Activity No. Midterm-TestSkill	
Course Code: CPE-201L	**Program:** BSCPE
Course Title: Data Structure and Algorithm	**Date Performed:** 09/06/2025
Section: 2A	**Date Submitted:** 09/06/2025
Name: Ampong, J-kevin L.	**Instructor:** Engr. Maria Rizzette H. Sayo

1.Objectives

- Distinguish between the different approaches to solve the problem and build and fundamental skill to help to grow and solve more complex problems.
- Ability to solve a particular problem while building a fundamental skill different types a data structure and allows to improve.

2. Discussion

LinkedList is one of a power data structure that can used to solve a problem what makes it different from another data is that it allows for more efficient operation like adding deleting and accessing since it used a pointer allows to reduce time need to find the next element also this allows the data to be anywhere in memory unlike another where its location must be on the sequence and fix.

3. Materials and Equipment

- **Colab** for editing, running and sharing the code it allow to run the code with installing the python itself
- **Github** for storing and sharing and collaborating to another developer

4. Procedure

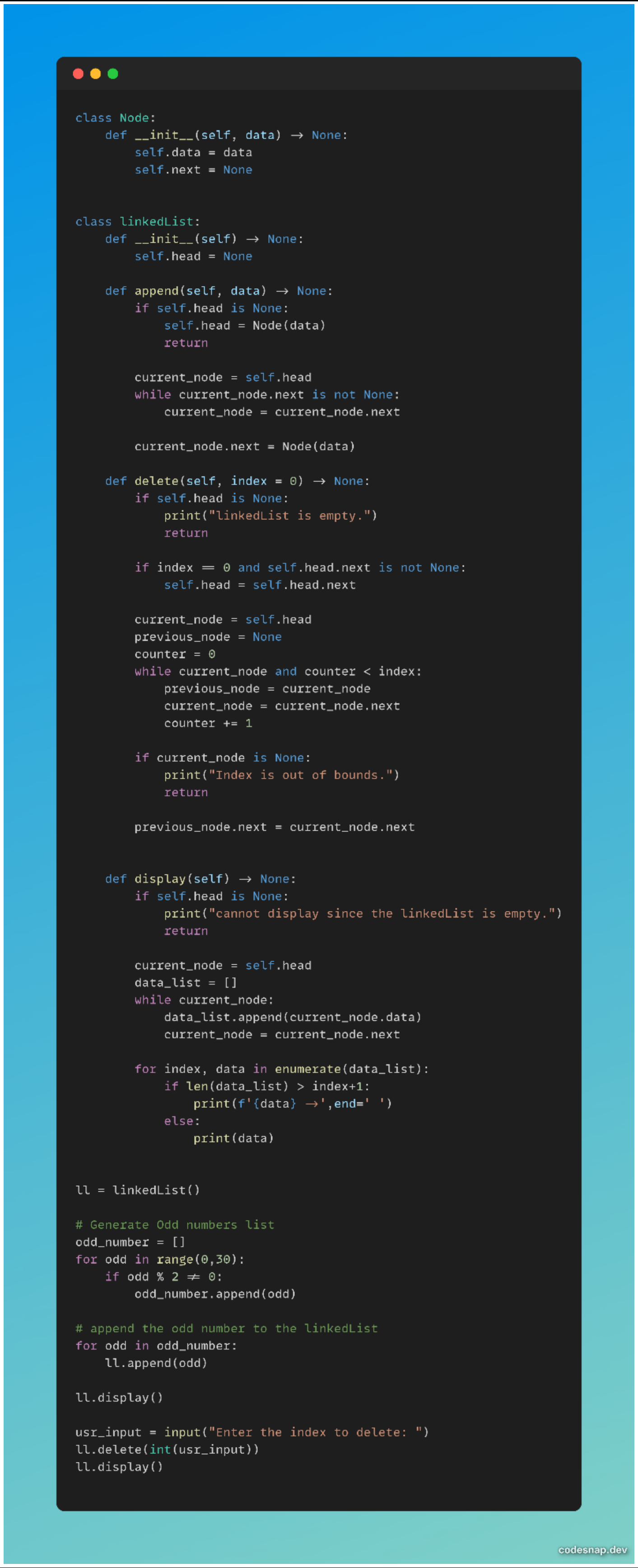
This procedure outlines the steps to create a singly linked list, populate it with data, and then delete a node at a specified index. A **singly linked list** is a linear data structure where each element, or **node**, contains a value and a pointer to the next node in the sequence. The list is accessed starting from a special node called the **head**.

