LABORATORIO LINKED LISTS

Algoritmos y Estructuras de datos (AYED) 2024-1

FECHA
07-04-2024

ANDERSSON DAVID SÁNCHEZ MÉNDEZ
CRISTIAN SANTIAGO PEDRAZA RODRÍGUEZ

DAVID EDUARDO SALAMANCA AGUILAR

LABORATORIO LINKED LISTS

CONCEPTOS

- **Linked Lists:** estructuras de datos lineales en el que los elementos no están almacenados en bloques continuos de memoria, sino más bien, se almacenan en diferentes sectores de memoria. Se puede acceder a estos mediante punteros.

  Algunas ventajas de usar Linked List:

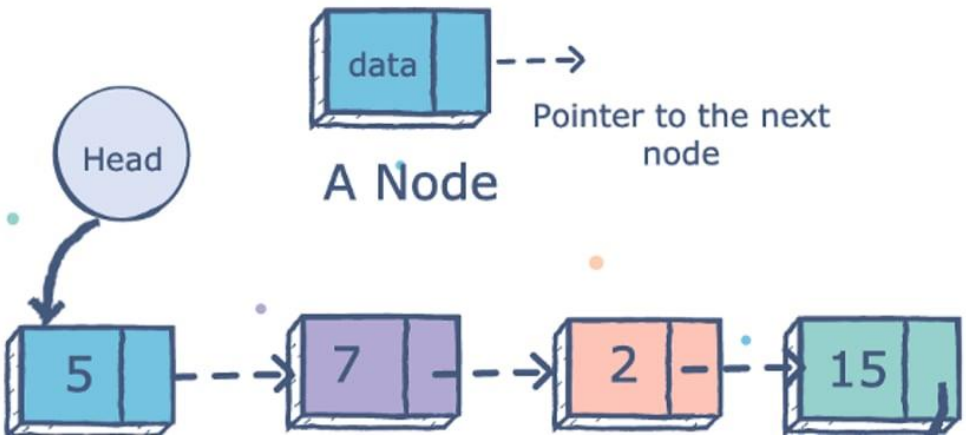
  - No tiene tamaño fijo, es decir el tamaño es dinámico.
  - Procesos fáciles para agregar o mover elementos.

  Por el otro lado, algunas desventajas son:

  - El acceso aleatorio (random) no está permitido, para solucionar este problema, se necesita iniciar desde el nodo para hacer un bucle con los elementos que se requieran.

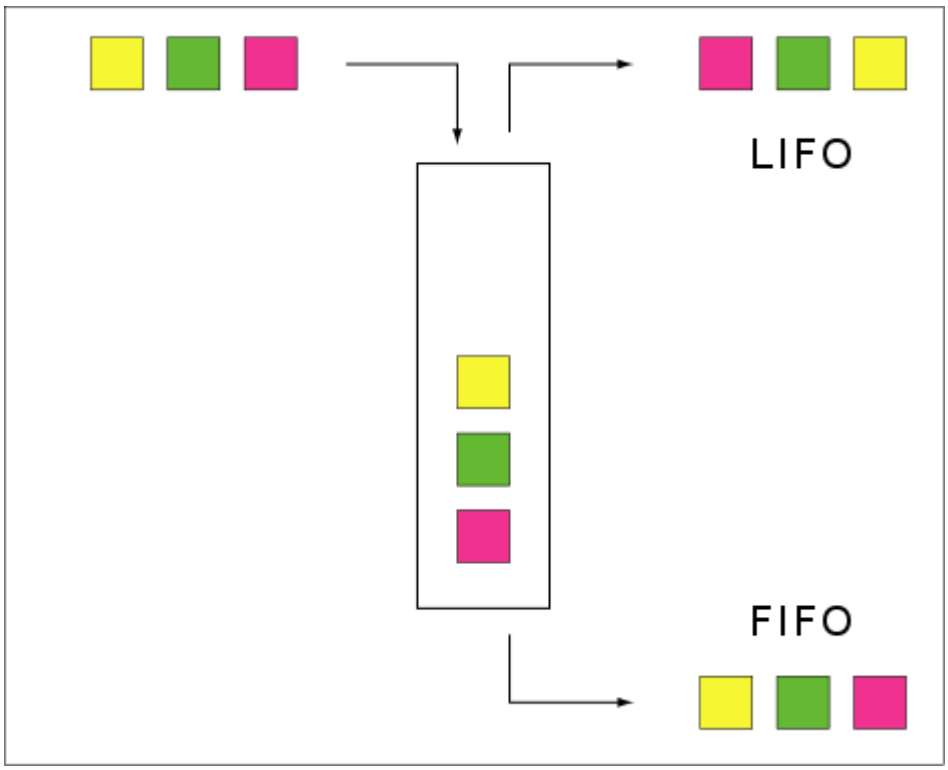
  Los tipos de Linked List son:

