exists.\n");
    }

    return 0;
}

bool isValid(int board[SIZE][SIZE], int row, int col, int num) {
    // Check row and column
    for (int i = 0; i < SIZE; i++) {
        if (board[row][i] == num || board[i][col] == num) {
            return false;
        }
    }

    // Check 3x3 sub-grid
    int startRow = row - row % 3;
    int startCol = col - col % 3;
```



```c
bool isValid(int board[SIZE][SIZE], int row, int col, int num) {

    // Check 3x3 sub-grid
    int startRow = row - row % 3;
    int startCol = col - col % 3;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[i + startRow][j + startCol] == num) {
                return false;
            }
        }
    }
return true;
}

bool solveSudoku(int board[SIZE][SIZE]) {
    int row, col;

    if (!findEmpty(board, &row, &col)) {
        return true; // Board is filled
    }

    for (int num = 1; num <= SIZE; num++) {
        if (isValid(board, row, col, num)) {
            board[row][col] = num;

            if (solveSudoku(board)) {
                return true;
            }

            board[row][col] = 0; // Backtrack
        }
    }

    return false;
}
bool findEmpty(int board[SIZE][SIZE], int* row, int* col) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (board[i][j] == 0) {
                *row = i;
                *col = j;
                return true;
            }
        }
    }

    return false;
}

void printBoard(int board[SIZE][SIZE]) {
```



```c
    return false;
}

void printBoard(int board[SIZE][SIZE]) {
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            printf("%d ", board[i][j]);
            if ((j + 1) % 3 == 0 && j < 8) {
                printf(" ");
            }
        }
        printf("\n");
        if ((i + 1) % 3 == 0 && i < 8) {
            printf("\n");
        }
    }
}

```

Figure 8: showing C code for solving a 9x9 Sudoku grid

<u>PREFERRED CHOICE OF CODE AND WHY???</u>

1. **<u>Traversal of a single-dimension, 2-dimension, and 3-dimension arrays and determine the base address of each array</u>**

The preferred choice of code: C

Why??: Easier access to base addresses, unlike in Java where the identityHashCode() method must be used.

2. **<u>Create two single dimension arrays and merge them as a single array</u>**

Preferred choice of code: Java

Why??: There isn't much of a difference in this instance between the C and Java programs but what gives Java an advantage is the ability to use array.length attribute that simplifies modification.

3. **<u>Create a two-dimension array and design two separate functions that return the minimum element and maximum element with their locations</u>**

Preferred choice of code: Java

Why??: Easy to implement since it's an Object-Oriented Language, unlike C where structs had to be utilized to create an object with the required attributes.

4. **How do row, column, and sub-grid constraints map to array indices?**

Row Constraint

- Row index i corresponds to board[i][0-8]

- **Example:** board[0][0-8] represents the first row

Column Constraint

- Column index j corresponds to board[0-8][j]

- **Example:** board [0-8][0] represents the first column

Sub-Grid Constraint

- Sub-grid index (i, j) corresponds to board[3*(i/3) + (i%3)][3*(j/3) + (j%3)]

Example:

- Top-left sub-grid: board[0-2][0-2]
- Top-right sub-grid: board[0-2][6-8]
- Bottom-left sub-grid: board[6-8][0-2]
- Bottom-right sub-grid: board[6-8][6-8]

TASK 3: LINKED LISTS

a) In a list of multiple occurrences of the element KEY, make sure that there remains only a single occurrence of the element KEY in the list.

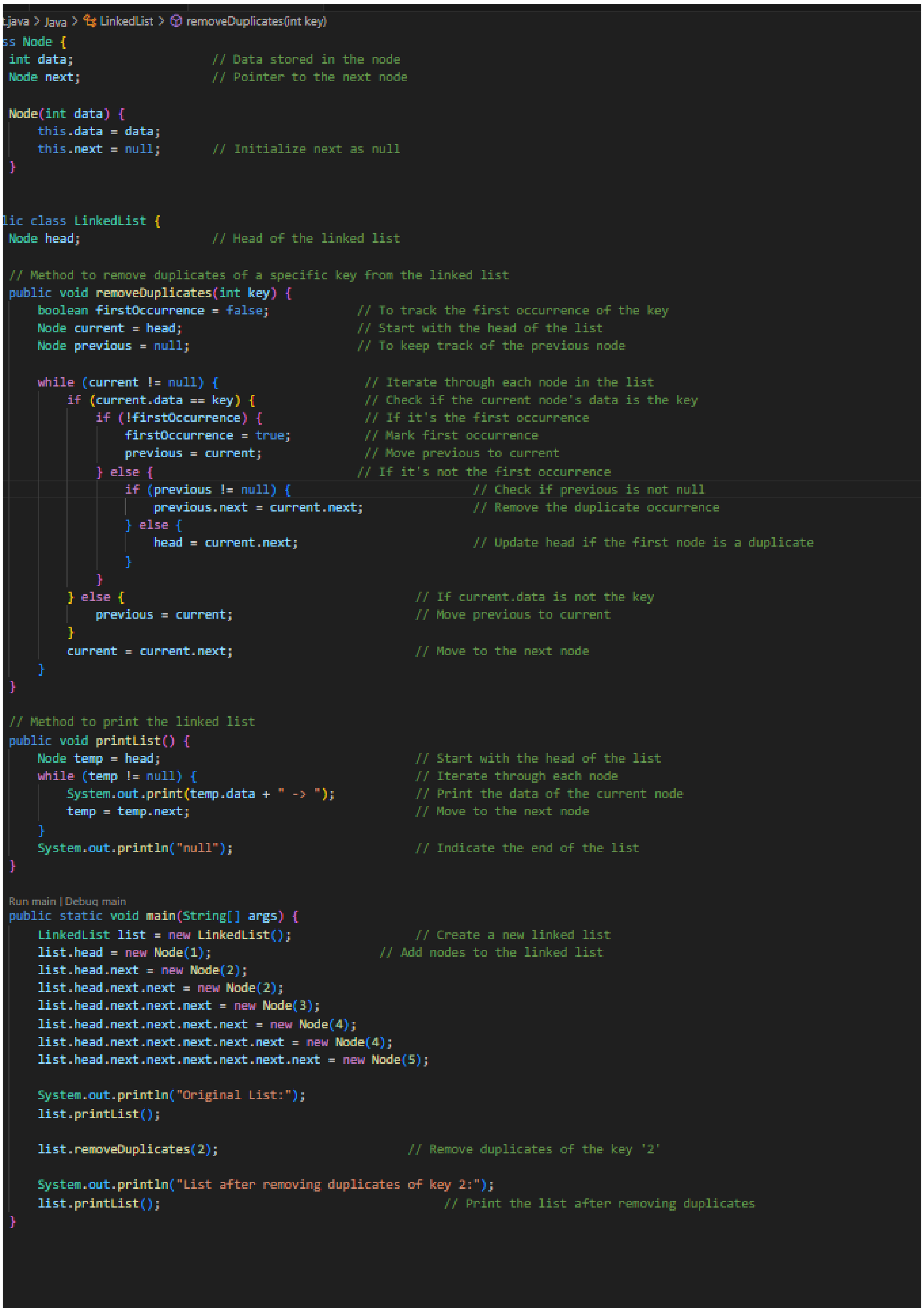
Pseudocode

```
1. Initialize a boolean variable firstOccurrence as False          // To track the first occurrence of KEY
2. Initialize a Node variable current as head                      // Start with the head of the list
3. Initialize a Node variable previous as null                     // To keep track of the previous node
4. While current is not null                                       // Iterate through each node in the list
    1. If current. data equals KEY                                 // Check if the current node's data is KEY
    2. If firstOccurrence is False                                 // If it's the first occurrence
        1. Set firstOccurrence to True                             // Mark first occurrence
        2. Set previous to current                                 // Move previous to current
        3. Else                                                    // If it's not the first occurrence
        4. If previous is not null                                 // Check if previous is not null
            1. Set previous. Next to current. Next                 // Remove the duplicate occurrence
        5. Else                                                    // If the previous node is null
            1. Set head to current. next                           // Update head if the first node is a duplicate
        6. Else                                                    // If current. data is not KEY
            1. Set previous to current                             // Move previous to the current node
            2. Set current to current. next            // Move to the
               next node
        7. End While
        8. End Method
```

Method: RemoveDuplicates (HEADER, KEY)

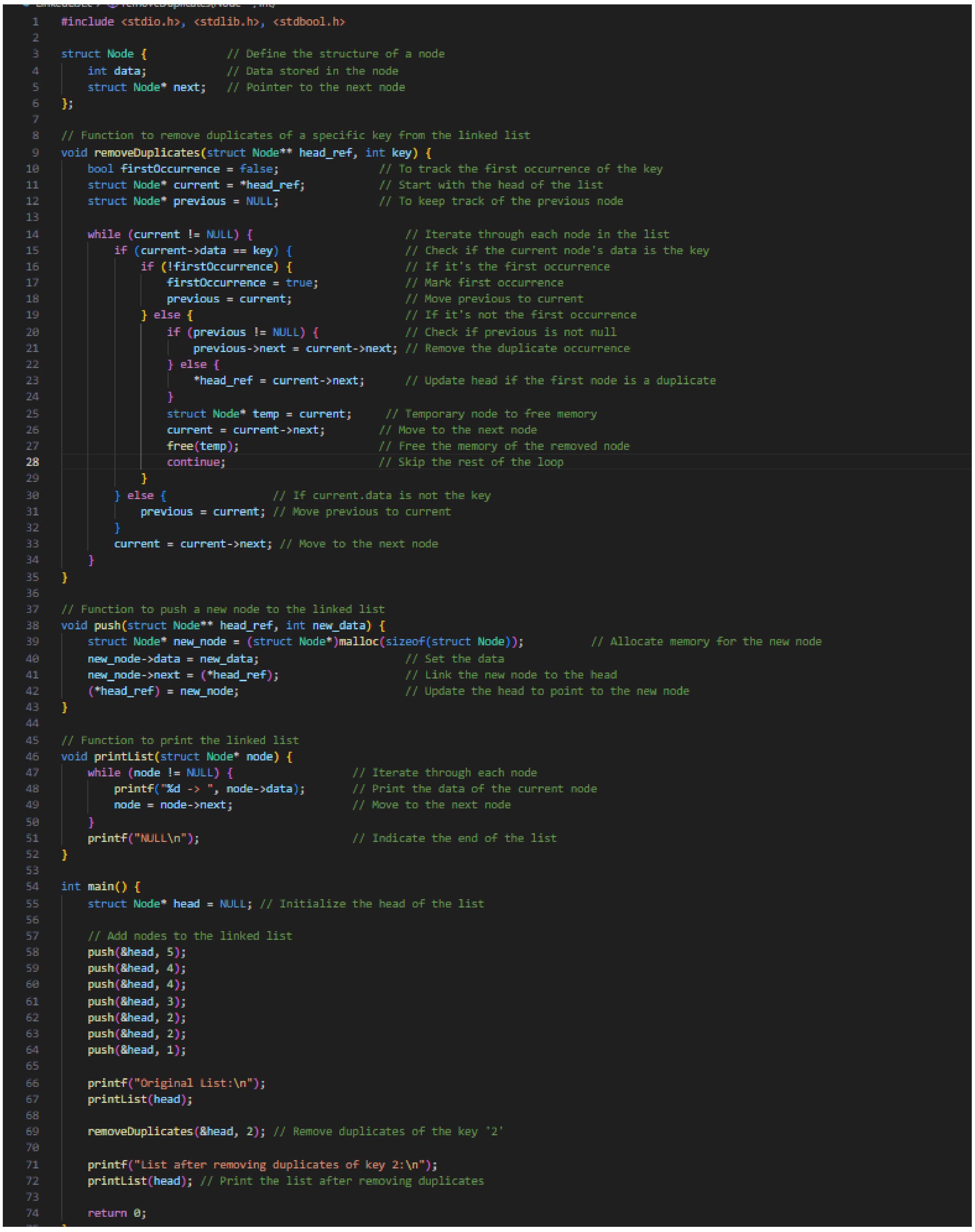
```
{
  Ptr = HEADER                          // Ptr is initialized to pointer to the current node
  Previous = NULL                       // Previous is initialized to NULL
  FirstOccurrence = False               // To track the first occurrence of KEY
  While (Ptr != NULL)                   // Continue until the last node
  {
    If (Ptr ->data == KEY)              // Check if the current node's data is KEY
    {
      If (FirstOccurrence == False)     // If it's the first occurrence
      {
        FirstOccurrence = True          // Mark first occurrence
        Previous = Ptr                  // Move previous to current
      }
      Else {                            // If it's not the first occurrence
       If (Previous != NULL)            // Check if previous is not NULL
        {
          Previous->next = Ptr->next    // Remove the duplicate occurrence
        }
        Else {                          // If previous is NULL
          HEADER = Ptr->next            // Update HEADER if the first node is a duplicate
        }
      }
```

```
        }
        Else                                                    // If Ptr->data is not KEY
        {
          Previous = Ptr                                        // Move previous to current
        }
        Ptr = Ptr->next                                         // Move to the next node
    }
}
```



```
java > Java > LinkedList > removeDuplicates(int key)
ss Node {
 int data;                   // Data stored in the node
 Node next;                  // Pointer to the next node

 Node(int data) {
     this.data = data;
     this.next = null;       // Initialize next as null
 }


lic class LinkedList {
 Node head;                  // Head of the linked list

 // Method to remove duplicates of a specific key from the linked list
 public void removeDuplicates(int key) {
     boolean firstOccurrence = false;           // To track the first occurrence of the key
     Node current = head;                       // Start with the head of the list
     Node previous = null;                      // To keep track of the previous node

     while (current != null) {                   // Iterate through each node in the list
         if (current.data == key) {              // Check if the current node's data is the key
             if (!firstOccurrence) {             // If it's the first occurrence
                 firstOccurrence = true;         // Mark first occurrence
                 previous = current;             // Move previous to current
             } else {                           // If it's not the first occurrence
                 if (previous != null) {                        // Check if previous is not null
                     previous.next = current.next;              // Remove the duplicate occurrence
                 } else {
                     head = current.next;                       // Update head if the first node is a duplicate
                 }
             }
         } else {                                   // If current.data is not the key
             previous = current;                    // Move previous to current
         }
         current = current.next;                    // Move to the next node
     }
 }

 // Method to print the linked list
 public void printList() {
     Node temp = head;                              // Start with the head of the list
     while (temp != null) {                         // Iterate through each node
         System.out.print(temp.data + " -> ");      // Print the data of the current node
         temp = temp.next;                          // Move to the next node
     }
     System.out.println("null");                    // Indicate the end of the list
 }

 Run main | Debug main
 public static void main(String[] args) {
     LinkedList list = new LinkedList();            // Create a new linked list
     list.head = new Node(1);                  // Add nodes to the linked list
     list.head.next = new Node(2);
     list.head.next.next = new Node(2);
     list.head.next.next.next = new Node(3);
     list.head.next.next.next.next = new Node(4);
     list.head.next.next.next.next.next = new Node(4);
     list.head.next.next.next.next.next.next = new Node(5);

     System.out.println("Original List:");
     list.printList();

     list.removeDuplicates(2);                     // Remove duplicates of the key '2'

     System.out.println("List after removing duplicates of key 2:");
     list.printList();                                  // Print the list after removing duplicates
 }
```

Figure 9: Java code showing implementation of linked lists specifically how to remove multiple occurrences of an element KEY and remain with only a single occurrence of the element KEY in a list.



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

struct Node {            // Define the structure of a node
    int data;            // Data stored in the node
    struct Node* next;   // Pointer to the next node
};

// Function to remove duplicates of a specific key from the linked list
void removeDuplicates(struct Node** head_ref, int key) {
    bool firstOccurrence = false;               // To track the first occurrence of the key
    struct Node* current = *head_ref;           // Start with the head of the list
    struct Node* previous = NULL;               // To keep track of the previous node

    while (current != NULL) {                       // Iterate through each node in the list
        if (current->data == key) {                 // Check if the current node's data is the key
            if (!firstOccurrence) {                 // If it's the first occurrence
                firstOccurrence = true;             // Mark first occurrence
                previous = current;                 // Move previous to current
            } else {                                // If it's not the first occurrence
                if (previous != NULL) {             // Check if previous is not null
                    previous->next = current->next; // Remove the duplicate occurrence
                } else {
                    *head_ref = current->next;      // Update head if the first node is a duplicate
                }
                struct Node* temp = current;     // Temporary node to free memory
                current = current->next;        // Move to the next node
                free(temp);                     // Free the memory of the removed node
                continue;                       // Skip the rest of the loop
            }
        } else {                 // If current.data is not the key
            previous = current; // Move previous to current
        }
        current = current->next; // Move to the next node
    }
}

// Function to push a new node to the linked list
void push(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));          // Allocate memory for the new node
    new_node->data = new_data;                      // Set the data
    new_node->next = (*head_ref);                   // Link the new node to the head
    (*head_ref) = new_node;                         // Update the head to point to the new node
}

// Function to print the linked list
void printList(struct Node* node) {
    while (node != NULL) {                  // Iterate through each node
        printf("%d -> ", node->data);       // Print the data of the current node
        node = node->next;                  // Move to the next node
    }
    printf("NULL\n");                       // Indicate the end of the list
}

int main() {
    struct Node* head = NULL; // Initialize the head of the list

    // Add nodes to the linked list
    push(&head, 5);
    push(&head, 4);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 2);
    push(&head, 1);

    printf("Original List:\n");
    printList(head);

    removeDuplicates(&head, 2); // Remove duplicates of the key '2'

    printf("List after removing duplicates of key 2:\n");
    printList(head); // Print the list after removing duplicates

    return 0;
```

Figure 10: C code showing implementation of linked lists specifically how to remove multiple occurrences of an element KEY and remain with only a single occurrence of the element KEY in a list.

