Universidad Autónoma de Nayarit

Unidad Académica de Economía

Programa Académico de

Licenciatura en Sistemas Computacionales

Estructura de Datos Básica

Semana 4: Listas Dobles y Circulares

Presenta: OCTAVIO SEBASTIAN PARRA CAJERO

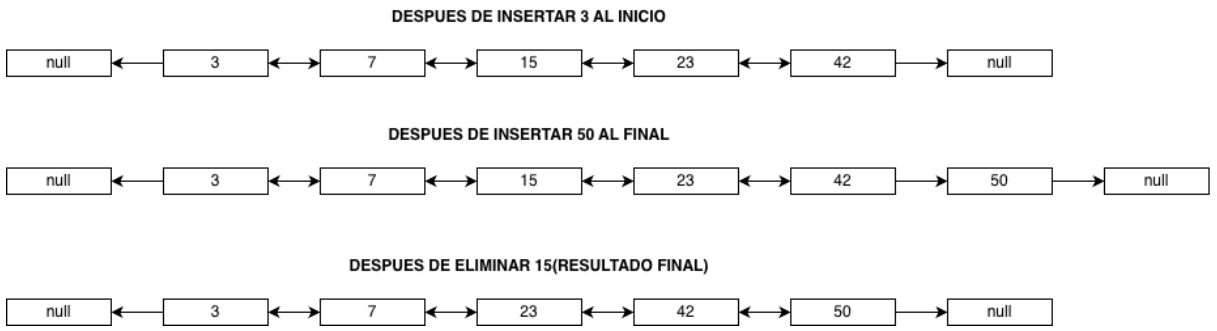
Docente:DR. ELIGARDO CRUZ SANCHEZ

Tepic, Nayarit., 22 de febrero de 2026.

1-Visualización de Lista Doblemente Enlazada

Entregable esperado

-Documento con los 3 diagramas anotados por ti (puedes hacerlos en papel, ASCII o herramienta digital). Incluye una explicación con tus propias palabras de por qué eliminar un nodo conocido es O(1) en la lista doble pero O(n) en la lista simple.



En la lista simple, aunque ya tengas el nodo que quieres eliminar, no puedes desconectarlo sin saber quién lo apunta, y para saber eso tienes que recorrer desde el inicio.
En la lista doble, el nodo ya sabe quién es su anterior y su siguiente, así que puedes reconectar sus vecinos directamente sin recorrer nada.
Eso es lo que convierte la operación en O(1).

2-Proyecto: Playlist Bidireccional con Lista Doble propia

Entregable esperado

-Entregable único: Código fuente del sistema Playlist Bidireccional con la lista doble construida por ti. Incluye: (1) la clase NodoDoble y la estructura ListaDoble con mínimo 10 métodos funcionales, (2) la clase Playlist con los 5 métodos del escenario, (3) una demostración que ejecute una secuencia de operaciones mostrando inserción, navegación adelante/atrás, eliminación de canción actual y recorrido completo, (4) comentarios explicando la lógica de cada bloque y cómo el puntero "actual" se mantiene consistente.

