Array and Linked List

Memory Storage of Data

Memory and Data Storage

- **Memory (RAM)**: It's a volatile storage area where data and instructions are stored for quick access by the CPU. Each memory location has a unique address.
- **Data Storage**: Data in memory is typically stored in bytes. Data structures like arrays and linked lists utilize this memory differently to manage and organize data.

Arrays

- **Storage in Memory**: Arrays store elements in contiguous memory locations. Each element can be accessed directly using its index, which makes it efficient for random access.
- **Pros**:
  - **Fast Access**: Direct access to elements using indices.
  - **Memory Efficiency**: Requires less memory overhead compared to linked lists.
- **Cons**:
  - **Fixed Size**: Arrays have a fixed size, which must be defined at the time of allocation.
  - **Insertion/Deletion Overhead**: Inserting or deleting elements can be costly, as it may require shifting elements.

Example from "Grokking Algorithms"

Consider an array of integers `[3, 5, 7, 9, 11]`. If you want to access the third element, you can do so in constant time using its index (e.g., `array[2]`).

Linked Lists

- **Storage in Memory**: Linked lists store elements in nodes that are not necessarily contiguous in memory. Each node contains data and a

reference (or pointer) to the next node.

- **Pros**:
  - **Dynamic Size**: Linked lists can grow or shrink in size dynamically.
  - **Efficient Insertions/Deletions**: Inserting or deleting elements can be done efficiently, especially if the position is known.
- **Cons**:
  - **Slower Access**: Accessing elements requires traversing the list from the head, leading to linear time complexity.
  - **Memory Overhead**: Each node requires additional memory for storing pointers.

Example from "Grokking Algorithms"

Consider a linked list with nodes containing integers `[3 -> 5 -> 7 -> 9 -> 11]`. To access the third element, you must traverse the list from the head through each node until you reach the desired element.

Operations on Arrays and Linked Lists

Arrays

1. **Accessing**: $O(1)$ - Direct access using indices.
2. **Insertion**: $O(n)$ - Worst case when inserting at the beginning or middle.
3. **Deletion**: $O(n)$ - Worst case when deleting from the beginning or middle.
4. **Update**: $O(1)$ - Direct access and update using indices.

Linked Lists

1. **Accessing**: $O(n)$ - Requires traversal.
2. **Insertion**: $O(1)$ - If inserting at the head or after a known node.
3. **Deletion**: $O(1)$ - If deleting the head or a known node.
4. **Update**: $O(n)$ - Requires traversal to the node.

Exercises

Exercise 1: Implementing Array Operations

1. **Accessing an Element**: Write a function to access an element in an array by index.
2. **Inserting an Element**: Write a function to insert an element at a specified position in an array.
3. **Deleting an Element**: Write a function to delete an element from a specified position in an array.

Exercise 2: Implementing Linked List Operations

1. **Accessing an Element**: Write a function to access an element in a linked list by position.
2. **Inserting an Element**: Write a function to insert a new node at the end of a linked list.
3. **Deleting an Element**: Write a function to delete a node from a specified position in a linked list.

Example Code

C# - Linked List Operations

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

public class LinkedList
{
    public Node head;

    // Inserting at the end
    public void Insert(int data)
    {
```

```
        Node newNode = new Node(data);
        if (head == null)
        {
            head = newNode;
            return;
        }
        Node last = head;
        while (last.next != null)
        {
            last = last.next;
        }
        last.next = newNode;
    }

    // Deleting by position
    public void Delete(int position)
    {
        if (head == null) return;

        Node temp = head;

        if (position == 0)
        {
            head = temp.next;
            return;
        }

        for (int i = 0; temp != null && i < position - 1; i
++)
            temp = temp.next;

        if (temp == null || temp.next == null)
            return;

        Node next = temp.next.next;

        temp.next = next;
    }
```

```
    // Accessing by position
    public int Access(int position)
    {
        Node current = head;
        int count = 0;
        while (current != null)
        {
            if (count == position)
                return current.data;
            count++;
            current = current.next;
        }
        return -1;
    }
}
```

Python - Array Operations

```
def access_element(arr, index):
    return arr[index]

def insert_element(arr, element, position):
    return arr[:position] + [element] + arr[position:]

def delete_element(arr, position):
    return arr[:position] + arr[position+1:]

# Example usage
arr = [3, 5, 7, 9, 11]
print(access_element(arr, 2))  # Output: 7
print(insert_element(arr, 6, 2))  # Output: [3, 5, 6, 7, 9,
11]
print(delete_element(arr, 2))  # Output: [3, 5, 9, 11]
```

Conclusion

Both arrays and linked lists have their advantages and disadvantages based on how they store data in memory. Arrays provide fast access but are fixed in size and inefficient for insertions and deletions. Linked lists are dynamic and efficient for insertions and deletions but provide slower access due to the need for traversal. Understanding these differences helps in choosing the appropriate data structure for specific applications.