Step-by-Step Procedure

1. **Initialize the Linked List**: Begin by creating an instance of the linkedList class. This creates an empty list with its head pointer set to None.
2. **Populate the Linked List**: Add elements to the list by using the append() method. In this case, a loop is used to generate odd numbers from 1 to 29 and append each one as a new node to the end of the list.
3. **Display the List (Before Deletion)**: Call the display() method to print the current contents of the linked list. This will show the original sequence of odd numbers.
4. **Prompt for User Input**: Ask the user to enter the numerical **index** of the node they want to delete. For example, an input of 0 would target the first node, and 1 would target the second.
5. **Delete the Node**:
   - The delete() method is called with the user's input as the index argument.
   - The method first checks if the list is empty. If it is, a message is printed and the process stops.
   - It then handles a special case for deleting the **first node** (index 0). If the user enters 0, the head is simply moved to the next node.
   - For any other index, the code **traverses** the list from the head until it reaches the node **before** the one to be deleted. It keeps track of both the current_node and the previous_node.
   - Once the correct position is found, the next pointer of the previous_node is updated to

point to the node **after** the current_node. This effectively "skips over" and removes the current_node from the list.
    - If the index is out of bounds (i.e., the loop finishes but current_node is None), a message is printed to the user.
6. **Display the List (After Deletion)**: Finally, the display() method is called again to show the updated linked list, confirming that the specified node has been successfully deleted.

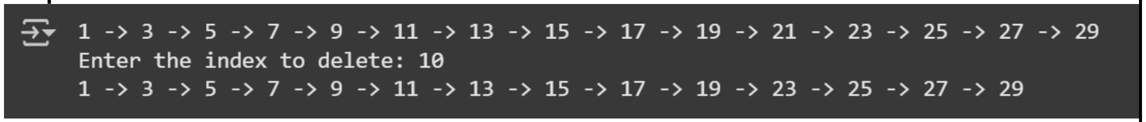
5. Output



```python
class Node:
    def __init__(self, data) -> None:
        self.data = data
        self.next = None


class linkedList:
    def __init__(self) -> None:
        self.head = None

    def append(self, data) -> None:
        if self.head is None:
            self.head = Node(data)
            return

        current_node = self.head
        while current_node.next is not None:
            current_node = current_node.next

        current_node.next = Node(data)

    def delete(self, index = 0) -> None:
        if self.head is None:
            print("linkedList is empty.")
            return

        if index == 0 and self.head.next is not None:
            self.head = self.head.next

        current_node = self.head
        previous_node = None
        counter = 0
        while current_node and counter < index:
            previous_node = current_node
            current_node = current_node.next
            counter += 1

        if current_node is None:
            print("Index is out of bounds.")
            return

        previous_node.next = current_node.next


    def display(self) -> None:
        if self.head is None:
            print("cannot display since the linkedList is empty.")
            return

        current_node = self.head
        data_list = []
        while current_node:
            data_list.append(current_node.data)
            current_node = current_node.next

        for index, data in enumerate(data_list):
            if len(data_list) > index+1:
                print(f'{data} ->',end=' ')
            else:
                print(data)


ll = linkedList()

# Generate Odd numbers list
odd_number = []
for odd in range(0,30):
    if odd % 2 != 0:
        odd_number.append(odd)

# append the odd number to the linkedList
for odd in odd_number:
    ll.append(odd)

ll.display()

usr_input = input("Enter the index to delete: ")
ll.delete(int(usr_input))
ll.display()
```

codesnap.dev

Output:



```
1 -> 3 -> 5 -> 7 -> 9 -> 11 -> 13 -> 15 -> 17 -> 19 -> 21 -> 23 -> 25 -> 27 -> 29
Enter the index to delete: 10
1 -> 3 -> 5 -> 7 -> 9 -> 11 -> 13 -> 15 -> 17 -> 19 -> 23 -> 25 -> 27 -> 29
```

6. Conclusion

Linked lists are another data structure that allows you to do the same operations as arrays, but they allow for faster operations because they store a pointer to the next node. This means that the node can be located anywhere in memory and can be dynamically expanded and shrunk, but it takes up more memory space since it is created as an object.