COL106 : Data Structures and Algorithms, Semester II 2024-25
Practice Programming Questions on Lists

January 2025

Instructions

- Please use the following questions as practice questions for learning about List datastructure.
- The questions with * next to them should be attempted during the lab sessions, and your solutions must be uploaded on moodlenew. Note that your submissions will not be evaluated, but will be used as a mark of your attendance. We will filter out all the submissions that are not from the lab workstations. So do not use your laptops for submitting the programs.

1 Questions

1. **Let's Build a Text Editor, Like VS Code (But Cooler!)**

   Ever struggled with buggy text editors during your lab assignments? Well, let's build our own! Since we're in COL106 (and clearly pros at data structures), why not create a text editor using good ol' linked lists? (Cooler than it sounds, trust me!)

   I'm personally a big fan of **VS Code** — lightweight, feature-packed, and perfect for editing all kinds of code. But we're about to make something even better (or at least as cool as a linked list can get). Here's what you need to do:

   Start With an Empty File:
   We begin with an empty file and a **cursor** that can move around and edit the content. You can **add** text at the cursor's position or **delete** text from there. Simple enough, right?

   Pro Tip: You're not a machine, so we won't have you typing at the speed of light (that would be crazy!). But let's imagine you're an editing machine, and for this, we're using a linked list to store text. This means you can add or delete text anywhere, not just at the end, which would be **super slow** if you used arrays.

   Methods to Implement:
   You will need to implement the following methods for your text editor:

   (a) **Create(filename)**:
   - Create and store the file name for reference.
   - Set up your text editor's initial state (empty file, cursor at the start).

   (b) **Write(txt)**:
   - Add the text at the **current cursor position** (not just the end, so we avoid that pesky performance hit we'd get with arrays).

   (c) **Delete(k)**:

- Delete up to **k characters** to the left of the cursor. Remember, the cursor doesn't have access to all of the text — it can only delete from where it's currently sitting.

(d) **MoveCursor(k)**:

- Move the cursor **k characters ahead** if $k > 0$ or **k characters back** if $k < 0$. But don't let it fall off the edge — the cursor can't move beyond the text.

The Fun Part: Version Control:
Here comes the twist — **Version Control**. Why? Because our code is precious, and we need to keep track of every change!

So, here's how we'll implement a **Save()** function:

- Each time you feel like your text is a masterpiece and needs saving, call `Save()`.
- When you call `Save()`, create a copy of the current text.
- The file name should follow this format: `filename-(version_number)`. For example:
  - After the first save: `Foo.java-(1)`
  - After the second save: `Foo.java-(2)`
  - And so on.
- Every time you save, you increment the version number. You don't need to worry about saving multiple versions recursively for now (just keep it simple).

This will simulate how real-world editors (like VS Code) use version control to track changes and provide history, all without needing an entire Git setup.

Bonus (Future Fun Stuff):
Once you're comfortable with the basics, feel free to extend the editor with more features:

- **Undo/Redo**: Imagine undoing that one typo or redo-ing an edit!
- **Find and Replace**: Perfect for dealing with that one annoying bug in your code.

But for now, focus on getting the core features working smoothly.

Summary of Tasks:
You need to implement the following methods:

- `Create()`, `Write()`, `Delete()`, `MoveCursor()`, and `Save()`.
- Save versions of your text in a `filename-(version_number)` format.
- Have fun with it! And remember, just like real editors, your creation should be **snappy** and **efficient**.

Additional Tips:

- Make sure your **linked list** is the backbone of your editor. The cursor and operations should respect the nature of a linked list — don't cheat by using arrays!
- Think of the linked list nodes as "text chunks," where each node stores a part of the text.

Extra Challenge!
Got a better idea? If you can think of a more efficient way to implement the same features without using linked lists, let us know! I'll be more than happy to treat you personally to a snack or coffee (or both)!