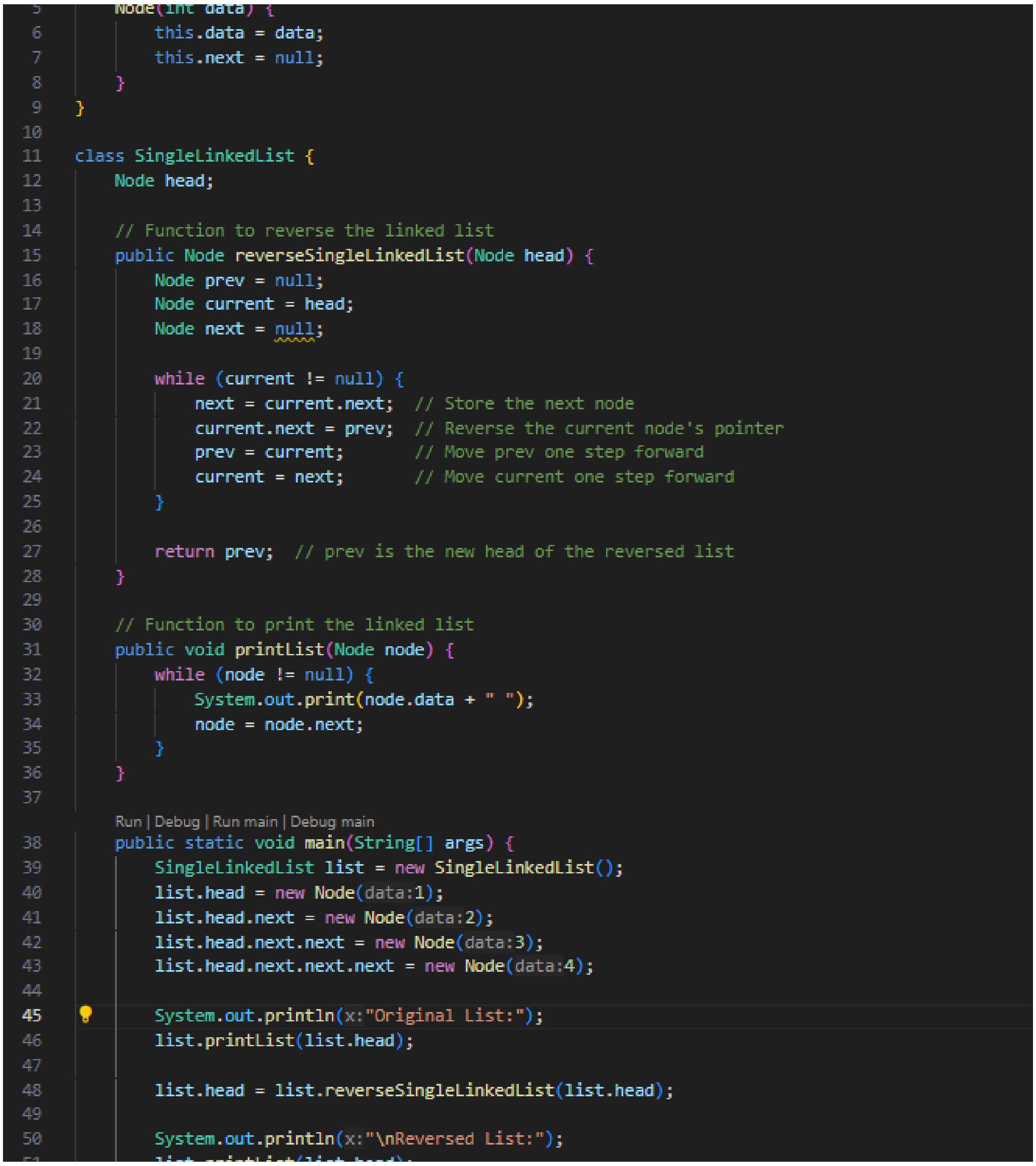
b) Write a function to reverse a singly linked list. The function should take the head of the list as input and return the head of the new reversed list.

Pseudocode

- Set a pointer Prev to NULL. //to help track the previous node
- Set another point Current to the head of the LL. //to help track the current list being processed
- Loop through the list till the end and save the next node by setting a temporary pointer Next to point to the Current's next node.
- Reverse the current node's link by making it point backward to the previous node.
- Move the previous pointer one step forward. //Prev to Current
- Move the current pointer one step forward to continue traversing. //Current to Next
- Once the loop ends, Prev will be pointing to the last node in the original list which is now the first node in the reversed list.
- Set the head of the linked list HEADER to Prev.
- Now the list is fully reversed, and the new head of the reversed list is in Prev.

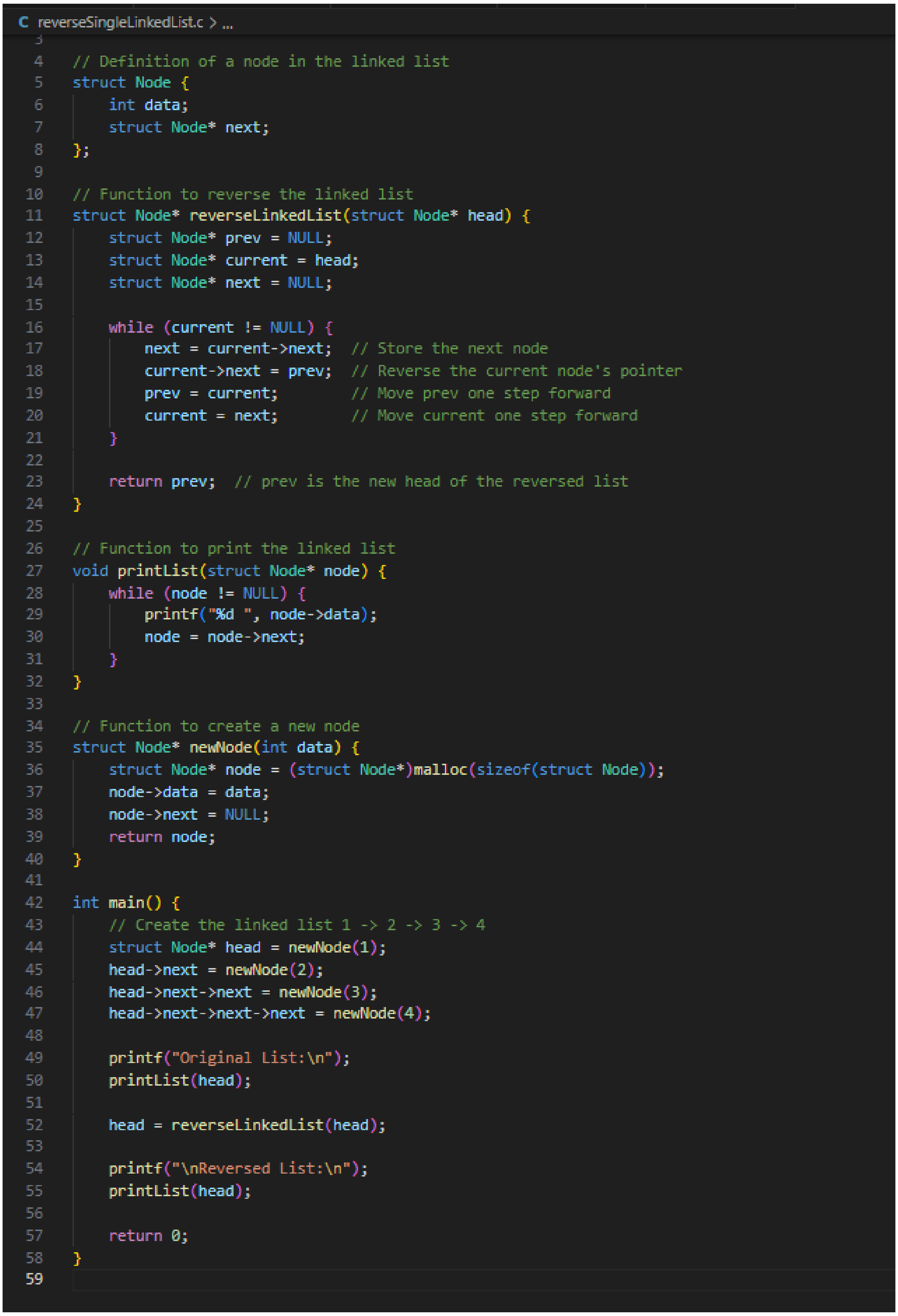
Method: reverseList

```
{
   Prev = NULL                                      // To track the previous node
   Current = HEADER                                 // Start from the HEADER node

   While (Current != NULL)                          // Traverse through the list
   {
      Next = Current -> LINK                        // Store the next node
      Current -> LINK = Prev                        // Reverse the LINK
      Prev = Current              // Move Prev one step forward (Prev becomes Current)
      Current = Next             // Move Current one step forward (Current becomes Next)
   }

   HEADER = Prev                  // Update HEADER to the new head of the reversed list
}
```



```java
    Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class SingleLinkedList {
    Node head;

    // Function to reverse the linked list
    public Node reverseSingleLinkedList(Node head) {
        Node prev = null;
        Node current = head;
        Node next = null;

        while (current != null) {
            next = current.next;  // Store the next node
            current.next = prev;  // Reverse the current node's pointer
            prev = current;       // Move prev one step forward
            current = next;       // Move current one step forward
        }

        return prev;  // prev is the new head of the reversed list
    }

    // Function to print the linked list
    public void printList(Node node) {
        while (node != null) {
            System.out.print(node.data + " ");
            node = node.next;
        }
    }

    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        SingleLinkedList list = new SingleLinkedList();
        list.head = new Node(data:1);
        list.head.next = new Node(data:2);
        list.head.next.next = new Node(data:3);
        list.head.next.next.next = new Node(data:4);

        System.out.println(x:"Original List:");
        list.printList(list.head);

        list.head = list.reverseSingleLinkedList(list.head);

        System.out.println(x:"\nReversed List:");
```

Figure 11: Java code showing how to reverse a singly linked list



```
C reverseSingleLinkedList.c > ...

// Definition of a node in the linked list
struct Node {
    int data;
    struct Node* next;
};

// Function to reverse the linked list
struct Node* reverseLinkedList(struct Node* head) {
    struct Node* prev = NULL;
    struct Node* current = head;
    struct Node* next = NULL;

    while (current != NULL) {
        next = current->next;  // Store the next node
        current->next = prev;  // Reverse the current node's pointer
        prev = current;        // Move prev one step forward
        current = next;        // Move current one step forward
    }

    return prev;  // prev is the new head of the reversed list
}

// Function to print the linked list
void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
}

// Function to create a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->next = NULL;
    return node;
}

int main() {
    // Create the linked list 1 -> 2 -> 3 -> 4
    struct Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);

    printf("Original List:\n");
    printList(head);

    head = reverseLinkedList(head);

    printf("\nReversed List:\n");
    printList(head);

    return 0;
}
```

Figure 12: C code showing how to reverse a singly linked list

TASK 4: INFIX, PREFIX, AND POSTFIX NOTATION

a) **With relevant examples, demonstrate infix, prefix (Polish), and postfix (Reverse Polish) notations.**

<u>INFIX NOTATION</u>

<u>Examples</u>

Expression: A + B

Infix: A + B

Expression: (A + B) * (C - D)

Infix: (A + B) * (C - D)

Expression: A * (B + C) – D / E

Infix: A * (B + C) – D / E

<u>PREFIX (POLISH) NOTATION</u>

<u>Examples</u>

Expression: A + B

Prefix: + A B

Expression: (A + B) * (C - D)

Prefix: * + A B - C D

Expression: A * (B + C) – D / E

Prefix: - * A + B C / D E

<u>Using a stack</u>

1. First, we reverse the infix expression e.g. 3rd expression

   Infix: A * (B + C) – D / E

   Reversed Infix: E / D – (C + B) * A.

2. Then convert the reversed expression into a postfix notation using the standard infix-to-postfix algorithm.

   Postfix: E D / C B + A * -

3. Reverse the postfix expression to get the prefix expression.

   Postfix: E D / C B + A * -

   Reversed postfix: - * A + B C / D E

<u>Standard infix-to-prefix algorithm</u>

1. Start by finding the operator with the lowest precedence that controls the overall expression.
2. Place that operator before its operands
3. Do the same thing to the sub-expressions (operands).
4. Combine all parts into the final prefix expression.
5. e.g. in 3rd expression
6. The outermost operator with the lowest precedence is –
7. Left sub-expression: A * (B + C) and right sub-expression: D / E
8. So, for the left, we have: * A + B C, and for the right: / D E
9. Final prefix notation: - * A + B C / D E

POSTFIX (REVERSE POLISH) NOTATION

Examples

Expression: A + B

Postfix: A B +

Expression: (A + B) * (C - D)

Postfix: A B + C D - *

Expression: A * (B + C) – D / E

Postfix: A B C + * D E / -

Using a stack

1. Scan the expression from left to right, meet an operand (e.g. A, B, etc.) then append it to the postfix expression.
2. Meet an operator (e.g. *, +) push to stack after popping those of higher or equal precedence (i.e. trying to push a + but there's already a *, first pop the * then push the + then the * again)
3. Push a (onto the stack and pop operators when encountering) until (is reached.
4. Pop the remaining operators from the stack and append them to the postfix expression after the entire expression is scanned.

   Expression: A * (B + C) – D / E

   Postfix: A B C + * D E / -

Standard infix-to-postfix algorithm

1. Start by breaking down the expression based on precedence.
2. Starting from the innermost sub-expression, converting each to postfix as you go.
3. Combine all parts into the final postfix expression ensuring the operators follow the operands they operate on.
4. e.g. in 3rd expression
5. Innermost sub-expression 1: (B + C) to postfix B C +
6. Sub-expression 2: A * (B + C) to postfix A B C + *
7. Sub-expression 3: D / E to postfix D E /
8. Final notation: A B C + * D E / -

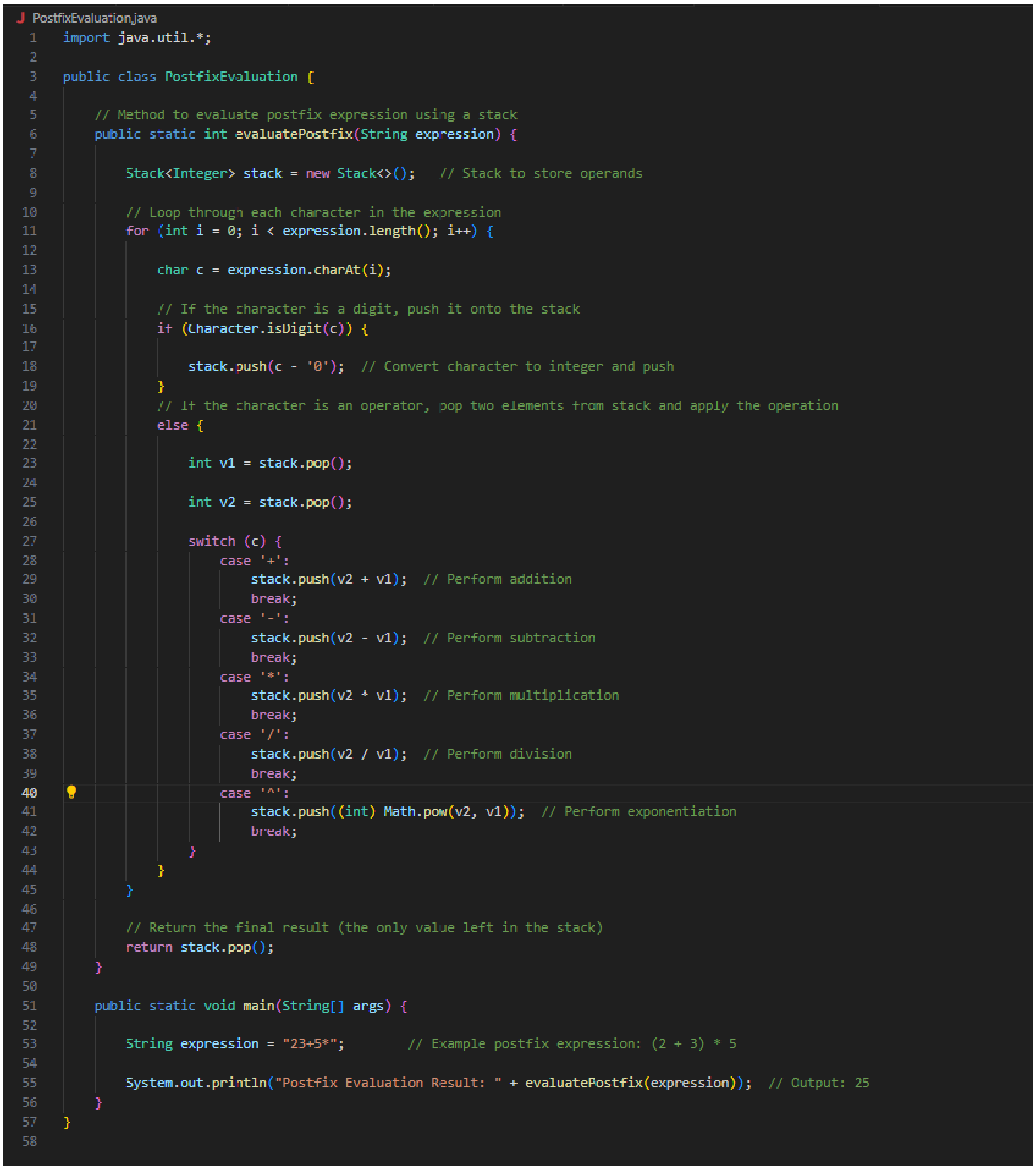
4. Use postfix notation to evaluate any arithmetic expression with the aid of a stack data structure.

- Arithmetic expression : (2 + 3) * 5..........Postfix : 2 3 + 5 *
- Initial stack is empty [ ]
- Process 2: Push ‘2’ onto stack -> [2]
- Process 3: Push ‘3’ onto stack -> [2, 3]
- Process +: Pop ‘3’ and ‘2’, add them (2 + 3 = 5) then push ‘5’ to stack -> [5]
- Process 5: Push ‘5’ onto stack -> [5, 5]
- Process *: Pop ‘5’ and ‘5’, multiply them (5 * 5 = 25), and the push 25 to stack -> [25]
- Final result: The result is 25, which is returned.