  - ✓ **Simple Linked List:** navegación siempre hacia adelante.
  - ✓ **Doubly Linked List:** navegación en ambos sentidos (hacia adelante o hacia atrás).
  - ✓ **Circular Linked List:** existe un puntero que apunta desde el primer elemento al elemento final y viceversa (del elemento final apunta al primer elemento).



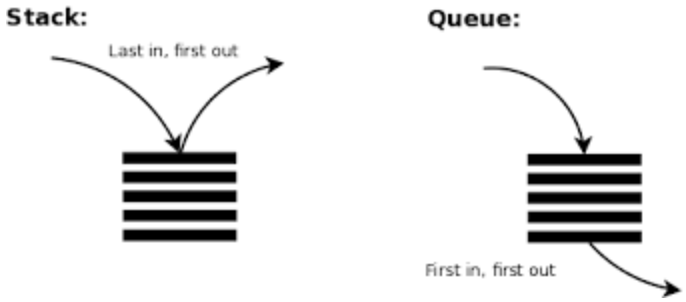
- **LIFO:** Last In First Out (Último en entrar es el primero en salir). Método usado en estructuras de datos, contabilidad de costes y teoría de pilas.

  FIFO: First In First Out (Primero en entrar es el primero en salir). Método usado en estructuras de datos, contabilidad de costes y teoría de colas.



- **Queue:** Estructura de datos basado en el método FIFO. Para insertar datos se usa la función enqueue() en donde añade los elementos al final(append) y la función dequeue() hace el método FIFO, es decir, elimina el último elemento en entrar.

  Stack: Estructura de datos basado en el método LIFO. Para insertar datos se usa la función push() en donde añade los elementos en el TOP y la función pop() hace el método LIFO, es decir, elimina el primer elemento en entrar.