Want to Learn Git? If you're interested in learning more about version control and Git, which is used by real-world editors, here's a resource that might help:

- **Git Tutorial for Beginners**: click here

2. * **Merge K Sorted Linked Lists**

   You are given an array of K sorted linked lists. Your task is to merge all the linked lists into one final sorted linked list.

   Example:

   - **Input:**
     $$\text{Lists} = \{\{1, 2, 5\}, \{3, 6\}, \{2, 9\}\}$$
   - **Output:**
     $$\{1, 2, 2, 3, 5, 6, 9\}$$

   Constraints:

   - $1 \leq K \leq 10^4$
   - Total number of nodes across all linked lists: $1 \leq N \leq 10^6$
   - Elements in each linked list are sorted in ascending order.

   Accepted Solution:

   - **Time Complexity:** $O(N \cdot \log K)$, where N is the total number of nodes and K is the number of linked lists.
   - **Space Complexity:** $O(1)$.

3. * **Rotate List by *k* Steps**

   Given an array of integers `nums` and a non-negative integer `k`, rotate the array to the right by `k` steps. The rotation should be performed *in place*, i.e., without the usage of extra memory, and should have $O(|\texttt{nums}|)$ time complexity.

   Implementation: Construct a class `RotateArray` containing a publicly accessible method `void rotateArray(int[] nums, int k)`. This method should rotate `nums` as required.

   Example:

   - **Input:**
     `nums = {1, 2, 3, 4, 5, 6, 7}`
     `k = 3`
   - **Result:**
     `nums = {5, 6, 7, 1, 2, 3, 4}`

   Things to note:

   - `k` may be greater than the length of the array.
   - Does Java use pass-by-value or pass-by-reference for arguments? What do you understand by this?

4. **Skip List** So far, you might be familiar with the concept of a linked list, a dynamic data structure where each node is connected to the next via a pointer. Linked lists are great for dynamic memory allocation, and they allow elements to be inserted or removed efficiently. However, searching for an element in a linked list can be slow. To find a specific value, you must traverse the list sequentially, resulting in a time complexity of O(n), even if the list is sorted.

Compare this to a sorted array: using binary search, you can quickly locate an element in O(log n) by jumping to the middle, narrowing the search space each time. But linked lists don't have direct access to indices, so this isn't possible.

Now, what if we could add shortcuts to our linked list? These shortcuts would let us skip over multiple nodes during a search, enabling faster traversal. This idea leads us to the **skip list**, a probabilistic data structure that introduces multiple levels of linked lists stacked on top of the base list.

A skip list is essentially a linked list with additional levels that act as "express lanes." The base level contains all the nodes, just like a standard linked list. The next level up contains fewer nodes, skipping some from the base level, and higher levels skip even more nodes. When searching for an element, you start at the highest level, quickly narrowing down the range before dropping down to lower levels for precise traversal. This design enables an average time complexity of O(log n) for search, insertion, and deletion.

In this assignment we will be implementing the Skiplist class as given below.

```
class ListNode {
    int val;
    ListNode next, down;

    public ListNode(int val) {
        // complete
    }
}

public class Skiplist {
    private List<ListNode> levels; // List of levels
    private Random random; // Random generator

    public Skiplist() {
        // complete
    }

    public boolean search(int target) {
        // complete
    }

    public void add(int num) {
        // complete
    }

    public boolean erase(int num) {
        // complete
    }
}
```

Listing 1: Skiplist Class Implementation

Note : we are using random number generator to decide whether the num to be added in the higher level or not

You can read more about skip list here

5. **Problem: Linked List and Grid Traversal**

   (a) **Finding the Middle Element of a Linked List in One Pass**
   Given a singly linked list, write a function to find the middle element in one pass. If the list has

an even number of elements, return the second middle element.

Constraints:

- Use $O(1)$ extra space only.
- Time complexity: $O(n)$.

Input: The head node of the linked list.
Output: The value of the middle node.