Method: evaluatePostfix(expression)

```
1. Stack S = new Stack()    // Initialize an empty stack to store operands.
2. For each character c in expression:    //Loop through each character in the expression
    1. If c is a digit:     // If the character is a digit (operand), push it onto the stack
        1. S.push(c - '0')  // Convert the character to an integer ('0' ensures correct conversion)
    2. Else if c is an operator:   // If the character c is an operator, pop two operands from the stack for the operation
        1.        v1 = S.pop()  // Pop the top value (v1) from the stack (this is the right operand)
        2.        v2 = S.pop()  // Pop the next value (v2) from the stack (this is the left operand)
```

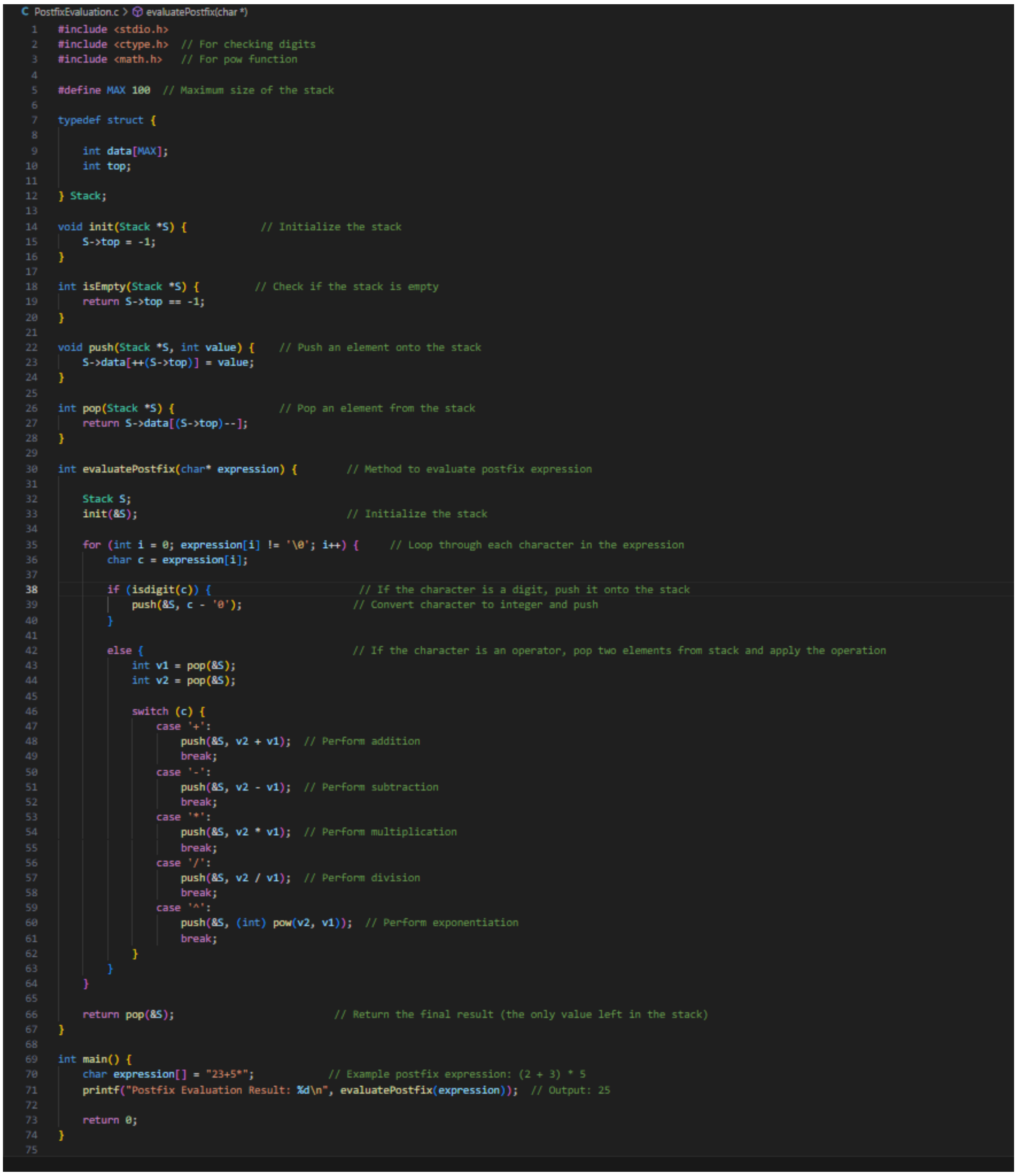
```
            3.   Switch (c):    //Operate based on the operator and push result
                     Case '+':
                                S.push(v2 + v1)  // Add the two popped values
                     Case '-':
                       S.push(v2 - v1)  // Subtract the two popped values
                     Case '*':
                       S.push(v2 * v1)  // Multiply the two popped values
                     Case '/':
                       S.push(v2 / v1)  // Divide the two popped values
                     Case '^':
                       S.push(pow(v2 , v1))  // Compute v2 raised to the power v1
                 endIf
        3.   Return S.pop()
3. End
```



PostfixEvaluation.java

```java
import java.util.*;

public class PostfixEvaluation {

    // Method to evaluate postfix expression using a stack
    public static int evaluatePostfix(String expression) {

        Stack<Integer> stack = new Stack<>();   // Stack to store operands

        // Loop through each character in the expression
        for (int i = 0; i < expression.length(); i++) {

            char c = expression.charAt(i);

            // If the character is a digit, push it onto the stack
            if (Character.isDigit(c)) {

                stack.push(c - '0');  // Convert character to integer and push
            }
            // If the character is an operator, pop two elements from stack and apply the operation
            else {

                int v1 = stack.pop();

                int v2 = stack.pop();

                switch (c) {
                    case '+':
                        stack.push(v2 + v1);  // Perform addition
                        break;
                    case '-':
                        stack.push(v2 - v1);  // Perform subtraction
                        break;
                    case '*':
                        stack.push(v2 * v1);  // Perform multiplication
                        break;
                    case '/':
                        stack.push(v2 / v1);  // Perform division
                        break;
                    case '^':
                        stack.push((int) Math.pow(v2, v1));  // Perform exponentiation
                        break;
                }
            }
        }

        // Return the final result (the only value left in the stack)
        return stack.pop();
    }

    public static void main(String[] args) {

        String expression = "23+5*";        // Example postfix expression: (2 + 3) * 5

        System.out.println("Postfix Evaluation Result: " + evaluatePostfix(expression));  // Output: 25
    }
}
```

*Figure 13: Java code showing implementation of stacks using the postfix notation to evaluate an arithmetic expression (2 + 3) * 5 whose postfix expression is 23 + 5 **



```c
#include <stdio.h>
#include <ctype.h>  // For checking digits
#include <math.h>   // For pow function

#define MAX 100  // Maximum size of the stack

typedef struct {

    int data[MAX];
    int top;

} Stack;

void init(Stack *S) {             // Initialize the stack
    S->top = -1;
}

int isEmpty(Stack *S) {          // Check if the stack is empty
    return S->top == -1;
}

void push(Stack *S, int value) {    // Push an element onto the stack
    S->data[++(S->top)] = value;
}

int pop(Stack *S) {                 // Pop an element from the stack
    return S->data[(S->top)--];
}

int evaluatePostfix(char* expression) {        // Method to evaluate postfix expression

    Stack S;
    init(&S);                                       // Initialize the stack

    for (int i = 0; expression[i] != '\0'; i++) {     // Loop through each character in the expression
        char c = expression[i];

        if (isdigit(c)) {                          // If the character is a digit, push it onto the stack
            push(&S, c - '0');                    // Convert character to integer and push
        }

        else {                                    // If the character is an operator, pop two elements from stack and apply the operation
            int v1 = pop(&S);
            int v2 = pop(&S);

            switch (c) {
                case '+':
                    push(&S, v2 + v1);  // Perform addition
                    break;
                case '-':
                    push(&S, v2 - v1);  // Perform subtraction
                    break;
                case '*':
                    push(&S, v2 * v1);  // Perform multiplication
                    break;
                case '/':
                    push(&S, v2 / v1);  // Perform division
                    break;
                case '^':
                    push(&S, (int) pow(v2, v1));  // Perform exponentiation
                    break;
            }
        }
    }

    return pop(&S);                           // Return the final result (the only value left in the stack)
}

int main() {
    char expression[] = "23+5*";            // Example postfix expression: (2 + 3) * 5
    printf("Postfix Evaluation Result: %d\n", evaluatePostfix(expression));  // Output: 25

    return 0;
}
```

*Figure 14: C code showing the implementation of stacks using the postfix notation to evaluate an arithmetic expression (2 + 3) * 5 whose postfix expression is 23 + 5 **

TASK 5: APPLICATIONS OF STACKS

BROWSER NAVIGATION SYSTEM

1. You are tasked with creating a simple browser navigation system that simulates the "Back" and "Forward" functionality using stacks. Your program should handle the following operations:
2. Visit a new webpage: This should push the current page onto the "Back" stack and make the new page the current page. The "Forward" stack should be cleared whenever a new page is visited.
3. Back: This should move the current page to the "Forward" stack, and the last visited page should be popped from the "Back" stack and made the current page.
4. Forward: This should move the current page to the "Back" stack, and the last page from the "Forward" stack should be made the current page.

Initial Setup:

1. Start with a homepage, e.g., "Home".
2. The "Back" and "Forward" stacks are initially empty.

Operations to Implement:

Visit(page): Visit a new webpage (e.g., Visit("Page1"))

Back(): Go back to the previous page.

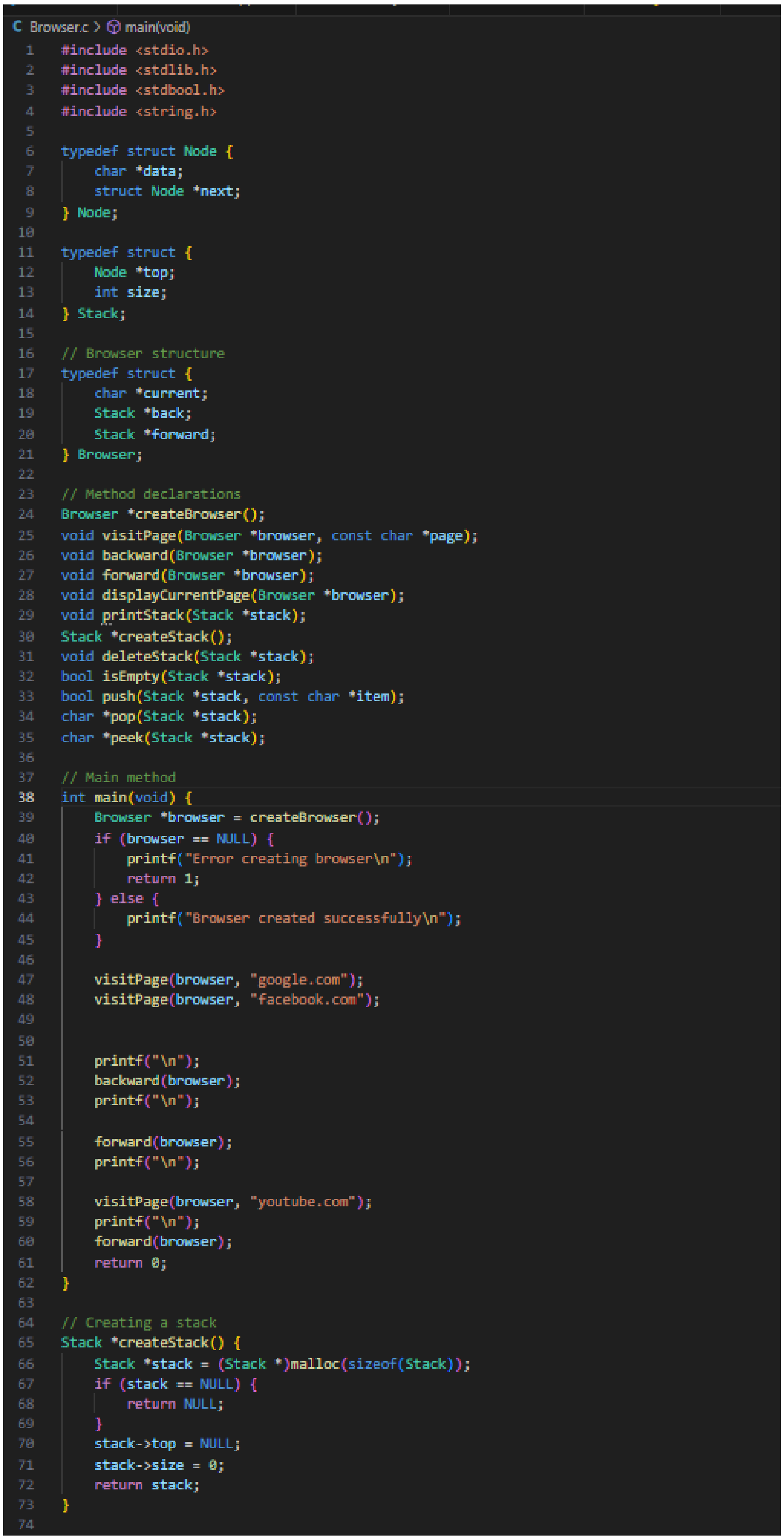
Forward(): Go forward to the next page.

DisplayCurrentPage(): Output the current page.

DisplayBackStack(): Output the contents of the "Back" stack.

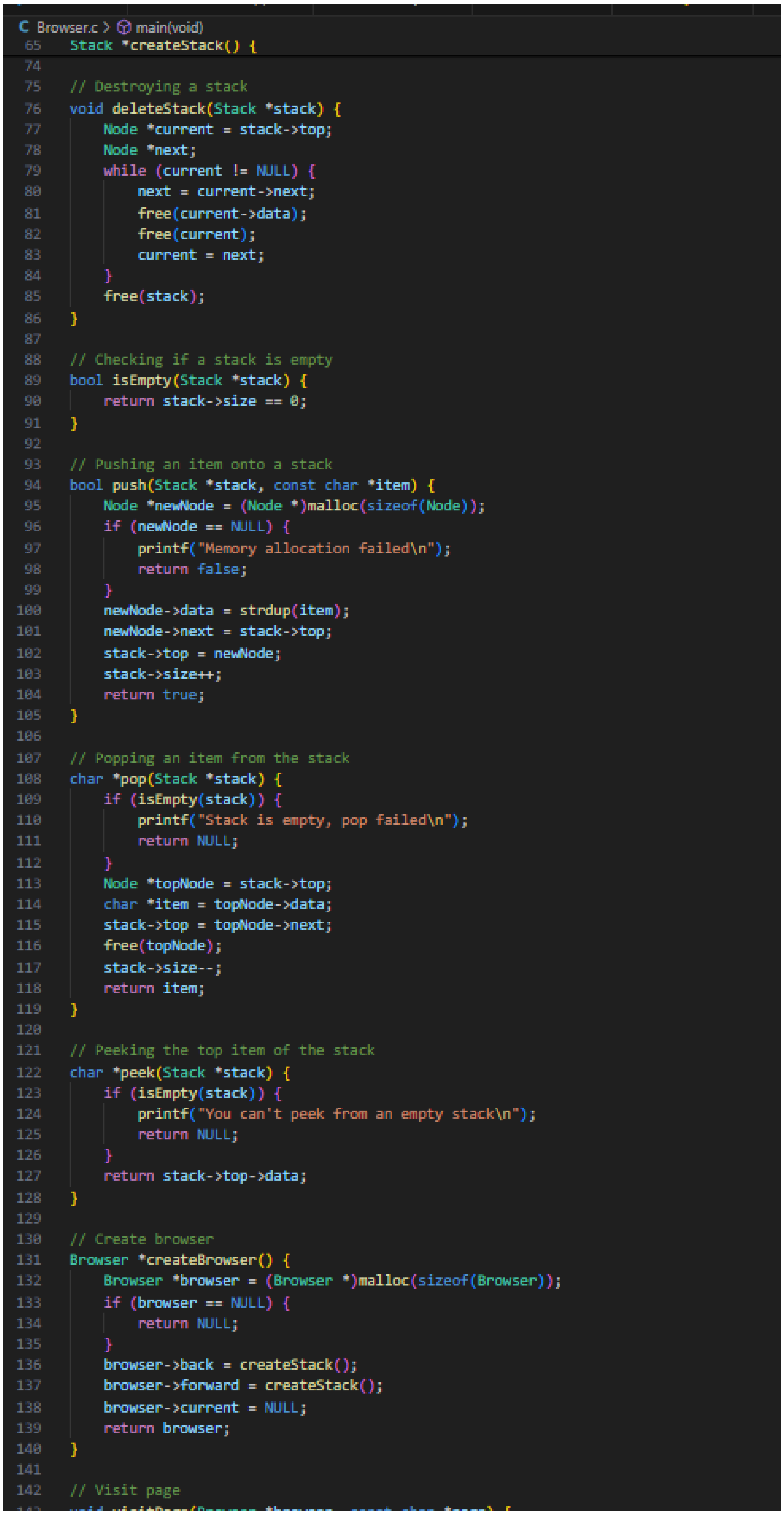
DisplayForwardStack(): Output the contents of the "Forward" stack.