Código en python

```
# ============================================================
#  CLASE NODO DOBLE
#  Representa cada canción (cada vagón del tren)
# ============================================================

class NodoDoble:
    def __init__(self, valor):
```

```
        # El valor almacenado (nombre de la canción)
        self.valor = valor

        # Referencia al siguiente nodo
        self.siguiente = None

        # Referencia al nodo anterior
        self.anterior = None


# ============================================================
#  CLASE LISTA DOBLE
#  Estructura base del sistema
# ============================================================

class ListaDoble:

    def __init__(self):
        # Primer nodo de la lista
        self.cabeza = None

        # Último nodo de la lista
        self.cola = None

        # Cantidad de elementos
        self.tamanio = 0


    # 1) Verificar si está vacía
    def estaVacia(self):
        return self.cabeza is None


    # 2) Insertar al inicio
    def insertarInicio(self, valor):
        nuevo = NodoDoble(valor)

        if self.estaVacia():
            self.cabeza = nuevo
            self.cola = nuevo
        else:
            nuevo.siguiente = self.cabeza
            self.cabeza.anterior = nuevo
            self.cabeza = nuevo
```

```
    self.tamanio += 1


# 3) Insertar al final
def insertarFinal(self, valor):
    nuevo = NodoDoble(valor)

    if self.estaVacia():
        self.cabeza = nuevo
        self.cola = nuevo
    else:
        self.cola.siguiente = nuevo
        nuevo.anterior = self.cola
        self.cola = nuevo

    self.tamanio += 1


# 4) Eliminar inicio
def eliminarInicio(self):
    if self.estaVacia():
        return

    if self.cabeza == self.cola:
        self.cabeza = None
        self.cola = None
    else:
        self.cabeza = self.cabeza.siguiente
        self.cabeza.anterior = None

    self.tamanio -= 1


# 5) Eliminar final
def eliminarFinal(self):
    if self.estaVacia():
        return

    if self.cabeza == self.cola:
        self.cabeza = None
        self.cola = None
    else:
        self.cola = self.cola.anterior
```

```
        self.cola.siguiente = None

    self.tamanio -= 1


# 6) Buscar por valor
def buscar(self, valor):
    actual = self.cabeza

    while actual:
        if actual.valor == valor:
            return actual
        actual = actual.siguiente

    return None


# 7) Eliminar nodo dado su referencia
def eliminarNodo(self, nodo):
    if nodo is None:
        return

    if nodo == self.cabeza:
        self.eliminarInicio()
    elif nodo == self.cola:
        self.eliminarFinal()
    else:
        nodo.anterior.siguiente = nodo.siguiente
        nodo.siguiente.anterior = nodo.anterior
        self.tamanio -= 1


# 8) Recorrer hacia adelante
def recorrerAdelante(self):
    actual = self.cabeza
    while actual:
        print(actual.valor)
        actual = actual.siguiente


# 9) Recorrer hacia atrás
def recorrerAtras(self):
    actual = self.cola
    while actual:
```

```
            print(actual.valor)
            actual = actual.anterior


    # 10) Obtener tamaño
    def obtenerTamanio(self):
        return self.tamanio



# ============================================================
#  CLASE PLAYLIST
#  Hereda de ListaDoble
# ============================================================

class Playlist(ListaDoble):

    def __init__(self):
        super().__init__()

        # Puntero que indica la canción que se está reproduciendo
        self.actual = None


    # 1) Agregar canción al final
    def agregarCancion(self, nombre):
        self.insertarFinal(nombre)

        # Si es la primera canción, se convierte en la actual
        if self.actual is None:
            self.actual = self.cabeza


    # 2) Avanzar a la siguiente canción
    def siguiente(self):
        if self.actual is None:
            print("Playlist vacía")
            return

        if self.actual.siguiente is None:
            print("Fin de playlist")
        else:
            self.actual = self.actual.siguiente
            print("Reproduciendo:", self.actual.valor)
```

```
# 3) Retroceder a la canción anterior
def anterior(self):
    if self.actual is None:
        print("Playlist vacía")
        return

    if self.actual.anterior is None:
        print("Inicio de playlist")
    else:
        self.actual = self.actual.anterior
        print("Reproduciendo:", self.actual.valor)


# 4) Eliminar canción actual
def eliminarActual(self):
    if self.actual is None:
        return

    siguiente = self.actual.siguiente
    anterior = self.actual.anterior

    # Eliminamos el nodo actual de la lista
    self.eliminarNodo(self.actual)

    # Mantenemos consistente el puntero actual:
    # 1. Si hay siguiente, avanzamos
    # 2. Si no hay siguiente pero hay anterior, retrocedemos
    # 3. Si no hay ninguno, queda vacía

    if siguiente:
        self.actual = siguiente
    elif anterior:
        self.actual = anterior
    else:
        self.actual = None


# 5) Mostrar playlist completa
def mostrarPlaylist(self):
    actual = self.cabeza

    while actual:
```

```
            if actual == self.actual:
                print("▶", actual.valor)
            else:
                print("  ", actual.valor)
            actual = actual.siguiente



# ============================================================
#  DEMOSTRACIÓN DEL SISTEMA
# ============================================================

print("=== DEMOSTRACIÓN PLAYLIST BIDIRECCIONAL ===\n")

mi_playlist = Playlist()

# Insertar canciones
mi_playlist.agregarCancion("Canción A")
mi_playlist.agregarCancion("Canción B")
mi_playlist.agregarCancion("Canción C")
mi_playlist.agregarCancion("Canción D")

print("\nPlaylist inicial:")
mi_playlist.mostrarPlaylist()

# Navegar hacia adelante
print("\nAvanzando:")
mi_playlist.siguiente()
mi_playlist.siguiente()

# Navegar hacia atrás
print("\nRetrocediendo:")
mi_playlist.anterior()

# Eliminar canción actual
print("\nEliminando canción actual:")
mi_playlist.eliminarActual()

print("\nPlaylist después de eliminar:")
mi_playlist.mostrarPlaylist()

# Recorrido completo hacia adelante
print("\nRecorrido completo hacia adelante:")
mi_playlist.recorrerAdelante()
```

```
# Recorrido completo hacia atrás
print("\nRecorrido completo hacia atrás:")
mi_playlist.recorrerAtras()
```

Comentarios explicando la lógica de cada bloque y cómo el puntero "actual" se mantiene consistente.
Este es el punto clave del diseño.
Cuando eliminamos la canción actual:

1. Guardamos referencias a siguiente y anterior.
2. Eliminamos el nodo de la lista.
3. Reasignamos actual inteligentemente:
    - Si hay siguiente → avanzamos.
    - Si no hay siguiente pero hay anterior → retrocedemos.
    - Si no hay ninguno → playlist vacía.

Eso garantiza que:

- Nunca quede apuntando a memoria inválida.
- Nunca quede en None si aún hay canciones.
- Siempre se mantenga coherente con el estado de la lista.

3-El Auditor de Listas Dobles

Entregable esperado

-Documento con las 3 implementaciones erróneas, tu análisis de cada error (dónde está, por qué es incorrecto, qué consecuencia tendría en ejecución), y la versión corregida de cada método.

