Linked List Week 2: Advanced Variations and Applications

Authors: Refat Othman and Diaeddin Rimawi

Lecture 1: Linked List Variations

1. Tail References (Head and Tail Pointers)

- **Description:** A singly linked list typically maintains a reference only to the head node. By adding a tail pointer, we keep a direct reference to the last node in the list. This enhances the efficiency of operations that append to the end.
- **Advantages:**
  - Constant time (O(1)) insertion at the end.
  - Efficient queue implementation (we'll discuss later).
- **Basic Implementation (Python):**

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

    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail = new_node
```

- **Other operations:**
  - Update tail when deleting last node:

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

2. Circular Linked List

- **Description:** A circular linked list is a list where the last node points back to the first node (head), forming a circle.
- **Advantages:**
  - Can start traversal from any node.
  - Useful for several basic scenarios where continuous looping is needed. For example:
    - You can start traversing from any node and reach all others.
    - It avoids null checks at the end, simplifying the traversal logic.
    - It's useful in simple simulations or scenarios like managing a playlist that repeats.
- **Traversal Example:**

```python
def traverse_circular_list(head):
    if head is None:
        return
    temp = head
    while True:
        print(temp.data, end=' -> ')
        temp = temp.next
        if temp is head:
            break
```

- **Detect Circular List:**

```python
def is_circular(head):
    if head is None:
        return False
    temp = head.next
    while temp is not None and temp is not head:
        temp = temp.next
    return temp is head
```

3. Doubly Linked List

- **Description:** In a doubly linked list, each node contains references to both the next and the previous node.
- **Advantages:**
  - Bidirectional traversal.

    - Efficient insertion and deletion from both ends.

- **Node and List Implementation (Python):**

```python
class DNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def insert_at_start(self, data):
        new_node = DNode(data)
        if self.head:
            self.head.prev = new_node
            new_node.next = self.head
        self.head = new_node

    def insert_at_end(self, data):
        new_node = DNode(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node
        new_node.prev = temp

    def insert_at_index(self, index, data):
        if index == 0:
            self.insert_at_start(data)
            return
        temp = self.head
        count = 0
        while temp and count < index:
            temp = temp.next
            count += 1
        if not temp:
            self.insert_at_end(data)
            return
        new_node = DNode(data)
        prev_node = temp.prev
        new_node.prev = prev_node
        new_node.next = temp
        prev_node.next = new_node
        temp.prev = new_node
```

- **Traversal Forward and Backward:**

```python
def traverse_forward(head):
    while head:
        print(head.data, end=' <-> ')
        if head.next is None:
            tail = head  # store last for reverse
        head = head.next
    print("NULL")

def traverse_backward(tail):
    while tail:
        print(tail.data, end=' <-> ')
        tail = tail.prev
    print("NULL")
```

3.1 Circular Doubly Linked List

- **Description:** A circular doubly linked list is a combination of both circular and doubly linked lists. Each node has a reference to both the next and previous nodes, and the last node links back to the head, while the head's previous pointer refers to the last node.
- **Advantages:**
  - Eliminates null references.
  - Insertion and deletion become more uniform as there is no need to check for null pointers.
- **Basic Structure:**

```python
class CDNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class CircularDoublyLinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = CDNode(data)
        if not self.head:
            self.head = new_node
            new_node.next = new_node
            new_node.prev = new_node
        else:
            tail = self.head.prev
            tail.next = new_node
            new_node.prev = tail
            new_node.next = self.head
            self.head.prev = new_node
```

- **Traversal Forward and Backward:**

```
def traverse_forward(head):
    if not head:
        return
    temp = head
    while True:
        print(temp.data, end=' <-> ')
        temp = temp.next
        if temp is head:
            break
    print('(head)')

def traverse_backward(head):
    if not head:
        return
    temp = head.prev
    while True:
        print(temp.data, end=' <-> ')
        temp = temp.prev
        if temp.next is head:
            break
    print('(head)')
```

4. Insertion Sort using Singly Linked List

- **Description:** Insertion sort can be implemented using a singly linked list by building a new sorted list node by node. For each node in the input list, we find the appropriate position in the sorted portion and insert it.
- **Advantages in Linked List over Array:**
  - No need for shifting elements; only pointers are rearranged.
  - Better suited for scenarios with frequent insertions.
- **Time Complexity:**
  - Worst-case: O(n^2)
  - Best-case: O(n) (already sorted)
- **Python code for Singly Linked List Insertion Sort:**

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

```python
def insertion_sort_sll(head):
    sorted_head = None
    current = head

    while current:
        next_node = current.next
        if sorted_head is None or current.data < sorted_head.data:
            current.next = sorted_head
            sorted_head = current
        else:
            search = sorted_head
            while search.next and search.next.data < current.data:
                search = search.next
            current.next = search.next
            search.next = current
        current = next_node

    return sorted_head
```

Lecture 2: Exercises and Applications

Exercise 1: Josephus Elimination Game

- **Problem Description:** You are given a group of `n` people, each assigned a unique identifier from 1 to `n`, standing in a sequence. Starting from the first person, a counting process is carried out repeatedly: every time, the `k`-th person in the sequence is removed. After each removal, counting resumes from the next person in line. This elimination process continues until only one person remains. Your task is to determine the identifier of that last remaining person.
- **Example:** Suppose `n = 5` and `k = 3`. The initial configuration is:

  The elimination proceeds as follows:

  - Start at 1, count 1 → 2 → eliminate 3
  - Then 4 → 5 → eliminate 1
  - Then 2 → 4 → eliminate 5
  - Then 2 → 4 → eliminate 2
  - Remaining: [4]

Exercise 2: List Reduction to Non-Decreasing Order

- **Problem Description:** You are given a singly linked list containing a sequence of integers. You can perform the following operation any number of times:

  1. Identify the adjacent pair of nodes with the minimum sum. If multiple such pairs exist, choose the leftmost one.
  2. Replace the two nodes with a single new node containing the sum of their values, preserving the structure of the singly linked list.

Your objective is to determine the minimum number of operations required to transform the list into a non-decreasing sequence—where each node's value is greater than or equal to the one before it.

- **Example 1:**
  - Input: [5] -> [2] -> [3] -> [1]
  - Output: 2
  - Explanation:
    - The pair (3,1) has the minimum sum of 4. After replacement: [5] -> [2] -> [4]
    - The pair (2,4) has the minimum sum of 6. After replacement: [5] -> [6]
    - Now the list is non-decreasing.

Exercise 4: Reverse a Doubly Linked List

- You are given a doubly linked list of integers. Your task is to write a function that reverses the entire list in-place.
- After reversing, the head should point to the last element, and traversal should continue correctly using the updated `prev` and `next` pointers.
- Example: Input: [1] <-> [2] <-> [3] <-> [4] After reversal: [4] <-> [3] <-> [2] <-> [1]

Assignment

Implement a class `SortedCircularLinkedList` that maintains elements in sorted order on each insertion.

Example: Insert the following integers into the list in this order: 7, 3, 9, 1, 4.

The list should organize itself automatically to remain sorted after each insertion:

- After inserting 7: [7]
- After inserting 3: [3] -> [7]
- After inserting 9: [3] -> [7] -> [9]
- After inserting 1: [1] -> [3] -> [7] -> [9]
- After inserting 4: [1] -> [3] -> [4] -> [7] -> [9]

Each node's `next` pointer should continue in sorted order, and the last node should point back to the head to preserve the circular structure.

Your implementation should support:

- An `insert` method
- A method to print the list contents starting from the head and ending when the traversal completes a full cycle

---