1. Realizar la implementación del método delete en la clase LinkedList código de referencia "Laboratorio - LinkedLists".

Código

```python
from random import randint
from time import time
from uuid import uuid1

SIZE = int(2e6)

class Node:
    def __init__(self, value = None):
        self.value = None
        self.next = None
        self.setValue(value)

    def setValue(self, value):
        self.value = value

    def getValue(self):
        return self.value

    def getNext(self):
        return self.next

    def setNext(self, next):
        if not( isinstance(next, Node) or next is None):
            raise Exception("El tipo del atributo next no es del tipo de dato esperado.")
        self.next = next

    def __str__(self):
        return "Node({}) ⟶ {}".format(self.value, self.next if self.next else "X")

class LinkedList:
    def __init__(self, elements = []):
        self.head = None
        self.tail = None
        self.len = 0
        for e in elements:
            self.append(e)

    def __len__(self):
        return self.len

    def setHead(self, node):
        if not isinstance(node, Node):
            raise Exception("El tipo del atributo head no es del tipo de dato esperado.")
        self.head = node

    def setTail(self, node):
        if not isinstance(node, Node):
            raise Exception("El tipo del atributo tail no es del tipo de dato esperado.")
        self.tail = node
        if self.tail is not None:
            self.tail.setNext(None)

    def append(self, value):
        new_node = Node(value)
        if self.isEmpty():
            self.setHead(new_node)
            self.setTail(new_node)
        else:
            tail = self.tail
            tail.setNext(new_node)
            self.setTail(new_node)
        self.len += 1

    def find(self, value):
        node_result = self.head
        while node_result is not None and node_result.getValue() != value:
            node_result = node_result.getNext()
        return node_result

    def update(self, old_value, new_value):
        node_to_update = self.find(old_value)
        if node_to_update is not None:
            node_to_update.setValue(new_value)
```

```
    def isEmpty(self):
        return self.head is None

    def __str__(self):
        return str(self.head)

    def join(self, ll2):
        if not isinstance(ll2, LinkedList):
            raise Exception("El tipo del parametro ll2 no es del tipo de dato esperado.")
        if self.isEmpty():
            return ll2
        if not ll2.isEmpty():
            self.tail.setNext(ll2.head)
            self.tail = ll2.tail
            self.len = self.len + len(ll2)
        return self

    def delete(self, value):  # Nuevo método para eliminar un nodo dado su valor
        if self.isEmpty():
            return

        if self.head.getValue() == value:  # Caso especial si el valor está en la cabeza de la lista
            self.head = self.head.getNext()
            if self.head is None or self.head.getNext() is None:  # Si la lista tiene 1 o 0 nodos después de la eliminación
                self.tail = self.head
            return

        prev_node = self.head
        curr_node = self.head.getNext()
        while curr_node is not None and curr_node.getValue() != value:
            prev_node = curr_node
            curr_node = curr_node.getNext()

        if curr_node is not None:  # Si se encontró el nodo
            prev_node.setNext(curr_node.getNext())
            if curr_node.getNext() is None:  # Si el nodo a eliminar es la cola
                self.tail = prev_node

def main():
    ll = LinkedList([1,2,3])
    print(ll, len(ll))
    ll.append(500)
    print(ll, len(ll))
    ll2 = LinkedList([3, 4, 5])
    result = ll.join(ll2)
    print(result, len(result))
    arr1 = [ uuid1() for i in range(SIZE)]
    arr2 = [ uuid1() for i in range(SIZE)]
    ll1 = LinkedList(arr1)
    ll2 = LinkedList(arr2)
    t0 = time()
    arr3 = arr1 + arr2
    t1 = time()
    print("Classic Join {}".format(t1-t0))
    t0 = time()
    ll3 = ll1.join(ll2)
    t1 = time()
    print("New Join {}".format(t1-t0))

main()
```

Explicación DELETE

1. **Objetivo:**
   - Eliminar un nodo con un valor específico de la lista enlazada.

2. **Algoritmo:**
   - Recorrer la lista enlazada para encontrar el nodo con el valor dado.
   - Actualizar los punteros del nodo anterior y siguiente para omitir el nodo que se va a eliminar.
   - Si el nodo a eliminar es la cabeza o la cola, ajustar los punteros de la cabeza o la cola.

3. **Detalles:**
   - Como único parámetro: el valor a eliminar.
   - Verificar si la lista enlazada está vacía. Si lo está, no hay nada que eliminar.
   - Si no, iteramos a través de la lista:
     - Si el valor del nodo actual coincide con el valor objetivo:
       - Actualizar los punteros del nodo anterior y siguiente para excluir el nodo actual.
       - Si el nodo actual es la cabeza o la cola, ajustamos los punteros de la cabeza o la cola.
       - Disminuir la longitud de la lista enlazada.
       - Salir del método.
     - Si no, avanzar al siguiente nodo.
   - Si no se encuentra el valor objetivo, el método terminará sin cambios.

4. **Complejidad Temporal:**
   - Complejidad de O(n) en el peor caso, donde n es el número de nodos en la lista enlazada. Esto ocurre porque se debe recorrer toda la lista para encontrar el valor objetivo.

5. **Casos de Uso:**
   - Gestionar datos dinámicos, como la eliminación de elementos de una cola, pila o cualquier otra estructura de datos basada en listas enlazadas.

Ejemplo de Uso:
Se tiene una lista enlazada con los siguientes elementos: 1 -> 2 -> 3 -> 4 -> 5. Si se llama a delete(3), la lista resultante será: 1 -> 2 -> 4 -> 5.

2. Realizar la implementación de la lista doblemente enlazada usando como referencia el código fuente, extendiendo la definición de Nodo a tener un apuntador a aun nodo previo.