C-LINKED LISTS

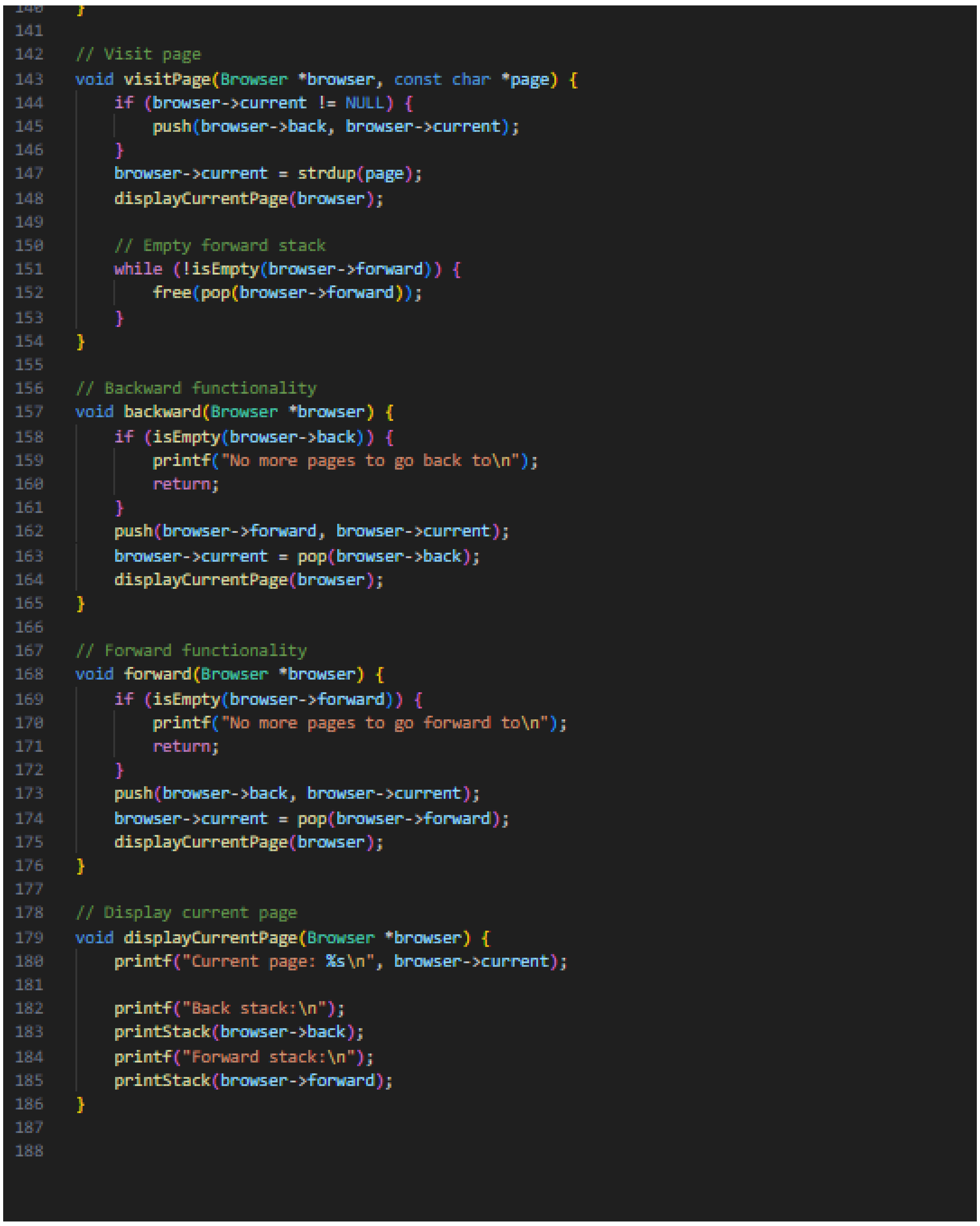


Browser.c > main(void)

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

typedef struct Node {
    char *data;
    struct Node *next;
} Node;

typedef struct {
    Node *top;
    int size;
} Stack;

// Browser structure
typedef struct {
    char *current;
    Stack *back;
    Stack *forward;
} Browser;

// Method declarations
Browser *createBrowser();
void visitPage(Browser *browser, const char *page);
void backward(Browser *browser);
void forward(Browser *browser);
void displayCurrentPage(Browser *browser);
void printStack(Stack *stack);
Stack *createStack();
void deleteStack(Stack *stack);
bool isEmpty(Stack *stack);
bool push(Stack *stack, const char *item);
char *pop(Stack *stack);
char *peek(Stack *stack);

// Main method
int main(void) {
    Browser *browser = createBrowser();
    if (browser == NULL) {
        printf("Error creating browser\n");
        return 1;
    } else {
        printf("Browser created successfully\n");
    }

    visitPage(browser, "google.com");
    visitPage(browser, "facebook.com");


    printf("\n");
    backward(browser);
    printf("\n");

    forward(browser);
    printf("\n");

    visitPage(browser, "youtube.com");
    printf("\n");
    forward(browser);
    return 0;
}

// Creating a stack
Stack *createStack() {
    Stack *stack = (Stack *)malloc(sizeof(Stack));
    if (stack == NULL) {
        return NULL;
    }
    stack->top = NULL;
    stack->size = 0;
    return stack;
}

```

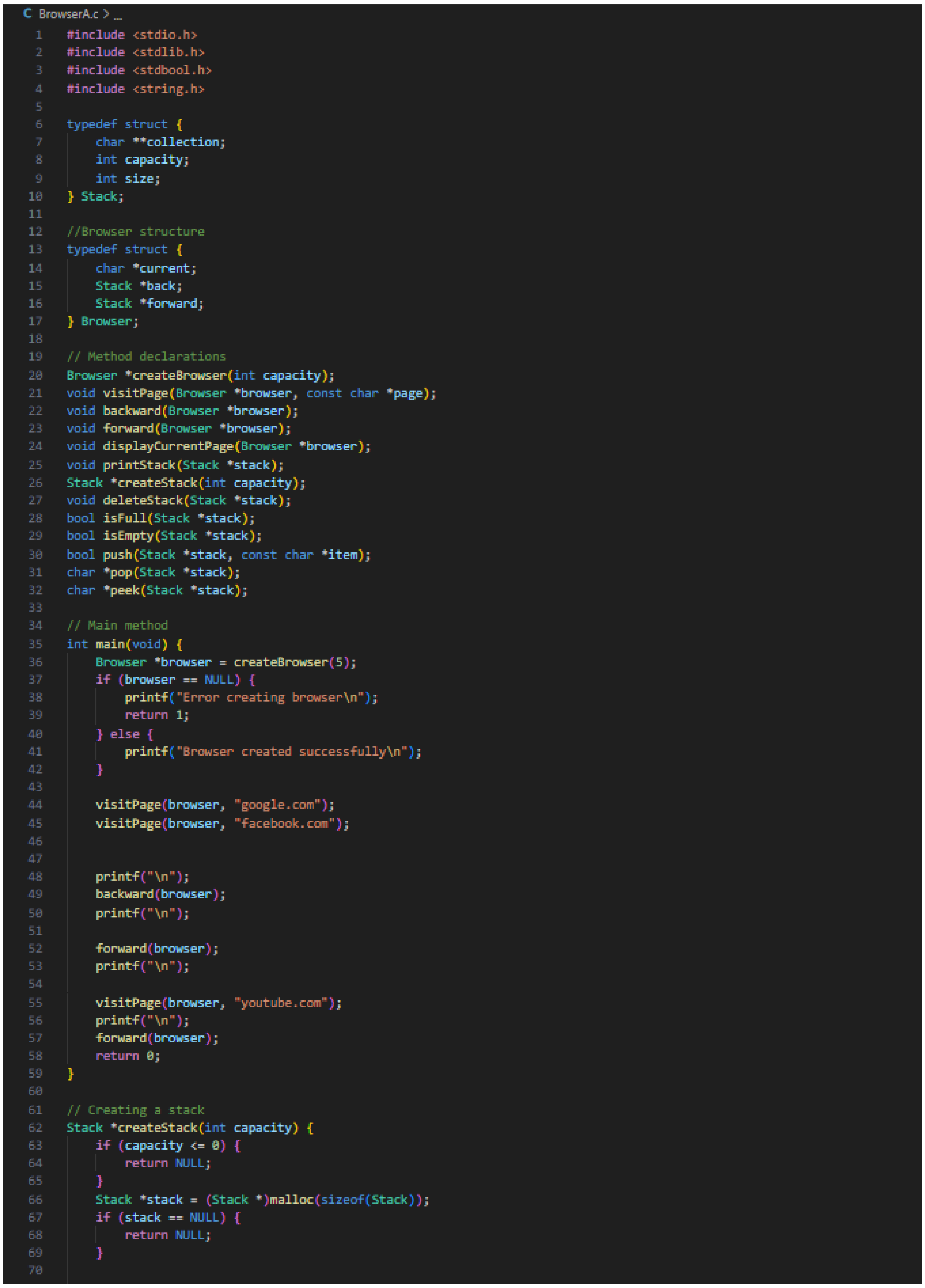


```c
Stack *createStack() {

// Destroying a stack
void deleteStack(Stack *stack) {
    Node *current = stack->top;
    Node *next;
    while (current != NULL) {
        next = current->next;
        free(current->data);
        free(current);
        current = next;
    }
    free(stack);
}

// Checking if a stack is empty
bool isEmpty(Stack *stack) {
    return stack->size == 0;
}

// Pushing an item onto a stack
bool push(Stack *stack, const char *item) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        return false;
    }
    newNode->data = strdup(item);
    newNode->next = stack->top;
    stack->top = newNode;
    stack->size++;
    return true;
}

// Popping an item from the stack
char *pop(Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty, pop failed\n");
        return NULL;
    }
    Node *topNode = stack->top;
    char *item = topNode->data;
    stack->top = topNode->next;
    free(topNode);
    stack->size--;
    return item;
}

// Peeking the top item of the stack
char *peek(Stack *stack) {
    if (isEmpty(stack)) {
        printf("You can't peek from an empty stack\n");
        return NULL;
    }
    return stack->top->data;
}

// Create browser
Browser *createBrowser() {
    Browser *browser = (Browser *)malloc(sizeof(Browser));
    if (browser == NULL) {
        return NULL;
    }
    browser->back = createStack();
    browser->forward = createStack();
    browser->current = NULL;
    return browser;
}

// Visit page
void visitPage(Browser *browser, const char *page) {
```



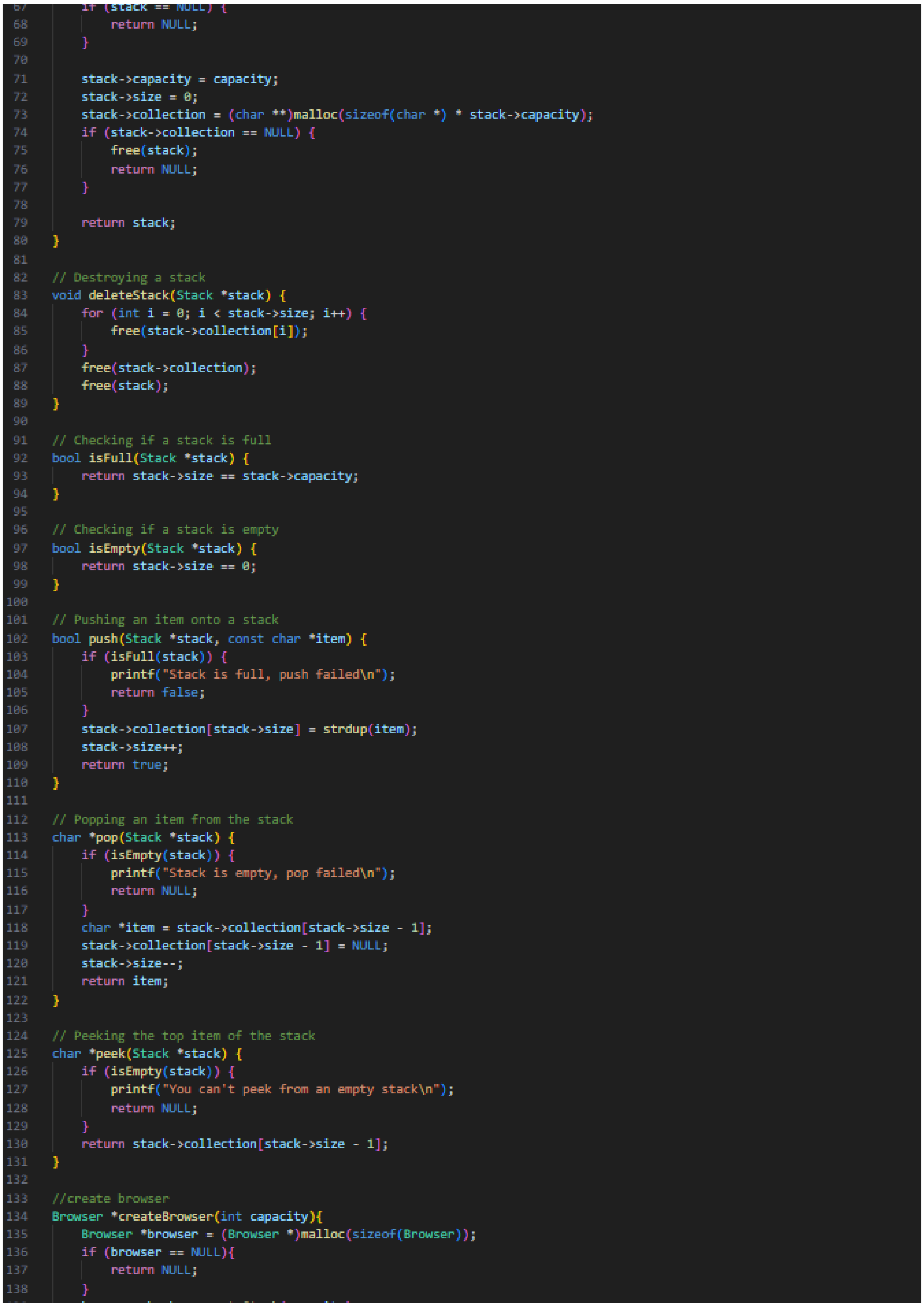
```c
}

// Visit page
void visitPage(Browser *browser, const char *page) {
    if (browser->current != NULL) {
        push(browser->back, browser->current);
    }
    browser->current = strdup(page);
    displayCurrentPage(browser);

    // Empty forward stack
    while (!isEmpty(browser->forward)) {
        free(pop(browser->forward));
    }
}

// Backward functionality
void backward(Browser *browser) {
    if (isEmpty(browser->back)) {
        printf("No more pages to go back to\n");
        return;
    }
    push(browser->forward, browser->current);
    browser->current = pop(browser->back);
    displayCurrentPage(browser);
}

// Forward functionality
void forward(Browser *browser) {
    if (isEmpty(browser->forward)) {
        printf("No more pages to go forward to\n");
        return;
    }
    push(browser->back, browser->current);
    browser->current = pop(browser->forward);
    displayCurrentPage(browser);
}

// Display current page
void displayCurrentPage(Browser *browser) {
    printf("Current page: %s\n", browser->current);

    printf("Back stack:\n");
    printStack(browser->back);
    printf("Forward stack:\n");
    printStack(browser->forward);
}


```

C-ARRAYS

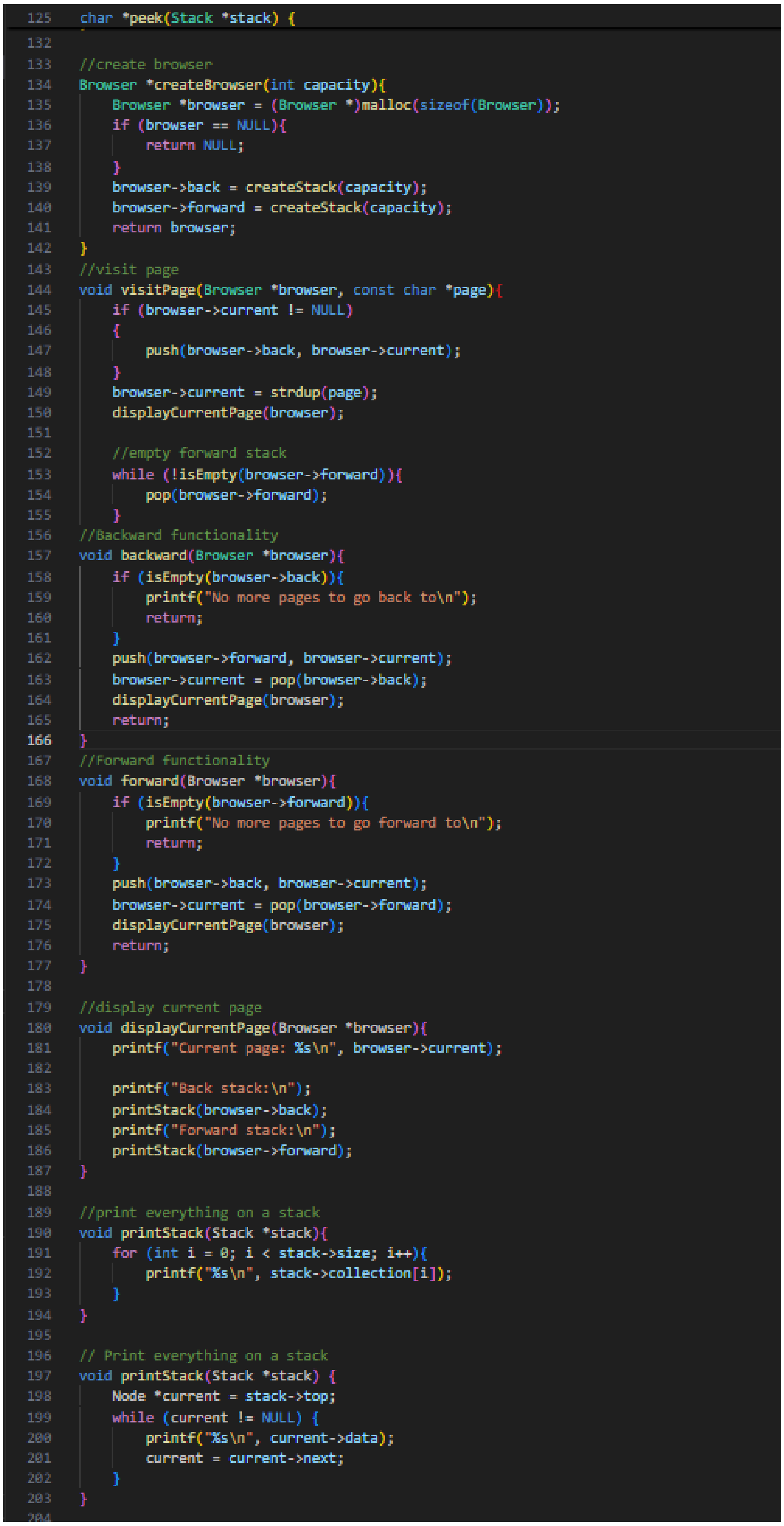


BrowserA.c > ...

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

typedef struct {
    char **collection;
    int capacity;
    int size;
} Stack;

//Browser structure
typedef struct {
    char *current;
    Stack *back;
    Stack *forward;
} Browser;

// Method declarations
Browser *createBrowser(int capacity);
void visitPage(Browser *browser, const char *page);
void backward(Browser *browser);
void forward(Browser *browser);
void displayCurrentPage(Browser *browser);
void printStack(Stack *stack);
Stack *createStack(int capacity);
void deleteStack(Stack *stack);
bool isFull(Stack *stack);
bool isEmpty(Stack *stack);
bool push(Stack *stack, const char *item);
char *pop(Stack *stack);
char *peek(Stack *stack);

// Main method
int main(void) {
    Browser *browser = createBrowser(5);
    if (browser == NULL) {
        printf("Error creating browser\n");
        return 1;
    } else {
        printf("Browser created successfully\n");
    }

    visitPage(browser, "google.com");
    visitPage(browser, "facebook.com");


    printf("\n");
    backward(browser);
    printf("\n");

    forward(browser);
    printf("\n");

    visitPage(browser, "youtube.com");
    printf("\n");
    forward(browser);
    return 0;
}

// Creating a stack
Stack *createStack(int capacity) {
    if (capacity <= 0) {
        return NULL;
    }
    Stack *stack = (Stack *)malloc(sizeof(Stack));
    if (stack == NULL) {
        return NULL;
    }

