Linked List Week: Two-Lecture Plan

Authors: Refat Othman and Diaeddin Rimawi

Lecture 1: Theoretical Foundation of Linked Lists

Section 1: Introduction to Arrays and 2D Arrays in C/C++

Arrays are fundamental data structures that store elements in contiguous memory locations. Their simplicity makes them useful for many applications, but they also have limitations.

1. Declaration and Initialization

```
int arr[5];
arr[0] = 10;
arr[1] = 20;
```

- **Fixed Size:** Once declared, the size cannot be changed.

2. Accessing Elements

- **Time Complexity:** O(1) (direct indexing via `arr[i]`)

```
printf("%d", arr[1]);  // Output: 20
```

3. Inserting Elements

- Inserting at a position `i` requires shifting elements to the right.
- **Time Complexity:** O(n) in the worst case.

4. Deleting Elements

- Requires shifting elements to fill the gap.
- **Time Complexity:** O(n)

5. Searching for Elements

- **Linear Search:** O(n)
- **Binary Search (sorted array):** O(log n)

6. 2D Arrays (Matrices)

```c
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
printf("%d", matrix[1][2]);  // Output: 6
```

- **Access Time:** O(1)
- **Insertion/Deletion:** Generally not performed dynamically without manual memory management

Section 2: Motivation for Using Linked Lists

- **Fixed-size issue in arrays (e.g., in C/C++):**
    - Arrays are static in size; cannot grow dynamically.
    - Risk of over-allocating or under-allocating memory.
    - Leads to memory inefficiency.
- **Example in C/C++:**

```c
int arr[5];
arr[0] = 10;
arr[1] = 20;
// Cannot add more than 5 elements even if needed later.
```

- **In contrast: Linked Lists are dynamic.**
    - Memory allocation on-the-fly.
    - Efficient insertion/deletion operations.

Section 3: Introduction to Linked Lists

- **Terminology:**

  A singly linked list consists of a series of nodes where each node points to the next one in sequence. The last node points to `None`, indicating the end of the list.

  Visual Representation:

  ```
  [data1] -> [data2] -> [data3] -> ... -> [datan] -> None
  ```

    - Node: basic unit with `data` and `next` reference.
    - Head: reference to the first node.
    - Next: points to the following node or `None` if last.
- **Python Code - Node Class:**

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

- **Creating a Head Node:**

```
head = None  # empty list
```

- **LinkedList Class (initial version):**

```
class LinkedList:
    def __init__(self):
        self.head = None
```

Section 4: Building the Linked List Class

Sec 4.1: Insert at Start

- **Description:** Adds a new node at the beginning.
- **Time Complexity:** O(1) (constant time)

```
    def insert_at_start(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
```

Sec 4.2: Insert at End

- **Time Complexity:** O(n)

```
    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
```

Sec 4.3: Insert Between Two Nodes

- **Time Complexity:** O(n) (need to traverse)

In this revised approach, we insert a new node based on a given index (position) in the linked list. We start traversing from the head, keeping track of the current position until we reach the desired index or the end of the list. Once there, we perform the insertion by linking the new node to the `next` of the current node, and updating the previous node's `next` to point to the new node.

This approach assumes 0-based indexing. If the index is 0, the node is inserted at the head. If the index is beyond the current length, the node is inserted at the end.

Example implementation:

```
    def insert_at_index(self, index, data):
        new_node = Node(data)
        if index == 0:
            self.insert_at_start(data)
            return
        current = self.head
        count = 0
        while current is not None and count < index - 1:
            current = current.next
            count += 1
        if current is None:
            print("Index out of bounds; inserting at the end.")
            self.insert_at_end(data)
        else:
            new_node.next = current.next
            current.next = new_node
```

Sec 4.4: toString Method

```
    def __str__(self):
        result = ""
        temp = self.head
        while temp:
            result += f"[{temp.data}]->"
            temp = temp.next
        result += "None"
        return result
```

Sec 4.5: Compute Length

- **Time Complexity:** O(n)

```
    def length(self):
        count = 0
```

```
        temp = self.head
        while temp:
            count += 1
            temp = temp.next
        return count
```

Sec 4.6: Delete from Start

- **Time Complexity:** O(1)

```
    def delete_at_start(self):
        if self.head:
            self.head = self.head.next
```

Sec 4.6: Delete from End

- **Time Complexity:** O(n)

```
    def delete_at_end(self):
        if self.head is None:
            return
        if self.head.next is None:
            self.head = None
            return
        temp = self.head
        while temp.next.next:
            temp = temp.next
        temp.next = None
```

Sec 4.7: Delete Between Two Nodes

- **Time Complexity:** O(n)

```
    def delete_node(self, key):
        temp = self.head
        if temp and temp.data == key:
            self.head = temp.next
            return
        prev = None
        while temp and temp.data != key:
            prev = temp
            temp = temp.next
        if temp is None:
            return
        prev.next = temp.next
```

Sec 4.8: Search for an Item

- **Time Complexity:** O(n)

```
    def search(self, key):
        temp = self.head
        while temp:
            if temp.data == key:
                return True
            temp = temp.next
        return False
```

Lecture 2: Exercises with Linked List

Exercise 1: Find the Middle of a Linked List

- Use slow and fast pointer approach.
- Display the middle node's value.

Exercise 2: Merge Two Sorted Linked Lists

- Given two linked lists, write a function to merge them into a new sorted linked list.

Exercise 3: Remove Duplicates

- Given a linked list with duplicates, remove all duplicate entries.
- Assume input list is sorted.

Assignment

Reversing a Linked List

- Implement a method to reverse a linked list.
- Print the linked list before and after.