Código

```
from random import randint
from time import time
from uuid import uuid1

SIZE = int(2e6)

class Node:
    def __init__(self, value = None):
        self.value = None
        self.next = None
        self.prev = None  # Nuevo atributo para el nodo anterior
        self.setValue(value)

    def setValue(self, value):
        self.value = value

    def getValue(self):
        return self.value

    def getNext(self):
        return self.next

    def getPrev(self):  # Nuevo método para obtener el nodo anterior
        return self.prev

    def setNext(self, next):
        if not( isinstance(next, Node) or next is None):
            raise Exception("El tipo del atributo next no es del tipo de dato esperado.")
        self.next = next

    def setPrev(self, prev):  # Nuevo método para establecer el nodo anterior
        if not( isinstance(prev, Node) or prev is None):
            raise Exception("El tipo del atributo prev no es del tipo de dato esperado.")
        self.prev = prev

    def __str__(self):
        return "Node({}) <-> {}".format(self.value, self.next if self.next else "X")

class LinkedList:
    def __init__(self, elements = []):
        self.head = None
        self.tail = None
        self.len = 0
        for e in elements:
            self.append(e)

    def __len__(self):
        return self.len

    def setHead(self, node):
        if not isinstance(node, Node):
            raise Exception("El tipo del atributo head no es del tipo de dato esperado.")
        self.head = node

    def setTail(self, node):
        if not isinstance(node, Node):
            raise Exception("El tipo del atributo tail no es del tipo de dato esperado.")
        self.tail = node
        if self.tail is not None:
            self.tail.setNext(None)
```

```
    def append(self, value):
        new_node = Node(value)
        if self.isEmpty():
            self.setHead(new_node)
            self.setTail(new_node)
        else:
            tail = self.tail
            tail.setNext(new_node)
            new_node.setPrev(tail)  # Establece el nodo anterior del nuevo nodo
            self.setTail(new_node)
        self.len += 1

    def find(self, value):
        node_result = self.head
        while node_result is not None and node_result.getValue() != value:
            node_result = node_result.getNext()
        return node_result

    def update(self, old_value, new_value):
        node_to_update = self.find(old_value)
        if node_to_update is not None:
            node_to_update.setValue(new_value)

    def isEmpty(self):
        return self.head is None

    def __str__(self):
        return str(self.head)

    def join(self, ll2):
        if not isinstance(ll2, LinkedList):
            raise Exception("El tipo del parametro ll2 no es del tipo de dato esperado.")
        if self.isEmpty():
            return ll2
        if not ll2.isEmpty():
            self.tail.setNext(ll2.head)
            ll2.head.setPrev(self.tail)  # Establece el nodo anterior del primer nodo de ll2
            self.tail = ll2.tail
            self.len = self.len + len(ll2)
        return self

def main():
    ll = LinkedList([1,2,3])
    print(ll, len(ll))
    ll.append(500)
    print(ll, len(ll))
    ll2 = LinkedList([3, 4, 5])
    result = ll.join(ll2)
    print(result, len(result))
    arr1 = [ uuid1() for i in range(SIZE)]
    arr2 = [ uuid1() for i in range(SIZE)]
    ll1 = LinkedList(arr1)
    ll2 = LinkedList(arr2)
    t0 = time()
    arr3 = arr1 + arr2
    t1 = time()
    print("Classic Join {}".format(t1-t0))
    t0 = time()
    ll3 = ll1.join(ll2)
    t1 = time()
    print("New Join {}".format(t1-t0))

main()
```

Explicación

a. ¿Cómo afecta el tener un nodo previo a las demás funciones?

   Espacio adicional: Cada nodo en una lista enlazada doble necesita espacio adicional para almacenar el puntero al nodo previo.

   Tiempo de actualización: Cuando se inserta o se elimina un nodo, los punteros al nodo previo y al nodo siguiente deben actualizarse.

   Recorrido en ambas direcciones: Una ventaja de tener un nodo previo es que la lista enlazada doble puede recorrerse en ambas direcciones.

   Complejidad de implementación: Implementar una lista enlazada doble puede ser más complejo que una lista enlazada simple debido a la necesidad de mantener y actualizar los punteros al nodo previo.