```



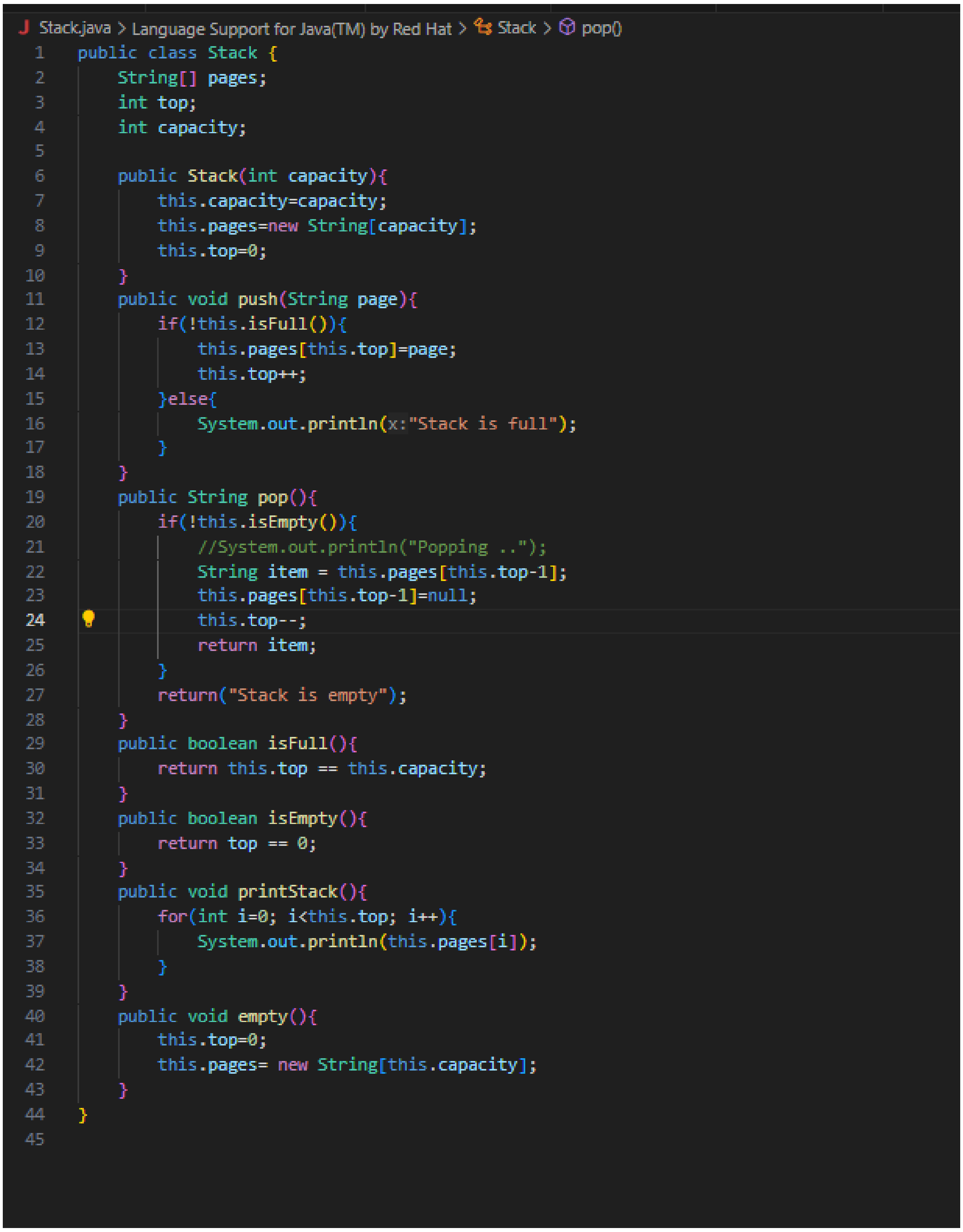
```c
    if (stack == NULL) {
        return NULL;
    }

    stack->capacity = capacity;
    stack->size = 0;
    stack->collection = (char **)malloc(sizeof(char *) * stack->capacity);
    if (stack->collection == NULL) {
        free(stack);
        return NULL;
    }

    return stack;
}

// Destroying a stack
void deleteStack(Stack *stack) {
    for (int i = 0; i < stack->size; i++) {
        free(stack->collection[i]);
    }
    free(stack->collection);
    free(stack);
}

// Checking if a stack is full
bool isFull(Stack *stack) {
    return stack->size == stack->capacity;
}

// Checking if a stack is empty
bool isEmpty(Stack *stack) {
    return stack->size == 0;
}

// Pushing an item onto a stack
bool push(Stack *stack, const char *item) {
    if (isFull(stack)) {
        printf("Stack is full, push failed\n");
        return false;
    }
    stack->collection[stack->size] = strdup(item);
    stack->size++;
    return true;
}

// Popping an item from the stack
char *pop(Stack *stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty, pop failed\n");
        return NULL;
    }
    char *item = stack->collection[stack->size - 1];
    stack->collection[stack->size - 1] = NULL;
    stack->size--;
    return item;
}

// Peeking the top item of the stack
char *peek(Stack *stack) {
    if (isEmpty(stack)) {
        printf("You can't peek from an empty stack\n");
        return NULL;
    }
    return stack->collection[stack->size - 1];
}

//create browser
Browser *createBrowser(int capacity){
    Browser *browser = (Browser *)malloc(sizeof(Browser));
    if (browser == NULL){
        return NULL;
    }
```



```c
char *peek(Stack *stack) {

//create browser
Browser *createBrowser(int capacity){
    Browser *browser = (Browser *)malloc(sizeof(Browser));
    if (browser == NULL){
        return NULL;
    }
    browser->back = createStack(capacity);
    browser->forward = createStack(capacity);
    return browser;
}
//visit page
void visitPage(Browser *browser, const char *page){
    if (browser->current != NULL)
    {
        push(browser->back, browser->current);
    }
    browser->current = strdup(page);
    displayCurrentPage(browser);

    //empty forward stack
    while (!isEmpty(browser->forward)){
        pop(browser->forward);
    }
//Backward functionality
void backward(Browser *browser){
    if (isEmpty(browser->back)){
        printf("No more pages to go back to\n");
        return;
    }
    push(browser->forward, browser->current);
    browser->current = pop(browser->back);
    displayCurrentPage(browser);
    return;
}
//Forward functionality
void forward(Browser *browser){
    if (isEmpty(browser->forward)){
        printf("No more pages to go forward to\n");
        return;
    }
    push(browser->back, browser->current);
    browser->current = pop(browser->forward);
    displayCurrentPage(browser);
    return;
}

//display current page
void displayCurrentPage(Browser *browser){
    printf("Current page: %s\n", browser->current);

    printf("Back stack:\n");
    printStack(browser->back);
    printf("Forward stack:\n");
    printStack(browser->forward);
}

//print everything on a stack
void printStack(Stack *stack){
    for (int i = 0; i < stack->size; i++){
        printf("%s\n", stack->collection[i]);
    }
}

// Print everything on a stack
void printStack(Stack *stack) {
    Node *current = stack->top;
    while (current != NULL) {
        printf("%s\n", current->data);
        current = current->next;
    }
}
```

JAVA IMPLEMENTATION

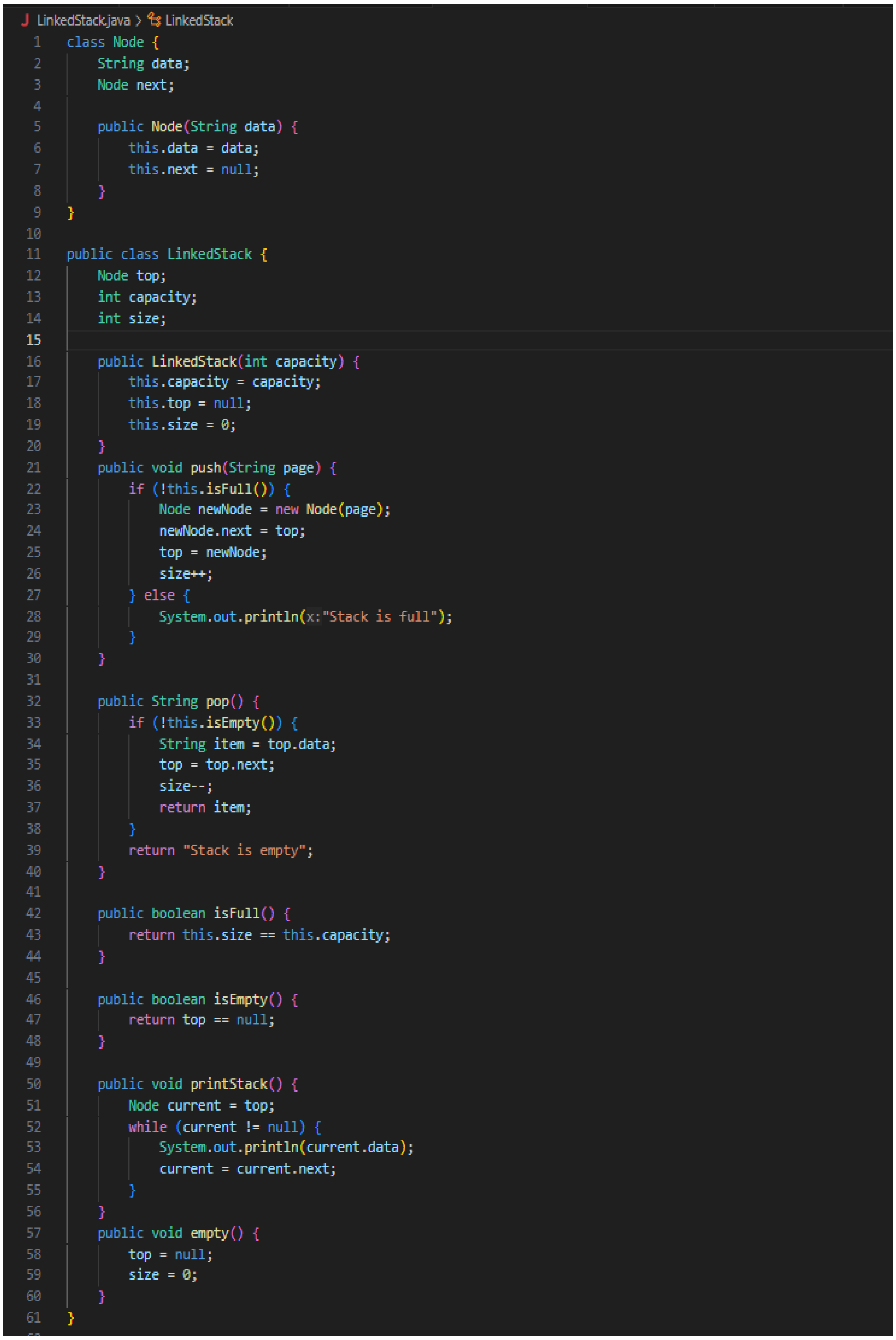
Array-Based Stack



Stack.java > Language Support for Java(TM) by Red Hat > Stack > pop()

```java
public class Stack {
    String[] pages;
    int top;
    int capacity;

    public Stack(int capacity){
        this.capacity=capacity;
        this.pages=new String[capacity];
        this.top=0;
    }
    public void push(String page){
        if(!this.isFull()){
            this.pages[this.top]=page;
            this.top++;
        }else{
            System.out.println(x:"Stack is full");
        }
    }
    public String pop(){
        if(!this.isEmpty()){
            //System.out.println("Popping ..");
            String item = this.pages[this.top-1];
            this.pages[this.top-1]=null;
            this.top--;
            return item;
        }
        return("Stack is empty");
    }
    public boolean isFull(){
        return this.top == this.capacity;
    }
    public boolean isEmpty(){
        return top == 0;
    }
    public void printStack(){
        for(int i=0; i<this.top; i++){
            System.out.println(this.pages[i]);
        }
    }
    public void empty(){
        this.top=0;
        this.pages= new String[this.capacity];
    }
}

```

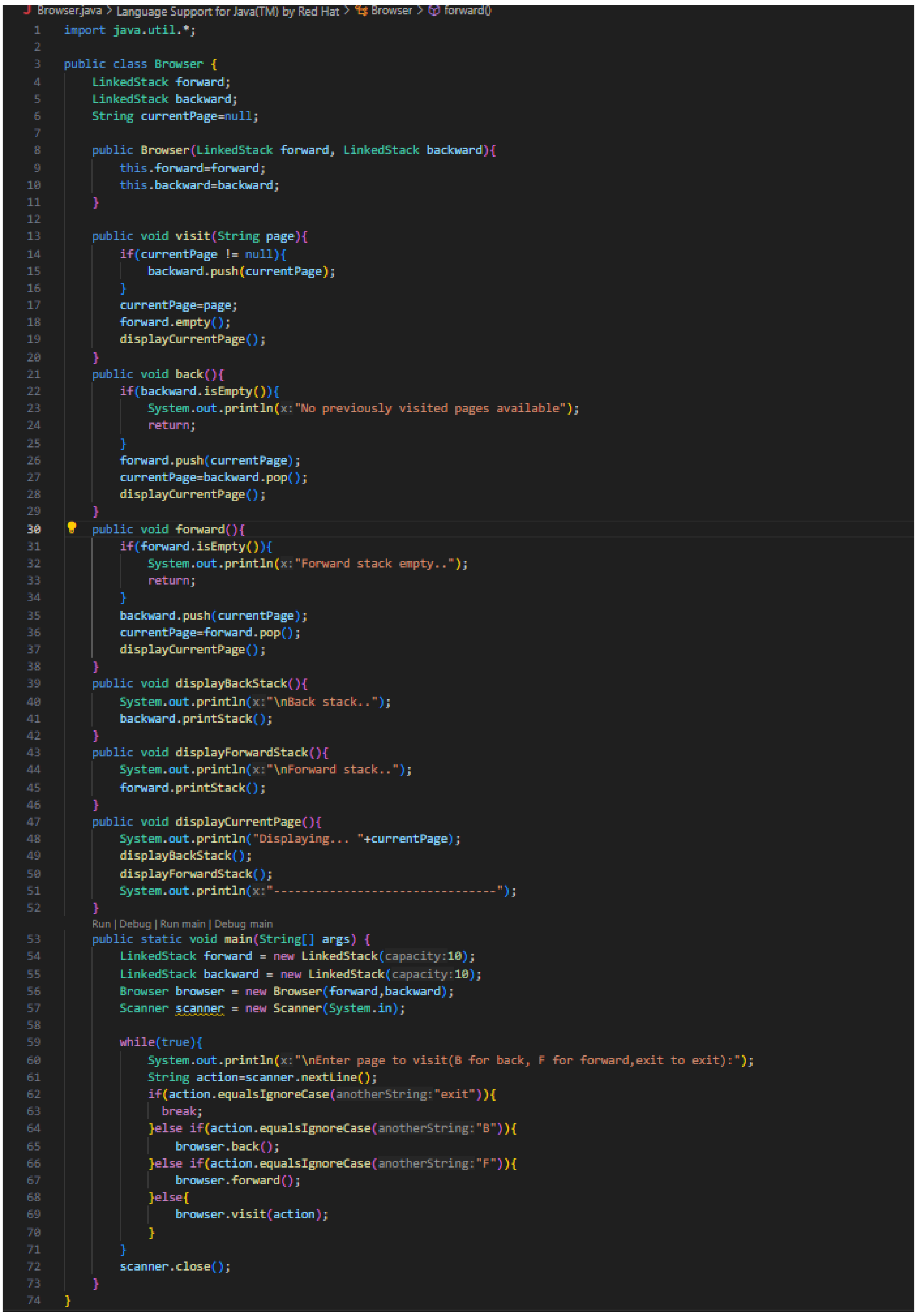
Linked List Stacks



LinkedStack.java > LinkedStack

```java
class Node {
    String data;
    Node next;

    public Node(String data) {
        this.data = data;
        this.next = null;
    }
}

public class LinkedStack {
    Node top;
    int capacity;
    int size;

    public LinkedStack(int capacity) {
        this.capacity = capacity;
        this.top = null;
        this.size = 0;
    }
    public void push(String page) {
        if (!this.isFull()) {
            Node newNode = new Node(page);
            newNode.next = top;
            top = newNode;
            size++;
        } else {
            System.out.println(x:"Stack is full");
        }
    }

    public String pop() {
        if (!this.isEmpty()) {
            String item = top.data;
            top = top.next;
            size--;
            return item;
        }
        return "Stack is empty";
    }

    public boolean isFull() {
        return this.size == this.capacity;
    }

    public boolean isEmpty() {
        return top == null;
    }

    public void printStack() {
        Node current = top;
        while (current != null) {
            System.out.println(current.data);
            current = current.next;
        }
    }
    public void empty() {
        top = null;
        size = 0;
    }
}
```

Browser


Browser.java > Language Support for Java(TM) by Red Hat > Browser > forward()

```java
import java.util.*;

public class Browser {
    LinkedStack forward;
    LinkedStack backward;
    String currentPage=null;

    public Browser(LinkedStack forward, LinkedStack backward){
        this.forward=forward;
        this.backward=backward;
    }

    public void visit(String page){
        if(currentPage != null){
            backward.push(currentPage);
        }
        currentPage=page;
        forward.empty();
        displayCurrentPage();
    }
    public void back(){
        if(backward.isEmpty()){
            System.out.println(x:"No previously visited pages available");
            return;
        }
        forward.push(currentPage);
        currentPage=backward.pop();
        displayCurrentPage();
    }
    public void forward(){
        if(forward.isEmpty()){
            System.out.println(x:"Forward stack empty..");
            return;
        }
        backward.push(currentPage);
        currentPage=forward.pop();
        displayCurrentPage();
    }
    public void displayBackStack(){
        System.out.println(x:"\nBack stack..");
        backward.printStack();
    }
    public void displayForwardStack(){
        System.out.println(x:"\nForward stack..");
        forward.printStack();
    }
    public void displayCurrentPage(){
        System.out.println("Displaying... "+currentPage);
        displayBackStack();
        displayForwardStack();
        System.out.println(x:"--------------------------------");
    }
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        LinkedStack forward = new LinkedStack(capacity:10);
        LinkedStack backward = new LinkedStack(capacity:10);
        Browser browser = new Browser(forward,backward);
        Scanner scanner = new Scanner(System.in);

        while(true){
            System.out.println(x:"\nEnter page to visit(B for back, F for forward,exit to exit):");
            String action=scanner.nextLine();
            if(action.equalsIgnoreCase(anotherString:"exit")){
                break;
            }else if(action.equalsIgnoreCase(anotherString:"B")){
                browser.back();
            }else if(action.equalsIgnoreCase(anotherString:"F")){
                browser.forward();
            }else{
                browser.visit(action);
            }
        }
        scanner.close();
    }
}
```

1. **How the "Back" and "Forward" stacks change with each operation.**

- When a new page is visited, the previous page is pushed onto the back stack where as the forward stack is emptied.
- When the back button is pressed, the current page is pushed onto the forward stack and the last added page on the back stack is popped and displayed.
- When the forward button is pressed, the current page that was being displayed is pushed onto the back stack and subsequently the last added page on the forward stack is popped and displayed.