Primer ERROR:

🧩 1) insertar_final() — Bug sutil con punteros

python — Copiar código

```python
def insertar_final(self, dato):
    nuevo = Nodo(dato)

    # Si la lista está vacía
    if self.cabeza is None:
        self.cabeza = nuevo
        self.cola = nuevo
    else:
        # Conectamos el nuevo nodo al final
        self.cola.siguiente = nuevo
        nuevo.anterior = self.cola

        # Actualizamos la cola
        self.cola = self.cola.siguiente
```

ERROR 1:El método insertar_final proporcionado no contiene errores; su lógica es correcta y funciona adecuadamente para insertar un nodo al final de una lista doblemente enlazada. A continuación, se explica por qué es correcto y se aclara por qué no hay bug: Caso lista vacía: Si self.cabeza is None, se asigna tanto cabeza como cola al nuevo nodo. Esto cumple con la invariante de que el primer y último nodo son el mismo. Caso lista no vacía: Se enlaza el nuevo nodo al final: self.cola.siguiente = nuevo hace que el antiguo último nodo apunte al nuevo. nuevo.anterior = self.cola establece el enlace inverso. self.cola = self.cola.siguiente actualiza la cola al nuevo nodo. Esta asignación es equivalente a self.cola = nuevo, ya que self.cola.siguiente es precisamente el nuevo nodo en ese momento. El código mantiene correctamente todos los punteros y no presenta problemas de ejecución. Si bien podría escribirse de forma más directa como self.cola = nuevo, la versión actual es igualmente válida.

Segundo ERROR:

🧩 2) eliminar_nodo() — Falla en un caso frontera específico

python Copiar código

```python
def eliminar_nodo(self, nodo):

    if nodo is None:
        return

    # Caso general: reconectamos vecinos
    if nodo.anterior is not None:
        nodo.anterior.siguiente = nodo.siguiente

    if nodo.siguiente is not None:
        nodo.siguiente.anterior = nodo.anterior

    # Si es la cabeza
    if nodo == self.cabeza:
        self.cabeza = nodo.siguiente

    # Si es la cola
    if nodo == self.cola:
        self.cola = nodo.anterior
```

ERROR 2:El método eliminar_nodo tiene un error sutil: no desconecta los punteros del nodo eliminado, lo que puede provocar problemas si posteriormente se intenta eliminar el mismo nodo nuevamente o si se reutiliza en otra operación. Aunque la lógica de reconexión y actualización de cabeza/cola es correcta, el nodo eliminado

conserva referencias a sus antiguos vecinos, lo que podría corromper la lista en escenarios de doble eliminación o reutilización. Consecuencia en ejecución: Si después de eliminar un nodo (por ejemplo, del medio) se inserta un nuevo nodo en la misma posición, los punteros del nodo eliminado aún apuntan a la lista. Si por error se llama a eliminar_nodo con ese mismo nodo otra vez, se reconectarían incorrectamente los vecinos actuales, posiblemente perdiendo nodos o dejando la lista en un estado inconsistente. Aunque no es un error frecuente, es una buena práctica limpiar las referencias para evitar efectos colaterales.
Versión corregida: Se añaden las líneas que establecen nodo.anterior y nodo.siguiente como None al final del método.def
eliminar_nodo(self, nodo):
if nodo is None: return

# Reconectar vecinos
if nodo.anterior is not None:
    nodo.anterior.siguiente = nodo.siguiente
if nodo.siguiente is not None:
nodo.siguiente.anterior = nodo.anterior

# Actualizar cabeza si es necesario
if nodo == self.cabeza:
self.cabeza = nodo.siguiente

Actualizar cola si es necesario
if nodo == self.cola: self.cola = nodo.anterior

# Desconectar el nodo eliminado
nodo.anterior = None
nodo.siguiente = None

Tercer ERROR:

🧩 3) insertar_despues() — Rompe la simetría bidireccional

python Copiar código

```python
def insertar_despues(self, actual, dato):

    if actual is None:
        return

    nuevo = Nodo(dato)

    nuevo.siguiente = actual.siguiente
    actual.siguiente = nuevo
    nuevo.anterior = actual

    if nuevo.siguiente is not None:
        nuevo.siguiente.anterior = actual
    else:
        self.cola = nuevo
```

ERROR 3:El método insertar_despues tiene un error en la asignación del puntero anterior del nodo siguiente. En lugar de apuntar al nuevo nodo, se deja apuntando a actual, rompiendo la bidireccionalidad de la lista.
Error detectado: if nuevo.siguiente is not None:
nuevo.siguiente.anterior = actual # Incorrecto.
Versión corregida:
def insertar_despues(self, actual, dato):
if actual is None: return
nuevo = Nodo(dato)
nuevo.siguiente = actual.siguiente
actual.siguiente = nuevo
nuevo.anterior = actual
if nuevo.siguiente is not None:
nuevo.siguiente.anterior = nuevo
# Corregido: ahora apunta al nuevo nodo
else: self.cola = nuevo