   Operaciones de inserción y eliminación en cualquier lugar: En una lista enlazada simple, las operaciones de inserción y eliminación en cualquier lugar que no sea el principio de la lista requieren recorrer la lista desde el principio. En una lista enlazada doble, si ya se tiene un puntero a un nodo en la lista (por ejemplo, a través de una operación de búsqueda), se puede insertar o eliminar nodos antes o después de ese nodo de manera más eficiente.

b. ¿Cómo optimiza el nuevo atributo (previo) la implementación de las funciones insert/update/delete?

   Eliminación eficiente: En una lista enlazada simple, para eliminar un nodo, generalmente se necesita recorrer la lista desde el principio hasta encontrar el nodo previo al nodo que se quiere eliminar. En una lista enlazada doble, se puede eliminar un nodo de manera más eficiente porque se tiene un puntero directo al nodo previo.

   Inserción y actualización eficiente en ciertos casos: Si se necesita insertar un nuevo nodo antes de un nodo dado en una lista enlazada doble, se puede hacer de manera eficiente utilizando el puntero al nodo previo. En una lista enlazada simple, se tendría que recorrer la lista desde el principio para encontrar el nodo previo.

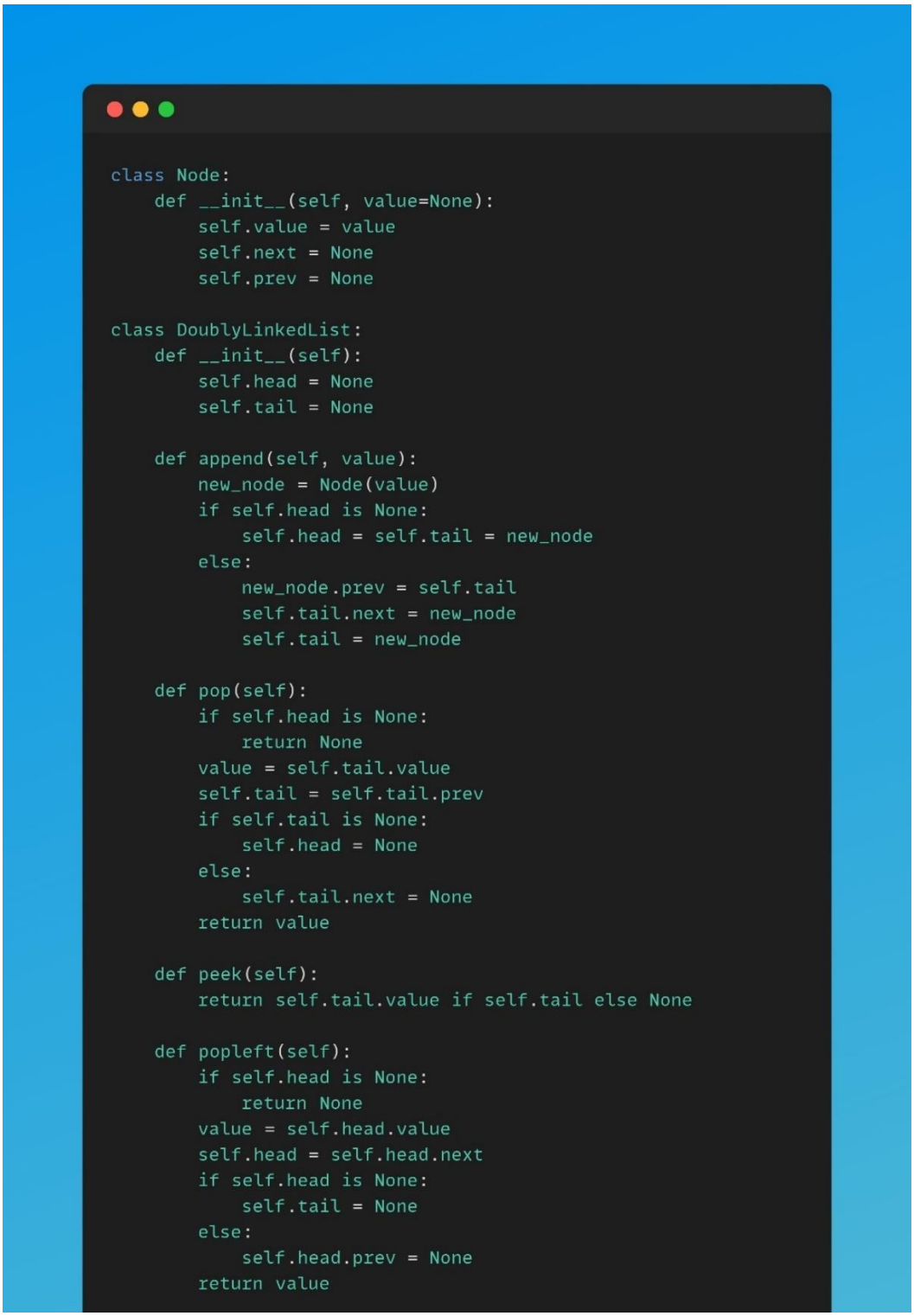
   Inserción eficiente al final: En una lista enlazada simple, para insertar un elemento al final de la lista, se debe recorrer toda la lista para llegar al último nodo. Pero en una lista enlazada doble, se puede acceder directamente al último nodo a través del puntero tail, lo que hace que la inserción al final sea más eficiente.