2. **Consider an edge case where the user tries to go back when the "Back" stack is empty. How should your program handle this?**

- If a user tries to go back when the back stack is empty, a message, 'No more pages to go back to' will be displayed to avoid errors or unexpected behavior.

3. **What happens if the user visits a new page after using the "Back" button but before using the "Forward" button?**

- If a user visits a new page after using the back button but before using the forward button, the current page will still be pushed to the back stack whereas the forward stack will be emptied because visiting a new page invalidates forward navigation. In other words, you can only move forward if you previously moved backward.

4. **Design a function that allows the user to print the navigation history (both Back and Forward stacks).**

- In this navigation system whenever a page is visited the displayCurrentPage method has to be called which triggers methods to print the Back and Forward stacks. In the Java implementation, the displayBackStack() and displayForwardStack() methods are called whereas in the C implementation, the printStack() method is called on the back and forward stacks to display the contents.

TASK 6: QUEUES

Question 1:
Explain the reasoning behind choosing which queue a customer joins when they arrive at the checkout area. Why is it optimal for a customer to always join the shortest queue?

Answer:
Choosing the shortest queue optimizes the waiting time for each customer since it is associated with a shorter time range between arrival and processing time for a particular customer to be worked on, and balances the workload across the counters. By always selecting the queue with the fewest customers, customers spend less time waiting and leave sooner, enhancing overall customer satisfaction. Additionally, this method prevents certain counters from becoming overloaded, which improves processing efficiency as all counters have an even distribution of customers.

Question 2:
What happens when two or more queues are of equal length? How should the program decide which queue the customer should join? Discuss possible tie-breaking strategies (e.g., always choose the first queue in order, or choose randomly).

Answer:

If for example, there are five counters in a supermarket, if n counters(such that n<5 and n>=2) are of equal length, the program should look at all the 5 counters and determine which is shortest. In case it is the n counters that are the shortest, the program should pick the queue of the counter(among the n counters) that received the first instance of that customer number i.e. if three counters are of equal length and are all the shortest with length 4, the counter that received the first customer 4 should be picked.

Otherwise(if the n counters are the longest) The program should consider the remaining queues i.e. 5-n and determine the shortest queue among them.

if there are multiple instances of counters of equal length i.e. 2 have length 5, another 2 have length 7, the program should pick the shortest queue number(length) counters and from them, assign the customer to the counter that received the first instance customer of that number.

When multiple queues have the same length, tie-breaking strategies ensure customers are distributed fairly without favouring specific counters. Among those strategies include:

- Choosing the First Queue: Consistently joining the first queue that registered that number of customers, in cases of a tie provides a predictable rule, but may create a bias toward certain counters over time.

-**Random Selection:** A randomized choice among equal-length queues can distribute customers more evenly, avoiding bias and potentially speeding up service if each counter has similar processing speeds.

Question 3:
Consider an edge case where all counters have long queues except one. If many customers arrive simultaneously, describe the effect on the queue lengths and the processing efficiency. How does the queue system handle sudden spikes in customer arrivals?

Answer:
In this scenario, all arriving customers will join the shortest queue, causing a rapid increase in its length while the other queues slowly change in length.The system's efficiency may decrease momentarily as one counter becomes overloaded while the others process their queues. With more customers arriving and if the once shortest queue becomes long, all queues will moderately increase in length, and their processing efficiency, due to sudden spike, will with time decrease causing higher wait time, congestion, and high balance workloads.

To handle spikes, the system could employ load-balancing algorithms or temporarily open additional counters to accommodate extra customers. Processing efficiency can be maintained by ensuring that customers are spread across counters or by dynamically redirecting them if queues become imbalanced.

Question 4:
Design an algorithm to periodically balance the queue lengths by moving customers from longer queues to shorter ones. What factors should be considered to avoid disrupting the order in which customers are served? How would you ensure fairness while redistributing customers?

Answer:

Algorithm: BalanceQueueLengths

Input:

- A list of n queues, where each queue represents a checkout counter.
- Each queue has a function **size()** to get the current number of customers, and **enqueue(customer)** and **dequeue()** methods to add or remove customers.

Output:
Balanced queues where no queue is significantly longer than others.

---

1. **Initialize**:
    - Let **counters** be a list of queues.
    - Let **numCounters** be the number of queues in **counters**.
2. **Find Longest and Shortest Queues**:
    - Set **longestQueue** and **shortestQueue** to the first queue in **counters.**
    - For each queu**e Q** in **counters**:
        - If **Q.size()** is greater than **longestQueue.size()**, set **longestQueue** to **Q**.
        - If **Q.size()** is less than **shortestQueue.size()**, set **shortestQueue** to **Q**.
3. **Check for Imbalance**:
    - If the difference in size between **longestQueue** and **shortestQueue** is greater than 1, continue to Step 4.
    - Otherwise, **stop** (queues are already balanced).
4. **Transfer Customer**:

    - Remove the first customer from **longestQueue** by calling **longestQueue.dequeue()**.
    - Add this customer to **shortestQueue** by calling **shortestQueue.enqueue(customer)**.
5. **Repeat** (optional):
    - To maintain balanced queues over time, periodically repeat Steps 2–4 or call this algorithm in response to significant changes in queue lengths.

Question 5:

Imagine one of the checkout counters suddenly breaks down and stops serving customers. How would you handle the customers in that queue? Describe how you would redistribute them to other counters while maintaining order.

Answer:

In case one of the checkout counters breaks down suddenly, I would assign the customers of that counter(according to the order that they were going to be served at the previous counter), each(starting with the first from the previous) to the other queues, based on factors which are the shortest. If not that, then the one which handles the first instance of the customer number since it's most likely to shorten first; until all have been redistributed.

TASK 7: ALGORITHMS

1. **Can we have an algorithm to design an Algorithm?**

- Yes, we can have an algorithm to design algorithms, but the scope and effectiveness depend on the problem domain and the clarity of the requirements.
- Meta-algorithms are powerful but come with limitations in complexity and practicality.

2. **Having understood the algorithm, is it possible to automate the transcription of a document containing an algorithm which is still riddled with a certain amount of abstractness into a compilable source file?**

- It is possible to automate the transcription of abstract algorithm descriptions to compilable code to a certain extent, but this works best when the abstraction level is moderate and context is provided.

TASK 8: ASYMPTOTIC NOTATION

Efficient Matrix Merging Strategies

Introduction

Efficient matrix merging is a critical topic in the realm of data processing and computing, particularly as the size and complexity of datasets continue to grow. In many applications, particularly in fields such as machine learning, data analysis, and scientific computing, the ability to combine matrices efficiently can significantly impact overall performance and resource utilization.

One of the key aspects of matrix merging is space complexity, which refers to the amount of memory required to store the matrices during the merging process. As matrices can be large, understanding how to minimize space usage is vital to optimize performance and ensure that the system remains responsive. When merging multiple matrices, especially in environments with limited memory, efficient algorithms can help reduce the total space required, allowing for smoother computations and the ability to handle larger datasets.

Memory constraints also play a crucial role in the efficient merging of matrices. In many practical scenarios, the available memory may not be sufficient to hold all the matrices being merged. This situation necessitates the development of strategies that can operate within these constraints while still achieving optimal results. Techniques such as in-place merging or utilizing disk-based storage can provide solutions that allow for efficient processing without exceeding memory limits.

Finally, optimization techniques for sparse matrices are particularly significant in the context of matrix merging. Sparse matrices, which contain a high proportion of zero values, can benefit from specialized algorithms that focus on the non-zero elements. By leveraging the sparsity of these matrices, developers can design merging strategies that not only save memory but also enhance computational efficiency. This page will delve into these critical aspects, providing a comprehensive overview of the methods and considerations involved in efficient matrix merging.

Space Complexity of the Merged Matrix C

When merging two matrices, A of dimensions (m * n) and B of dimensions (p * q), it is essential to understand the implications of space complexity on memory requirements. The space complexity of the resultant merged matrix C is denoted by the formula:

[O(m * *n + p* * q)]

This formula signifies that the total memory required for the merged matrix C is proportional to the sum of the memory occupied by both matrices A and B.

Here, (m) and (n) represent the number of rows and columns of matrix A, while (p) and (q) represent the rows and columns of matrix B, respectively.

Implications of Space Complexity

The significance of this formula can be illustrated through an example. Consider matrix A with dimensions (3 * 4) and matrix B with dimensions (2 * 5). The space complexity for matrix A would be (3 * 4 = 12) and for matrix B it would be (2 *5 = 10). Thus, the total space complexity for the merged matrix C is:

[O(12 + 10) = O(22)]

This means that a total of 22 units of memory would be required to store the merged matrix.

Visualization Analogy

To visualize this concept, think of merging two rectangular plots of land. If plot A is a rectangle measuring 3 meters by 4 meters, it occupies a total area of 12 square meters. Similarly, if plot B measures 2 meters by 5 meters, it occupies 10 square meters. When combined, the total area of the new plot (the merged matrix C) would be the sum of the areas of both plots, leading to a requirement of 22 square meters of land. This analogy highlights how space complexity directly correlates to the dimensions and overall size of the matrices involved in the merging process.

Understanding this relationship is crucial for developers when designing systems that need to efficiently manage memory resources, especially in scenarios with large datasets or limited memory availability.

Merging Matrices with Limited Memory (Sequential Processing)

When faced with the challenge of merging matrices under restricted memory conditions, sequential processing emerges as a practical approach. This method involves processing one row or column of the matrices at a time, thus minimizing the memory footprint. By focusing on smaller segments of data, sequential processing efficiently manages memory and allows for the merging of larger matrices without requiring them to be wholly loaded into memory at once.

Step-by-Step Row Processing

In a typical sequential processing scenario, the merging begins with the first row of the first matrix, followed by the corresponding row in the second matrix. Each row is processed individually, with results being stored in a temporary output structure. Once a row is fully processed and stored, the next row is loaded, and the process repeats. This approach significantly reduces memory usage since only a small portion of the matrices is held in memory at any given time.

Example of Sequential Merging

For example, consider two matrices, A (2 rows, 3 columns) and B (2 rows, 3 columns). Using sequential processing, the first row of A and the first row of B is merged, and the result is stored in an output array. After processing the first row, the algorithm moves to the second row, merging it similarly. In this scenario, at no point are both matrices fully loaded into memory, allowing for efficient memory usage.

Trade-offs in Time and Space Complexity

While sequential processing offers clear advantages in memory efficiency, it comes with trade-offs in terms of time complexity. The approach may result in longer processing times due to the repeated loading and unloading of data. Instead of accessing the entire dataset at once, the algorithm must perform multiple read operations, which can slow down the overall merging process. This can be likened to managing a desk with limited space: rather than spreading out all your documents at once, you only take out one folder at a time. While this keeps your workspace manageable, it may slow down your ability to find and organize everything quickly.

In summary, sequential processing for merging matrices under memory constraints effectively balances the trade-offs of space and time complexity. By adopting a focused approach to data handling, it allows for the efficient merging of matrices while conserving valuable memory resources.

Optimization for Sparse vs. Dense Matrices

In the realm of matrix operations, the nature of the matrices involved—whether sparse or dense—significantly influences the optimization strategies employed. A sparse matrix is characterized by a predominance of zero values, while a dense matrix contains a high proportion of non-zero values. When merging or performing operations on a sparse matrix and a dense matrix, special techniques can be utilized to enhance both memory efficiency and computational speed.

One effective approach to handle sparse matrices is through the use of compressed storage formats. One of the most common formats is the Coordinate List (COO) format. In the COO format, only the non-zero entries of the matrix are stored along with their corresponding row and column indices. This drastically reduces the memory footprint needed to represent sparse matrices.

Example of COO Format

Consider a sparse matrix (A) of dimensions (4 *4):

```
A = | 0  0  3  0 |
    | 0  0  0  0 |
    | 0  1  0  0 |
    | 0  0  0  4 |
```

In COO format, this matrix would be represented as:

- Row indices: [0, 2, 3]
- Column indices: [2, 1, 3]
- Values: [3, 1, 4]

Thus, instead of using (16) entries (for a (4 * 4) matrix), the COO format only uses (3) entries, leading to a significant reduction in space complexity.

Space Complexity Principle

The principle of reduced space complexity is essential when working with sparse matrices. The space complexity for a sparse matrix in COO format can be approximated as (O(n)), where (n) is the number of non-zero elements. This contrasts with dense matrices, which typically require (O(m \times n)) space, where (m) and (n) are the dimensions of the matrix. This stark difference emphasizes the advantage of using sparse matrix representations when applicable.

In merging a sparse matrix with a dense matrix, the algorithms can be tailored to exploit the properties of both types. For instance, during the merging process, operations can skip zero entries in the sparse matrix, focusing only on non-zero elements. This results in reduced computational overhead and optimized performance, particularly in large datasets where memory and speed are critical constraints.

By leveraging compressed formats like COO and understanding the implications of space complexity, developers can create efficient algorithms that maximize performance while minimizing resource usage when dealing with disparate matrix types.

Total Space Complexity with Sparse Representation

When dealing with sparse matrices, particularly in conjunction with dense matrices, calculating the total space complexity becomes crucial for optimizing memory usage and performance. The formula that encapsulates this calculation is given as:

[O(nnz(A) + p * q)]

In this equation, (nnz(A)) represents the number of non-zero elements in the sparse matrix (A), while (p) and (q) are the dimensions of the dense matrix (B). This formula highlights how the space complexity is influenced by the number of non-zero elements in the sparse matrix, in contrast to the complete storage of dense matrices.

Implications of Non-Zero Elements

Sparse matrices are characterized by a significant proportion of zero values, which allows for optimization in storage. By focusing on the non-zero elements, memory can be conserved. For instance, if a sparse matrix has only a few non-zero entries, storing these values along with their indices can lead to substantial reductions in the overall space required. Conversely, dense matrices need to retain all their elements, which can lead to higher memory consumption.

When analyzing the total space complexity, the presence of non-zero elements in the sparse matrix directly affects performance. For instance, if matrix (A) has (nnz(A) = 5) for a (1000 * 1000) matrix, the space complexity becomes (

O(5 + 1000 * 1000)). Here, while the contribution from the sparse matrix is negligible compared to the dense matrix, it still plays a vital role in the overall calculation.

Optimizing Memory Usage

Understanding how to efficiently represent sparse matrices can significantly impact performance. Utilizing formats such as Compressed Sparse Row (CSR) or the aforementioned Coordinate List (COO) can help in minimizing space usage while maintaining quick access to non-zero elements. This optimization is especially crucial in applications where matrices are large, but only a fraction of the entries are meaningful for computations.

In conclusion, when merging matrices of differing types, particularly sparse and dense, it is essential to account for both the non-zero elements in the sparse matrix and the dimensions of the dense matrix. The formula (O(nnz(A) + p * q)) succinctly captures the essence of the total space complexity, guiding developers in creating efficient algorithms that optimize both memory and performance.

5. Merging Techniques with Sparse and Dense Matrices

When dealing with the merging of sparse and dense matrices, specific techniques can be employed to optimize performance and resource utilization. Two notable methods are "Row-by-Row Concatenation" and "Row-Wise Merging." Each technique has distinct processes and applications that can significantly affect the efficiency of matrix operations.

Row-by-Row Concatenation

Row-by-row concatenation involves appending the rows of one matrix directly to the rows of another. This method is straightforward and particularly effective when merging matrices of similar column dimensions. The process entails iterating through each row of both matrices and placing them into a new resultant matrix.

Example:

Consider two matrices (A) and (B):

```
A = | 1 2 3 |
    | 4 5 6 |
```

B = | 7 8 9 |
   | 10 11 12 |

The row-by-row concatenation results in:

C = | 1 2 3 |
   | 4 5 6 |
   | 7 8 9 |
   | 10 11 12 |

This technique is efficient for dense matrices, as it allows for a simple copy of rows into a new array. However, when merging a sparse matrix with a dense matrix, care must be taken to ensure that the empty rows of the sparse matrix do not lead to unnecessary memory consumption.

Row-Wise Merging

Row-wise merging is a more complex technique that entails processing rows from both matrices simultaneously. This method is particularly useful when the matrices have different column dimensions or when one matrix is sparse.

In row-wise merging, each row from the dense matrix is combined with the corresponding row from the sparse matrix, only including non-zero elements from the sparse matrix. The result is stored in a new structure that accommodates the varying number of entries.

Example:

Given a sparse matrix (A) and a dense matrix (B):

A = | 0 0 3 |
   | 0 4 0 |

B = | 1 2 3 |
   | 4 5 6 |

The row-wise merging would yield:

C = | 1 2 6 |
   | 4 9 6 |

In this process, the algorithm checks each element of the sparse matrix row for non-zero values, effectively skipping the zero entries and reducing memory usage.

