Santiago de cali

Taller 4

Ingeniería en sistemas

Uniremington

Sebastián Casañas Salcedo

Cc:1005707698.

Iván González

Cc:1234193301.

Danna Michelle Caicedo

Cc:1086634456.

Angela María Medina Ruiz

Cc:1088034273

Profesor Jorge Gamarra

Análisis

1. Identifica cuáles operaciones son O(1) y cuáles son O(n).

R// O(1)

```
// Inserta un nodo al inicio de la lista (O(1))
insertFirst(value) {
  const newNode = new Node(value); // Crear nuevo nodo
  newNode.next = this.head;        // Apuntar al antiguo primer nodo
  this.head = newNode;             // Actualizar el head al nuevo nodo

  if (!this.tail) {                // Si la lista estaba vacía
    this.tail = newNode;           // El nuevo nodo también es el tail
  }

  this.size++;                     // Aumentar el tamaño de la lista
}
```

La operación insertFirst tiene complejidad O(1) porque solo realiza pasos fijos (crear nodo, ajustar punteros y actualizar tamaño) sin importar el tamaño de la lista. No necesita recorrer la lista, por eso su tiempo es constante.

```
// Inserta un nodo al final de la lista (O(1) si hay tail, O(n) si no)
insertLast(value) {
  const newNode = new Node(value); // Crear nuevo nodo

  if (!this.head) {                // Si la lista está vacía
    this.head = newNode;           // Nuevo nodo es el head
    this.tail = newNode;           // Y también el tail
  } else {
    this.tail.next = newNode;      // Conectar el último nodo al nuevo
    this.tail = newNode;           // Actualizar el tail
  }

  this.size++;                     // Aumentar el tamaño
}
```

La operación insertLast tiene complejidad O(1) porque agrega el nuevo nodo directamente al final de la lista usando el puntero tail, sin necesidad de recorrer la lista.

```
// Elimina el primer nodo (O(1))
removeFirst() {
  if (!this.head) return null;      // Lista vacía

  const removed = this.head;        // Guardar nodo eliminado
  this.head = this.head.next;       // Actualizar head al siguiente
  this.size--;                      // Reducir tamaño

  if (this.size === 0) {            // Si la lista queda vacía
    this.tail = null;               // El tail también se elimina
  }

  return removed.value;             // Retornar valor eliminado
}
```

La función removeFirst tiene complejidad O(1) porque elimina el primer elemento en un número fijo de pasos: actualiza el puntero head al siguiente nodo, ajusta tail si la lista queda vacía y disminuye el tamaño. Estas operaciones no dependen de la longitud de la lista, por lo que el tiempo es constante.

O(N)

```
insertAt(index, value) {
  if (index < 0 || index > this.size) return null; // Índice inválido

  if (index === 0) return this.insertFirst(value); // Insertar al inicio
  if (index === this.size) return this.insertLast(value); // Insertar al final

  const newNode = new Node(value); // Crear nuevo nodo
  let current = this.head;         // Nodo actual
  let previous = null;             // Nodo anterior
  let i = 0;

  while (i < index) {              // Recorrer hasta la posición deseada
    previous = current;
    current = current.next;
    i++;
  }

  previous.next = newNode;         // Conectar anterior con nuevo
  newNode.next = current;          // Conectar nuevo con siguiente
  this.size++;                     // Aumentar tamaño
}
```

La operación insertAt tiene complejidad O(n) debido a que, en el peor caso, debe recorrer la lista desde el inicio hasta la posición indicada (índice) para insertar el nuevo nodo. Este recorrido hace que el tiempo de ejecución crezca linealmente con el tamaño de la lista

```
// Elimina el último nodo (O(n))
removeLast() {
  if (!this.head) return null;      // Lista vacía

  if (this.size === 1) {            // Solo hay un nodo
    const removed = this.head;
    this.head = null;
    this.tail = null;
    this.size = 0;
    return removed.value;
  }

  let current = this.head;
  while (current.next !== this.tail) { // Recorrer hasta el penúltimo
    current = current.next;
  }

  const removed = this.tail;        // Guardar nodo eliminado
  current.next = null;              // Desconectar el último nodo
  this.tail = current;              // Actualizar tail
  this.size--;                      // Reducir tamaño

  return removed.value;
}
```

La operación removeLast tiene complejidad O(n) porque para eliminar el último nodo debe recorrer la lista desde el inicio hasta el penúltimo nodo. Este recorrido hace que el tiempo de ejecución crezca linealmente con el tamaño de la lista.

```
// Elimina un nodo en una posición específica (O(n))
removeAt(index) {
  if (index < 0 || index >= this.size) return null; // Índice inválido

  if (index === 0) return this.removeFirst();       // Eliminar al inicio
  if (index === this.size - 1) return this.removeLast(); // Eliminar al final

  let current = this.head;
  let previous = null;
  let i = 0;

  while (i < index) {              // Recorrer hasta el índice
    previous = current;
    current = current.next;
    i++;
  }

  previous.next = current.next;    // Saltar el nodo eliminado
  this.size--;                     // Reducir tamaño

  return current.value;            // Retornar valor eliminado
}
```

La función removeAt tiene una complejidad O(n) porque necesita desplazarse a través de la lista hasta llegar al nodo en la posición indicada para poder eliminarlo.

3. **Compara los tiempos medidos y confirma si los resultados coinciden con la teoría**
   Los resultados medidos muestran un incremento proporcional al tamaño de la lista, lo cual contradice la teoría y lo implementado en el codigo, donde ambas inserciones deberían ser operaciones de tiempo constante. Por lo tanto, los tiempos medidos NO coinciden con la teoría.

Preguntas de reflexión

1. ¿Qué ventajas y desventajas tiene una lista simplemente enlazada frente a un arreglo tradicional en términos de rendimiento y uso de memoria?

R// Las listas enlazadas ofrecen inserciones y eliminaciones rápidas y tamaño dinámico, pero tienen acceso lento y usan más memoria por los punteros. Los arreglos permiten acceso rápido y mejor uso de memoria, pero su tamaño es fijo y modificarlos es costoso en cuanto a sacrificar rendimiento.

2. **Si se quisiera implementar la eliminación de un nodo específico por valor y no por índice, ¿cómo afectaría eso la complejidad temporal?**

   R// Eliminar un nodo por valor en una lista simplemente enlazada requiere buscarlo primero, lo que toma tiempo lineal O(n). Luego, la eliminación es O(1), pero en total la operación sigue siendo O(n) debido a la búsqueda.

3. **¿Cómo cambiaría la complejidad temporal si en lugar de agregar al final de la lista se insertara un nodo en una posición aleatoria en cada operación?**
   R// Si en lugar de agregar un nodo al final de la lista se inserta en una posición aleatoria, la complejidad temporal cambia a O(n). Esto ocurre porque hay que recorrer la lista hasta esa posición para hacer la inserción, y recorrer la lista crece linealmente con el tamaño de la misma. En cambio, agregar al final usando un puntero tail es O(1) porque no requiere recorrido.