Eliminación eficiente al final: En una lista enlazada simple, para eliminar el último nodo, se debe recorrer toda la lista para llegar al penúltimo nodo. Pero en una lista enlazada doble, se puede acceder directamente al penúltimo nodo a través del puntero prev del nodo tail, lo que hace que la eliminación al final sea más eficiente.

Actualización eficiente de nodos adyacentes: Si se necesita actualizar los valores de los nodos adyacentes a un nodo dado, se puede hacer de manera eficiente en una lista enlazada doble porque se tienen punteros directos tanto al nodo previo como al nodo siguiente.

3. Realizar la implementación de Pila y Cola (Queue, Stack) en donde en su definición estructural se use cómo representación de datos una Lista doblemente enlazada . En las operaciones CRUD recuerde mantener la política de cada estructura (LIFO - FIFO).

Código



```python
class Node:
    def __init__(self, value=None):
        self.value = value
        self.next = None
        self.prev = None

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

    def append(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node

    def pop(self):
        if self.head is None:
            return None
        value = self.tail.value
        self.tail = self.tail.prev
        if self.tail is None:
            self.head = None
        else:
            self.tail.next = None
        return value

    def peek(self):
        return self.tail.value if self.tail else None

    def popleft(self):
        if self.head is None:
            return None
        value = self.head.value
        self.head = self.head.next
        if self.head is None:
            self.tail = None
        else:
            self.head.prev = None
        return value
```

```
class Stack:
    def __init__(self):
        self.list = DoublyLinkedList()

    def push(self, value):
        self.list.append(value)

    def pop(self):
        return self.list.pop()

    def peek(self):
        return self.list.peek()

class Queue:
    def __init__(self):
        self.list = DoublyLinkedList()

    def enqueue(self, value):
        self.list.append(value)

    def dequeue(self):
        return self.list.popleft()

    def peek(self):
        return self.list.head.value if self.list.head else None
def main():
    # Prueba de la pila
    print("Stack´s test:")
    stack = Stack()
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print(stack.pop())
    print(stack.peek())

    # Prueba de la cola
    print("\nQueue´s test:")
    queue = Queue()
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
    print(queue.dequeue())
    print(queue.peek())

main()
```

Explicación

1. **Lista Doblemente Enlazada (DoublyLinkedList):**
   - La clase DoublyLinkedList representa la estructura de datos básica. Tiene dos punteros: head (apuntando al primer nodo) y tail (apuntando al último nodo).
   - Los nodos (Node) contienen un valor (value) y dos punteros: next (al siguiente nodo) y prev (al nodo anterior).

2. **Pila (Stack):**
   - La clase Stack utiliza la política LIFO (Last In, First Out).
   - Para agregar un elemento a la pila, llamamos al método push(value), que agrega un nuevo nodo al final de la lista.
   - Para eliminar un elemento de la pila, llamamos al método pop(), que elimina el último nodo agregado y devuelve su valor.
   - El método peek() nos permite ver el valor del último elemento sin eliminarlo.