(b) **Detecting a Cycle in a Linked List in $O(n)$**

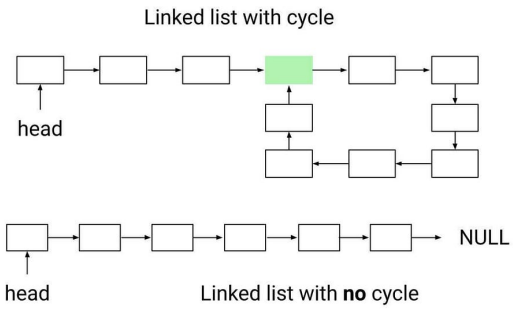

Figure 1: Linked List with cycle

Extend your implementation from part (a) to detect whether the linked list contains a cycle in $O(n)$ time

Constraints:

- Use $O(1)$ extra space only.
- Time complexity: $O(n)$.

Input: The head node of the linked list.
Output: Return `True` if a cycle is detected, otherwise `False`.

Up for a Challenge?!: Find the Intersection point of linear and circular parts of Linked List.

(c) **Grid with Arrows - Reaching the Bottom-Right Cell**
You are given an $n \times n$ grid, where each cell contains an arrow indicating a direction (`up`, `down`, `left`, or `right`). Starting at the top-left cell $(1, 1)$, determine if it is possible to reach the bottom-right cell (n, n). If it is impossible to reach the bottom-right cell by any reasons, return `False`.

Constraints:

- Time complexity: $O(n^2)$.
- Extra Space: $O(1)$.

Input:

- An integer n, the size of the grid.
- A 2D list of Characters, where each string represents the direction of the arrow in that cell (e.g., `L`,`R`,`D`,`U`).

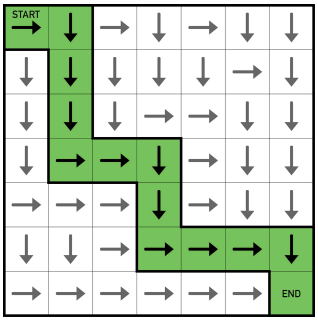

Figure 2: Grid with arrows

Output: Return `True` if it is possible to reach (n, n), otherwise return `False`.

6. **Operations on integers using Linked Lists**

   (a) **Representing an integer as a Linked List**
   Given a signed integer x, represent it as a singly linked list where each node contains one digit of x, starting from the least significant digit (rightmost digit) to the most significant digit (leftmost digit). Implement this functionality in a class called `Number` consisting of nodes belonging to a class called `Digit`. Remember to store the sign of x.
   Input:
   - A signed integer x (e.g., 123).

   Output:
   - A linked list representation of x (e.g., $3 \to 2 \to 1$ for 123).

   (b) **Comparison of two integers using Linked Lists**
   Implement a method in the `Number` class to compare the stored integer x to another integer y represented as a linked list. The method should return 1 if x is greater, -1 if y is greater, and 0 if they are equal.

   Input:
   - A linked list representing integer y.

   Output:
   - An integer (1, -1, or 0) representing the comparison result.

   Constraints:
   - **Time Complexity:** $O(N + M)$, where N and M are the number of digits of x and y, respectively.
   - No extra space should be used in arrays or other lists.

   (c) **Addition of two integers using Linked Lists**
   Given an integer y represented as a linked list (as described in **(a)**), implement a method in the `Number` class to perform addition to the stored integer x. The resulting sum should also be represented as a linked list. Remember to account for the sign.

   Input:
   - A linked list representing integer y.

   Output:

- A linked list representing the sum of x and y.

Constraints:

- **Time Complexity:** $O(N + M)$, where N and M are the number of digits of x and y, respectively.
- **Space Complexity:** $O(N + M)$, where N and M are the number of digits of x and y, respectively.
- No extra space should be used in arrays or other lists.
- No trailing zeroes should be present in the output.

(d) **Representation of stored integer**
Implement methods `outputAsString` and `outputAsInteger` in the `Number` class to output the stored integer x as a `String` or `int`. The integer stored in `Number` may be too large to output as an `int`. Hence, output $x \mod 10^9$.

Input:

- None.