Conclusion

The complexities involved in merging matrices cannot be understated, particularly when considering the interplay of space complexity, memory constraints, and the nature of the matrices being processed. Throughout this document, we have explored how the merging of matrices, especially under limited memory conditions, necessitates the application of strategic approaches to ensure optimal performance.

Key points include the importance of understanding space complexity, which is directly linked to the dimensions of the matrices being merged. The formula (O(m * n + p * q)) allows developers to anticipate the memory requirements for the resulting merged matrix, emphasizing the need for careful planning in environments with restricted resources.

The discussion around memory constraints has highlighted the efficacy of sequential processing methods. By processing one row or column at a time, developers can mitigate memory usage and effectively handle larger datasets without overwhelming system resources. This approach, however, brings forth trade-offs in time complexity, as multiple read operations can slow down the overall merging process.

Additionally, the optimization techniques presented for sparse versus dense matrices reveal the significant performance gains achievable through specialized algorithms. Utilizing compressed storage formats, such as COO, allows for a dramatic reduction in memory footprint, particularly in sparse matrices characterized by a high proportion of zero values.

Understanding these principles is not merely academic; they hold substantial significance in real-world applications. From machine learning models that rely on efficient data processing to scientific computations demanding high-performance algorithms, the strategies discussed provide foundational tools for developers seeking to maximize efficiency while navigating the constraints of modern computing environments.

TASK 9: BINARY TREES

Exploring Tree Traversals and Their Connection to Design Strategies: An In-Depth Analysis of Backtracking and Branch-and-Bound

Tree traversal is a foundational concept in computer science, used to systematically visit each node in a tree data structure. When traversing a tree, we rely on algorithms that allow us to explore the structure in various ways, each serving a unique purpose. Two primary types of tree traversal are Breadth-First Traversal (BFS) and Depth-First Traversal (DFS). Understanding these traversal techniques is essential, not only because of their individual applications but also because they form the basis of critical problem-solving strategies such as Backtracking and Branch and Bound. In this essay, we will examine BFS and DFS and analyze how they relate to these design strategies, revealing how these algorithms address different types of complex problems.

Understanding Tree Traversal Methods

Tree traversal involves visiting every node in a tree structure to perform operations or retrieve information systematically. The choice of traversal method impacts the sequence in which nodes are visited and processed, which can be crucial in applications that rely on finding specific elements or evaluating potential paths within a data structure.

1. Breadth-First Traversal (BFS)

Breadth-First Traversal, commonly abbreviated as BFS, is a traversal technique that explores a tree layer by layer, visiting all nodes at one level before moving down to the next. This process begins at the root node, followed by its immediate children, and continues across each subsequent level. BFS is typically implemented using a queue data structure, which ensures nodes are visited in the correct order. Each node is enqueued when first encountered, and nodes are dequeued as they are visited, making BFS a systematic, level-based approach to traversal.

The structure of BFS is especially suited to finding the shortest path in unweighted graphs, where all edges have equal weight. It is frequently used in applications like GPS navigation systems, where a solution to find the shortest path must be derived without considering the depth of each potential route. This makes BFS an optimal choice for exploring paths with a breadth-first expansion, where paths of minimal length are evaluated first.

2. Depth-First Traversal (DFS)

Depth-First Traversal, or DFS, contrasts with BFS by exploring as deeply as possible along each branch before backtracking. This traversal can be implemented using a stack data structure, which follows a last-in, first-out (LIFO) approach, or through recursion, which naturally supports the depth-first search pattern. DFS has three common forms:
- Pre-order (NLR): The node is visited first, followed by its left subtree and then its right subtree.
- In-order (LNR): The left subtree is visited first, then the node itself, followed by the right subtree.
- Post-order (LRN): The left subtree and right subtree are visited before the node itself.

DFS is well-suited for tasks where each possible path must be fully explored before moving to the next, such as in puzzle-solving scenarios where all potential solutions are evaluated. It is also effective in situations requiring in-depth analysis of subtrees, such as evaluating syntax trees in compilers, where each level must be assessed completely before moving on.

Connecting BFS and DFS to Design Strategies

The principles of BFS and DFS can be extended to two primary design strategies: Backtracking and Branch and Bound. These strategies apply BFS and DFS methodologies to solve complex computational problems effectively.

Backtracking and Its Relation to Depth-First Traversal

Backtracking is a problem-solving technique that builds solutions incrementally, rejecting solutions as soon as it determines they cannot yield valid results. In backtracking, partial solutions are developed step-by-step, and if a partial solution does not satisfy the problem constraints, the algorithm "backtracks" to a previous decision point to try a different path. Backtracking is inherently a Depth-First Search approach, as it explores each potential solution path as far as possible before backtracking to explore alternative options.

The DFS methodology is essential for backtracking because it allows for a deep exploration of each path. This approach is particularly effective in scenarios where the solution space is vast, but only a few paths lead to valid solutions. For instance, in the classic N-Queens problem, backtracking places queens on a chessboard one by one, checking if each placement is safe. If a placement results in a conflict, the algorithm backtracks to place the queen in a new position, continuing this depth-first approach until a solution is found. Similarly, backtracking can be seen in solving puzzles like Sudoku, where each number placement along a row, column, and grid must be validated before moving to the next cell.

Branch and Bound and Its Relation to Breadth-First Traversal

Branch and Bound is a strategy used for optimization problems, aiming to find the best possible solution based on a specific criterion (such as minimum cost or maximum profit). The strategy works by exploring branches of potential solutions but "bounding" or discarding paths that do not yield optimal outcomes. While Branch and Bound can sometimes be implemented with a DFS approach, it often benefits from a Breadth-First Search structure that allows the exploration of solutions level-by-level, particularly in optimization scenarios.

In the Branch and Bound technique, BFS is often used to evaluate multiple solution paths concurrently by progressing from one level to the next in a systematic, breadth-first fashion. For example, in the traveling salesman problem (TSP), each path represents a potential route. As paths are explored, the algorithm calculates the cost associated with each branch and abandons branches that exceed the cost of the best-known solution. This bounding mechanism prevents unnecessary exploration of non-promising solutions, significantly reducing computational time in large search spaces.

Branch and Bound also proves advantageous in knapsack problems, where the goal is to maximize the value of selected items without exceeding a weight limit. Each item is considered in terms of its value-to-weight ratio, and branches that exceed the weight limit or fall below a certain value threshold are bounded, ensuring that only viable paths are explored.

TASK 10: SEARCHING ALGORITHMS

Step-by-Step Breakdown of Binary Search Complexities

Introduction
This task provides a detailed breakdown of the time complexities of Binary Search, explaining how Big O, Big Omega, and Big Theta notations are derived for the algorithm.

Step 1: Understanding the Process of Binary Search
1. **Divide and Conquer:** Binary Search works by dividing the sorted array in half with each comparison.

2. **Compare Middle Element:** We check if the target value matches the middle element of the array.
   - If it matches, we stop, as we've found the target.
   - If the target is smaller, we repeat the process on the left half.
   - If the target is larger, we repeat on the right half.

3. **Repeat Until Found or Not Found:** This process continues until either the target is found, or there are no elements left to search.

Step 2: Establishing the Execution Time Equation
The execution time \(T(n) \) for Binary Search can be expressed with the recurrence relation:

T(n) = T(n/2) + O(1)

- T(n): Time to search in an array of n elements.
- T(n/2): Time to search in half the array (since each step cuts the array size in half).
- O(1): Constant time required to make one comparison between the middle element and the target.

This recurrence relation means that with each step, Binary Search reduces the problem size by half, adding a constant time operation at each division.

Step 3: Solving the Recurrence Relation for Complexity
Each step halves the size of the search space. Expanding T(n) a few times shows the pattern:

1. First Division: $T(n) = T(n/2) + O(1)$
2. Second Division: $T(n/2) = T(n/4) + O(1)$, so $T(n) = T(n/4) + O(1) + O(1)$
3. Third Division: $T(n/4) = T(n/8) + O(1)$, so $T(n) = T(n/8) + O(1) + O(1) + O(1)$

This pattern shows we are adding O(1) a logarithmic number of times, specifically log2(n) times, resulting in:

$T(n) = O(\log n)$

Step 4: Interpreting Big O, Big Omega, and Big Theta Notations

Big O Notation (O) - Worst Case Complexity
Big O gives us the upper bound, showing the maximum time taken. For Binary Search, the worst case occurs when the target is not found or is at the end, requiring O(log n) steps:

$T(n) = O(\log n)$

Big Omega Notation (Ω) - Best Case Complexity
Big Omega provides the lower bound, showing minimum time taken. For Binary Search, the best case occurs when the target is in the middle, requiring only one comparison, giving:

$T(n) = \Omega(1)$

Big Theta Notation (Θ) - Tight Bound Complexity
Big Theta provides a tight bound, representing growth of T(n) with Θ(log n) in all cases, making:

$T(n) = \Theta(\log n)$

Step 5: Identifying Best, Worst, and Average Cases for Binary Search

Best Case

- Condition: Target is at the middle of the array.
- Complexity: O(1)

Worst Case

- Condition: Target is either not in the array or located at one of the edges.
- Complexity: O(log n)

Average Case

- Condition: Target could be in any position, uniformly distributed.
- Complexity: O(log n)

Summary

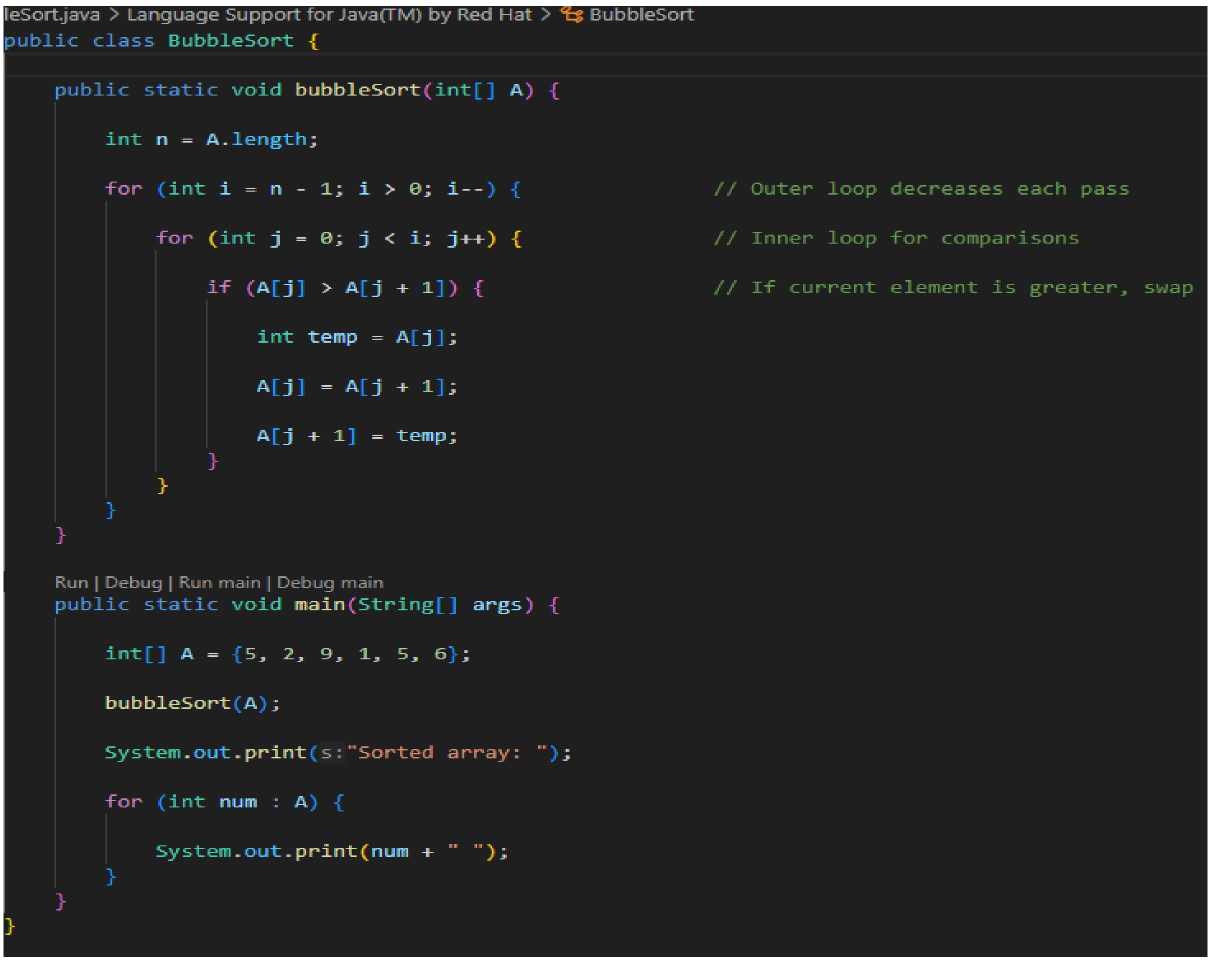
- **Binary Search:** Efficient for finding an element in a sorted array.
- **Execution Time Equation:** $T(n) = T(n/2) + O(1)$.
- **Time Complexities:**
  - Big O (Worst Case): $O(\log n)$
  - Big Omega (Best Case): $\Omega(1)$ or $\Omega(\log n)$
  - Big Theta (Tight Bound): $\Theta(\log n)$

TASK 11: SORTING ALGORITHMS

1. Implementation of bubble sort in C and in Java

Pseudocode

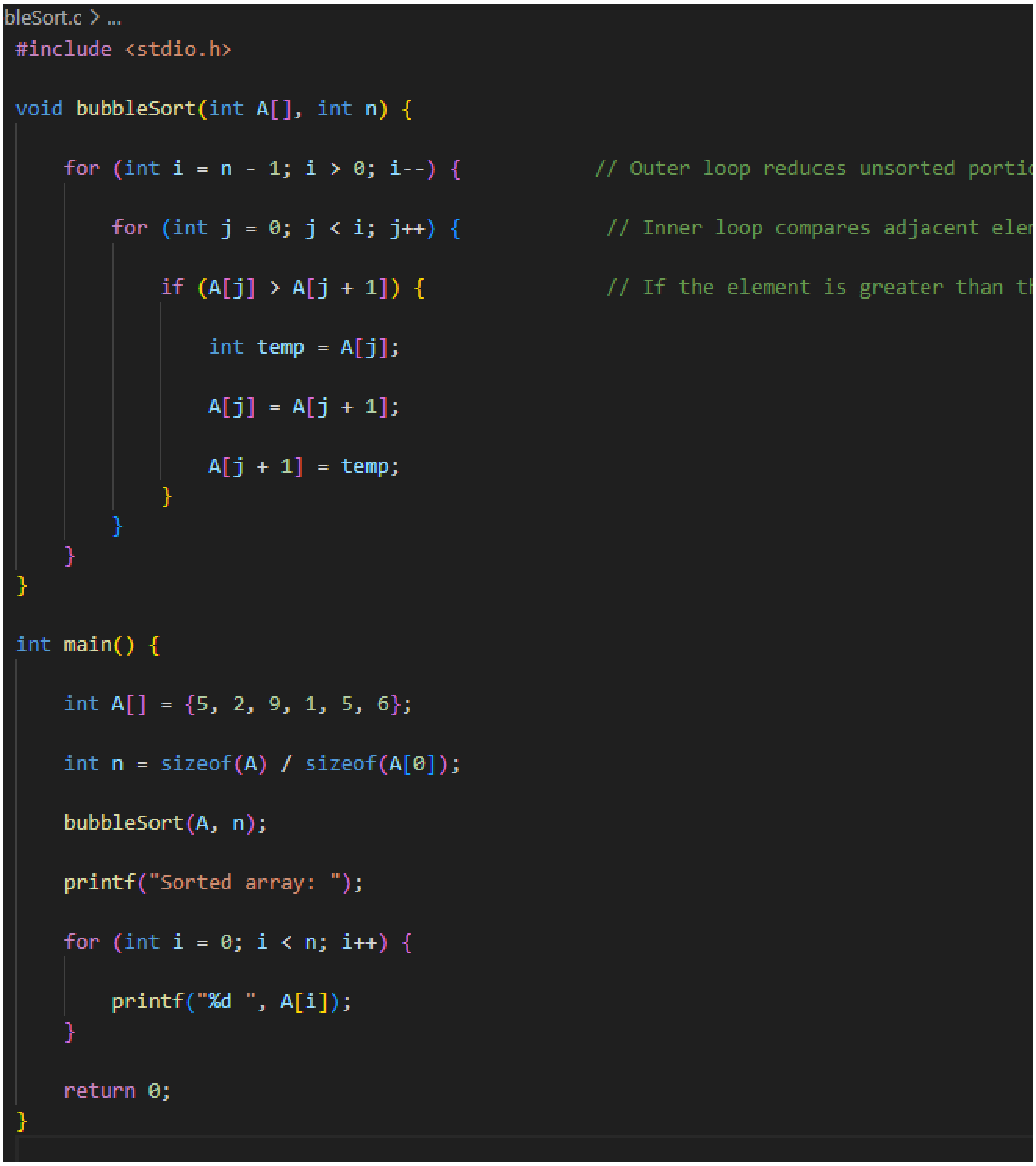
```
Bubble(A[n]){
 For(i=n-1 to 0)     // runs from the end of the array toward the beginning
 {
   For(j=0 to i-1) //i++     // runs up the current i
      If(A[j]>A[j+1] {       // checks if the current element is larger than the next
         Swap(A[j] ,  A[j+1])        // if so, it swaps the elements
      }
}
```



leSort.java > Language Support for Java(TM) by Red Hat > BubbleSort

```java
public class BubbleSort {

    public static void bubbleSort(int[] A) {

        int n = A.length;

        for (int i = n - 1; i > 0; i--) {                 // Outer loop decreases each pass

            for (int j = 0; j < i; j++) {                 // Inner loop for comparisons

                if (A[j] > A[j + 1]) {                    // If current element is greater, swap

                    int temp = A[j];

                    A[j] = A[j + 1];

                    A[j + 1] = temp;
                }
            }
        }
    }

    Run | Debug | Run main | Debug main
    public static void main(String[] args) {

        int[] A = {5, 2, 9, 1, 5, 6};

        bubbleSort(A);

        System.out.print(s:"Sorted array: ");

        for (int num : A) {

            System.out.print(num + " ");
        }
    }
}
```