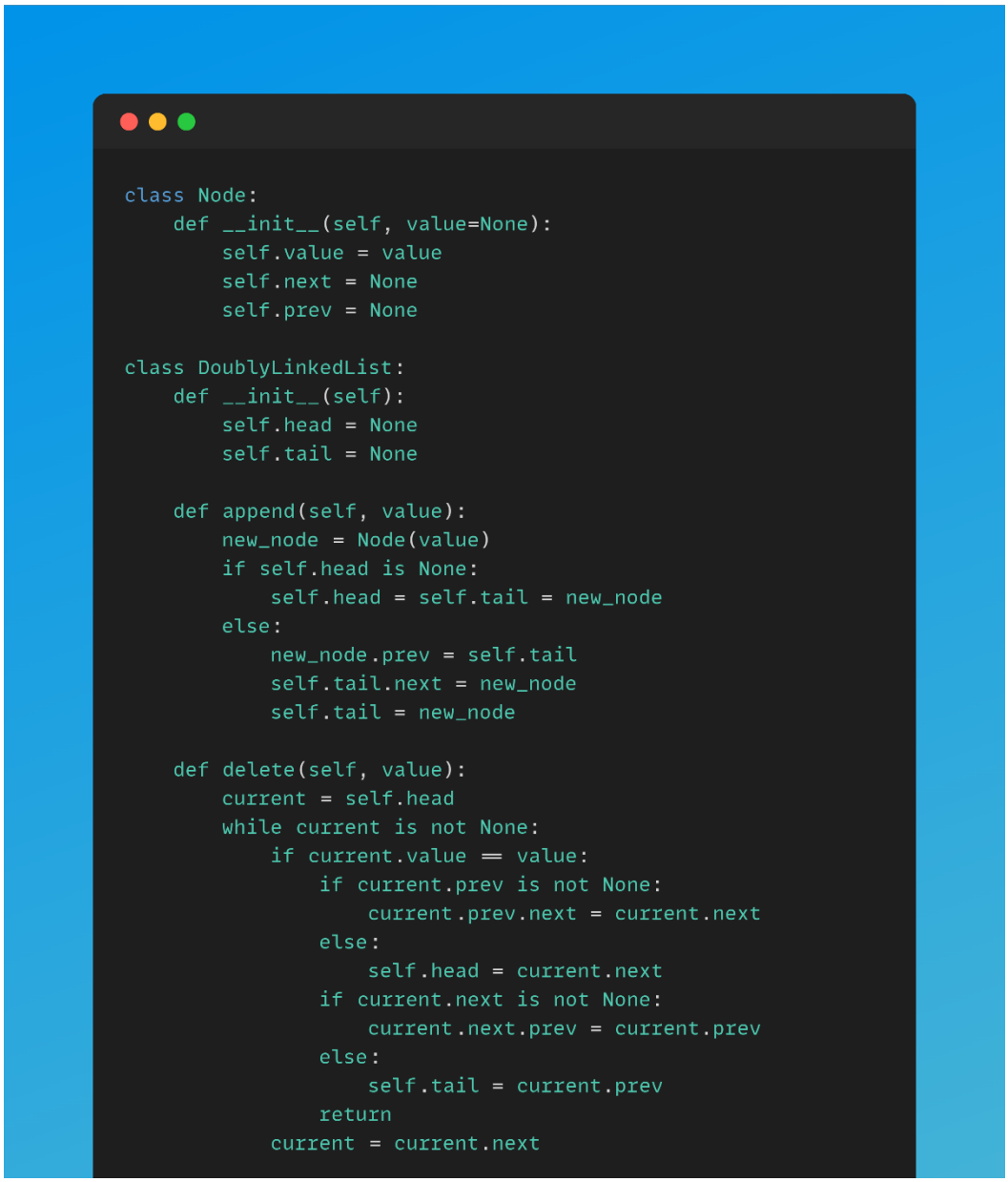
3. **Cola (Queue):**
   - La clase Queue utiliza la política FIFO (First In, First Out).
   - Para agregar un elemento a la cola, llamamos al método enqueue(value), que agrega un nuevo nodo al final de la lista.
   - Para eliminar un elemento de la cola, llamamos al método dequeue(), que elimina el primer nodo agregado y devuelve su valor.
   - El método peek() nos permite ver el valor del primer elemento sin eliminarlo.