Output:

- A `String` representing the stored integer or an `int` representing the stored integer $\mod 10^9$.

Constraints:

- **Time Complexity:** $O(N)$, where N is the number of digits of x.
- No extra space should be used in arrays or other lists.

7. **Snake Game Simulation Using Linked Lists**

You are tasked with implementing a snake game simulation on an $n \times n$ grid. Snake body can be modelled as a Contiguous cells in the grid(Need not be in a single line, can have turns). The snake(its head) starts at position $(1, 1)$, facing right, with an initial length of 1. The game consists of the following functionalities:

(a) **Move Snake and Collision Detection:**
Move the snake one step in the current direction. After each move, check for:

- **Boundary collisions** (snake hitting the grid edges).
- **Self-collisions** (snake head colliding with its body).
- **Apple** See (c)

(b) **Change Snake's Direction:**
The user can input commands to change the direction of the snake (up, down, left, right). Reversing direction is not allowed.

(c) **Snake Growth Upon Eating an Apple:**
When the snake eats an apple, its length increases by one, at the tail. On eating an apple, new apple appear at random positions, which could generated by a provided function.

Starter Code

The following starter code is provided, with certain functions left for you to implement. The function to generate a random apple position is included.

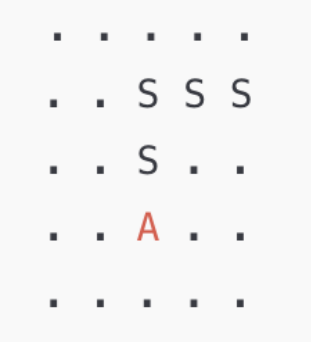

Figure 3: Snake Game

Java Code

```
import java.util.Random;

public class SnakeGame {
    private int n; // Grid size
    private ____ snake; // Snake's body
    private int[] apple; // Apple position
    private int[][] directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}}; // Right, down, left, up
    private int currentDirection = 0; // Start direction (0 = right)

    public SnakeGame(int n) {
        this.n = n;
        //Initialise Snake parameters, and initial apple position
    }

    // Function to generate a random apple position
    private void generateApple() {
        Random rand = new Random();
        int x, y;
        do {
            x = rand.nextInt(n) + 1;
            y = rand.nextInt(n) + 1;
        } while (isOccupiedBySnake(x, y)); // Ensure apple doesn't spawn on the snake
        this.apple = new int[]{x, y};
    }

    // TODO: Check if a position is occupied by the snake
    private boolean isOccupiedBySnake(int x, int y) {
        //Implement this
    }

    // TODO: Implement the function to change the direction of the snake
    public void changeDirection(int newDirection) {
        // Ensure new direction is not directly opposite
    }

    // TODO: Implement the function to move the snake
```

```
    public boolean moveSnake() {
        // Move the snake, grow if needed, and check for collisions
        return true; // Return false if collision occurs
    }

    // TODO: Implement collision detection logic
    private boolean isCollision(int newX, int newY) {
        // Check for boundary and self-collision
        return false;
    }

    // Print game state for debugging
    // TODO: Set position of snake by changing the relevant indices to S.
    public void printState() {
        char[][] grid = new char[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) grid[i][j] = '.';
        }

        // SET POSITION OF SNAKE

        grid[apple[0] - 1][apple[1] - 1] = 'A';

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                System.out.print(grid[i][j] + " ");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        SnakeGame game = new SnakeGame(5);
        boolean gameState = true;
        game.printState();
        while(gameState){
            Scanner sc = new Scanner(System.in);
            System.out.println("Enter direction (0 = right, 1 = down, 2 = left
                , 3 = up, any other number = exit): ");
            int direction = sc.nextInt();
            if(direction < 0 || direction > 3){
                gameState=false;
                break;
            }
            game.changeDirection(direction); // Change direction to down
            if(!game.moveSnake()){
                System.out.println("Illegal move");
                gameState = false;
            }
            game.printState();
        }
    }
}
```

Listing 2: Snake Game Starter Code