Figure 15: Java code showing implementation of bubble sort



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

void bubbleSort(int A[], int n) {

    for (int i = n - 1; i > 0; i--) {           // Outer loop reduces unsorted portic

        for (int j = 0; j < i; j++) {            // Inner loop compares adjacent eler

            if (A[j] > A[j + 1]) {               // If the element is greater than th

                int temp = A[j];

                A[j] = A[j + 1];

                A[j + 1] = temp;
            }
        }
    }
}

int main() {

    int A[] = {5, 2, 9, 1, 5, 6};

    int n = sizeof(A) / sizeof(A[0]);

    bubbleSort(A, n);

    printf("Sorted array: ");

    for (int i = 0; i < n; i++) {

        printf("%d ", A[i]);
    }

    return 0;
}
```

Figure 16: C code showing the implementation of bubble sort

2. Understanding of bubble sort in terms of space and time complexity.

Time Complexity

- **Worst-case scenario O(n^2):** Each element must be compared with every other element, resulting in a quadratic number of comparisons and swaps. This happens if the array is initially in reverse order

- **Best-case scenario O(n):** When the array is already sorted, Bubble Sort makes only one pass through the array without any swaps. This requires only n comparisons, making the best-case scenario linear.

Space Complexity

- **Worst-case scenario O(n^2):** This happens when the array is sorted in reverse order, as every element requires comparison and swapping with every other.
- **Best-case scenario O(n):** When the array is sorted, Bubble Sort makes only one pass without swaps, as it can recognize the order.
- **Average case O(n):** In most cases, Bubble Sort will perform close to the worst-case number of operations.

3. **Explain why the worst case scenario is O(n2) and the best case scenario is O(n) in bubble sort.**

Worst-case scenario

- The **worst-case scenario** for Bubble Sort is when the array is sorted in **reverse order**. Bubble Sort will need the **maximum number of swaps and comparisons** to sort it in ascending order because each element has to "bubble up" through all the other elements.
- Every time an element moves up one position, Bubble Sort has to go through the entire unsorted part of the array, which requires ≈n^2 / 2 comparisons and swaps.

$$(n-1) + (n-2) + (n-3) + \cdots + 1$$

Sum: n(n−1)\2

- The sum n(n−1)\2 grows **quadratically** with n, meaning that as n increases, the number of comparisons and swaps needed in the worst case increases roughly as n^2 which gives us a **time complexity of O(n^2)** in the worst case.

Best-case scenario

- If the array is **already sorted** in ascending order, we don’t need to make any swaps.
- However, the algorithm will still go through the array to check if any pairs are out of order, but since it finds everything in order on the first pass, it can stop early by using a **flag** (aka swapped) to keep track of whether any swaps were made during each pass.
- If the algorithm completes a full pass with no swaps, it knows the array is already sorted and can terminate.

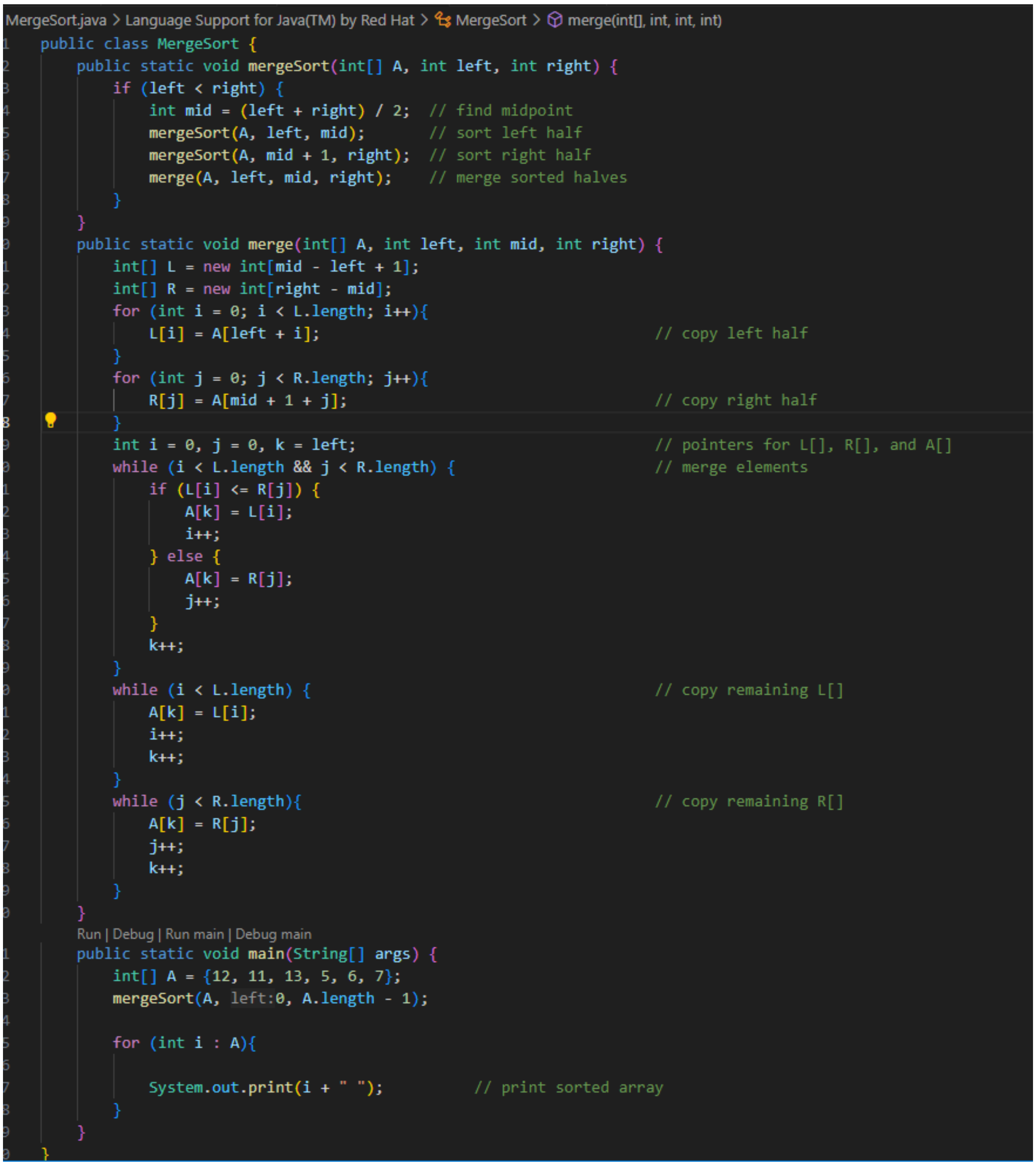
- During this pass, it performs n−1 comparisons but **no swaps**.
- Since it only requires a single pass through the array, the best-case time complexity is **linear**, or O(n). Therefore, the **total number of operations** is proportional to the size of array, which is O(n) in terms of time complexity.

4. Do merge sort for sorted and unsorted datasets and provide their space and time complexity.

Pseudocode

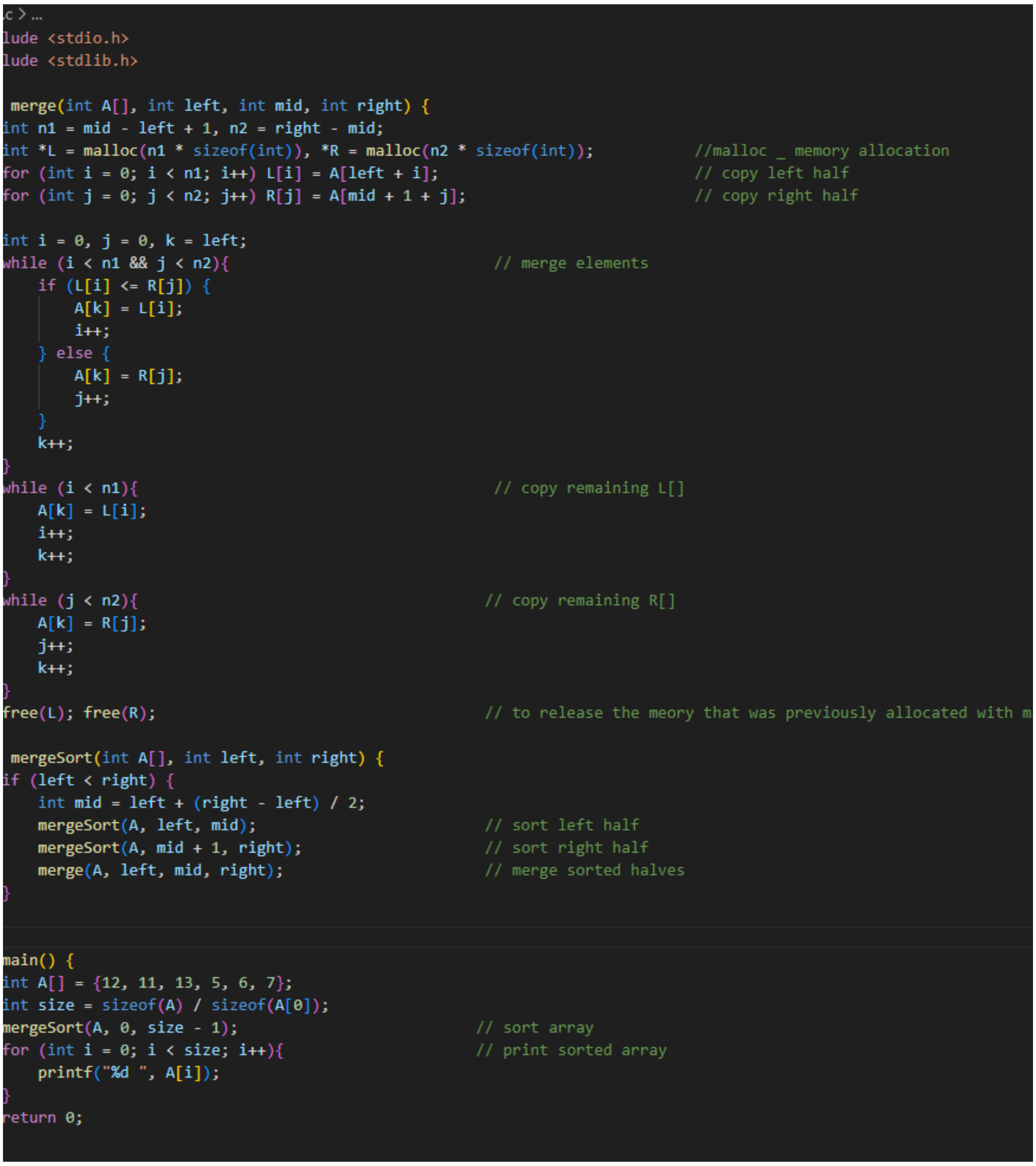
```
MergeSort(A, left, right)
{
  if (left < right){            // checks if there's  more than one element in array
    mid = (left + right)              // find the midpoint
    MergeSort(A, left, mid)                 // recursively sort the left half
    MergeSort(A, mid + 1, right)            // recursively sort the right half
    Merge(A, left, mid, right)              // merge the two sorted halves
  }
}
Merge(A, left, mid, right)
{
  create temporary arrays L[] and R[]              // for left and right halves
  copy elements from A[left...mid] into L[]                 // populate L[] with left half
  copy elements from A[mid+1...right] into R[]   // populate R[] with right half
i = 0, j = 0, k = left                              // initialize pointers for L[], R[], and A[]
 while (i < size of L[] and j < size of R[])     // merge elements back into A[]
  {
    if (L[i] <= R[j]) {                    // if current element in L[] is smaller or equal
        A[k] = L[i]                    // place L[i] in A[k]
        i = i + 1              // move pointer in L[]
    }
     else {
        A[k] = R[j]            // place R[j] in A[k]
        j = j + 1              // move pointer in R[]
```

```
            k = k + 1           // move pointer in A[]
        }
        copy remaining elements of L[] into A[]         // if any elements left in L[]
        copy remaining elements of R[] into A[]         // if any elements left in R[]
    }
```



```
MergeSort.java > Language Support for Java(TM) by Red Hat > MergeSort > merge(int[], int, int, int)
public class MergeSort {
    public static void mergeSort(int[] A, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;  // find midpoint
            mergeSort(A, left, mid);       // sort left half
            mergeSort(A, mid + 1, right);  // sort right half
            merge(A, left, mid, right);    // merge sorted halves
        }
    }
    public static void merge(int[] A, int left, int mid, int right) {
        int[] L = new int[mid - left + 1];
        int[] R = new int[right - mid];
        for (int i = 0; i < L.length; i++){
            L[i] = A[left + i];                                 // copy left half
        }
        for (int j = 0; j < R.length; j++){
            R[j] = A[mid + 1 + j];                              // copy right half
        }
        int i = 0, j = 0, k = left;                             // pointers for L[], R[], and A[]
        while (i < L.length && j < R.length) {                  // merge elements
            if (L[i] <= R[j]) {
                A[k] = L[i];
                i++;
            } else {
                A[k] = R[j];
                j++;
            }
            k++;
        }
        while (i < L.length) {                                  // copy remaining L[]
            A[k] = L[i];
            i++;
            k++;
        }
        while (j < R.length){                                   // copy remaining R[]
            A[k] = R[j];
            j++;
            k++;
        }
    }
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {
        int[] A = {12, 11, 13, 5, 6, 7};
        mergeSort(A, left:0, A.length - 1);

        for (int i : A){

            System.out.print(i + " ");          // print sorted array
        }
    }
}
```

Figure 17: Java code showing implementation of merge sort



```
c > ...
lude <stdio.h>
lude <stdlib.h>

 merge(int A[], int left, int mid, int right) {
int n1 = mid - left + 1, n2 = right - mid;
int *L = malloc(n1 * sizeof(int)), *R = malloc(n2 * sizeof(int));           //malloc _ memory allocation
for (int i = 0; i < n1; i++) L[i] = A[left + i];                            // copy left half
for (int j = 0; j < n2; j++) R[j] = A[mid + 1 + j];                         // copy right half

int i = 0, j = 0, k = left;
while (i < n1 && j < n2){                               // merge elements
    if (L[i] <= R[j]) {
        A[k] = L[i];
        i++;
    } else {
        A[k] = R[j];
        j++;
    }
    k++;
}
while (i < n1){                                         // copy remaining L[]
    A[k] = L[i];
    i++;
    k++;
}
while (j < n2){                                        // copy remaining R[]
    A[k] = R[j];
    j++;
    k++;
}
free(L); free(R);                                      // to release the meory that was previously allocated with m

 mergeSort(int A[], int left, int right) {
if (left < right) {
    int mid = left + (right - left) / 2;
    mergeSort(A, left, mid);                           // sort left half
    mergeSort(A, mid + 1, right);                      // sort right half
    merge(A, left, mid, right);                        // merge sorted halves
}

main() {
int A[] = {12, 11, 13, 5, 6, 7};
int size = sizeof(A) / sizeof(A[0]);
mergeSort(A, 0, size - 1);                            // sort array
for (int i = 0; i < size; i++){                       // print sorted array
    printf("%d ", A[i]);
}
return 0;
```

Figure 18: C code showing implementation of merge sort

Time Complexity

- Merge Sort consistently divides the array in half, and merging these halves back takes a linear amount of time for each level of recursion.
- At each level, the array is split into two halves (logarithmic division), leading to a height of $O(\log n)$.
- Each level's merge operation requires $O(n)$ comparisons and assignments, regardless of whether the data is sorted.
- **Divide Step**: Each division step splits the array into two halves, creating a recursive tree structure with $\log_2(n)$ levels.
- **Merge Step**: At each level, merging all elements requires a total of $O(n)$ time, as each element is compared and combined with another.
- Since there are $\log_2(n)$ levels, the total time complexity is $O(n\log n)$.
- **Overall Time Complexity**: $O(n\log n)$ for best, worst, and average cases.

Space Complexity

- Merge Sort requires **additional space** for the temporary arrays used during the merging process.
- The additional space usage comes from creating a new array (or arrays) of the same size as the input array to hold elements during the merge.
- **Temporary Storage**: For each merge operation, we create two temporary arrays L[] and R[] to store the left and right halves of the array, leading to $O(n)$ additional space at each recursive call.

- **Recursive Stack Space**: Since we perform recursive calls, the **call stack** depth goes up to O(log⁡n) levels. However, this space is not additive since each call completes before the next level's merge operations proceed.
- **Overall Space Complexity**: O(n) as the algorithm needs extra space for merging the halves.

<u>**Summary**</u>

Dataset Type	**Time Complexity**	**Space Complexity**
Unsorted	O(nlogn)	O(n)
Sorted	O(nlogn)	O(n)

TASK 12: HASHING

- In hashing, **probing** is a technique used to handle **collisions** (i.e., situations where two different keys hash to the same index in a hash table). When a collision occurs, probing helps find an alternative location for the new entry in the table.
- Main types:
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

LINEAR PROBING

- When a collision occurs, linear probing checks the next available slot in the hash table by moving linearly.
- For example, if a key hashes to index **i** but that index is occupied, linear probing will check **i+1**, **i+2**, and so on (and wraps around if it reaches the end of the table).
- **Formula**: new_index = **(i + j)** % table_size, where **j** increments by 1 each time until an open slot is found.

QUADRATIC PROBING

- **Quadratic probing** checks indices at increasing intervals based on a quadratic function.

- This technique reduces clustering (multiple keys clumping together), which can be an issue in linear probing.
- **Formula**: new_index} = **(i + j^2)** % table_size, where **j** increments by 1 each time.

DOUBLE HASHING

Double hashing uses a secondary hash function to determine the probe sequence, making it more effective at spreading out entries and reducing clustering.

- **Formula**: new_index} = **(i + j.h2(key))** % table_size, where h2(key) is a second hash function.

Importance of Probing

- Probing allows hash tables to remain relatively compact and efficient.
- Instead of expanding the table for every collision, probing techniques make it possible to use the available space effectively by finding alternative slots for colliding keys.

Example: Linear Probing

Consider a hash table of size 7 and a hash function h(key) = key % 7. If we try to insert keys `10`, `17`, and `24`, they all hash to index `3` (since 10 % 7 = 3, 17 % 7 = 3, and 24 % 7 = 3)

With **linear probing:**

1. Insert `10` at index `3`.
2. Insert `17` at index `4` (next available slot).
3. Insert `24` at index `5` (next available slot).

This way, each collision is resolved by moving to the next slot until a free one is found.