4. **Complejidad Temporal:**
   - Tanto la pila como la cola tienen complejidad O(1) para las operaciones de inserción y eliminación, ya que siempre afectan al primer o último nodo.

5. **Ejemplo de Uso:**
   - Se puede simular una cola de un banco o el apilamiento de libros utilizando estas estructuras.

4. Para los puntos 1 y 2 escriba un programa principal con una demostración de uso de las estructuras (Simulando la cola de un banco o del cine, el apilamiento de libros, tareas).

Código



```python
class Node:
    def __init__(self, value=None):
        self.value = value
        self.next = None
        self.prev = None

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

    def append(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node

    def delete(self, value):
        current = self.head
        while current is not None:
            if current.value == value:
                if current.prev is not None:
                    current.prev.next = current.next
                else:
                    self.head = current.next
                if current.next is not None:
                    current.next.prev = current.prev
                else:
                    self.tail = current.prev
                return
            current = current.next
```

```
class Stack:
    def __init__(self):
        self.list = DoublyLinkedList()

    def push(self, value):
        self.list.append(value)

    def pop(self):
        value = self.list.tail.value
        self.list.delete(value)
        return value

class Queue:
    def __init__(self):
        self.list = DoublyLinkedList()

    def enqueue(self, value):
        self.list.append(value)

    def dequeue(self):
        value = self.list.head.value
        self.list.delete(value)
        return value

def main():
    # Simulando la cola de un banco
    print("Simulando la cola de un banco:")
    bank_queue = Queue()
    bank_queue.enqueue("Cliente 1")
    bank_queue.enqueue("Cliente 2")
    bank_queue.enqueue("Cliente 3")
    print(bank_queue.dequeue())  # Debería imprimir: Cliente 1
    print(bank_queue.dequeue())  # Debería imprimir: Cliente 2

    # Simulando el apilamiento de libros
    print("\nSimulando el apilamiento de libros:")
    book_stack = Stack()
    book_stack.push("Libro 1")
    book_stack.push("Libro 2")
    book_stack.push("Libro 3")
    print(book_stack.pop())  # Debería imprimir: Libro 3
    print(book_stack.pop())  # Debería imprimir: Libro 2

main()
```

Explicación

Clase Node:

- Representa un nodo en una lista doblemente enlazada.
- Cada nodo tiene tres atributos:
    - value: Almacena el valor del nodo.
    - next: Apunta al siguiente nodo en la lista.
    - prev: Apunta al nodo anterior en la lista.

Clase DoublyLinkedList:

- La clase DoublyLinkedList implementa una lista doblemente enlazada.
- Tiene dos atributos:
    - head: Apunta al primer nodo de la lista.
    - tail: Apunta al último nodo de la lista.
- Métodos:
    - append(value): Agrega un nuevo nodo con el valor especificado al final de la lista.
    - delete(value): Elimina el nodo con el valor especificado de la lista.

Clase Stack:

- Utiliza una instancia de DoublyLinkedList para simular una pila (estructura LIFO).
- Métodos:
    - push(value): Agrega un valor a la parte superior de la pila.
    - pop(): Elimina y devuelve el valor de la parte superior de la pila.

Clase Queue:

- Utiliza una instancia de DoublyLinkedList para simular una cola (estructura FIFO).
- Métodos:
    - enqueue(value): Agrega un valor al final de la cola.
    - dequeue(): Elimina y devuelve el valor del frente de la cola.

Complejidad Temporal:

- Tanto la pila como la cola tienen complejidad O(1) para las operaciones de inserción y eliminación, ya que siempre afectan al primer o último nodo.

Referencias y Bibliografía

deleteLinkedList/py doubleLinkedListprev/py

queue-Stack-LinkedList/py structureDemonstration/py

- https://dev.to/ronnymedina/estructura-de-datos-linked-list-lista-enlazada-2h9