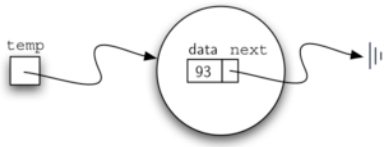
PANDUAN MATERI, PRAKTIKUM DAN TUGAS

Mata Kuliah: TPL1109 Algoritma dan Struktur Data

Topik: Linked List dalam Python

1. Apa Itu Linked List?

Linked List adalah struktur data linier yang terdiri dari node-node yang terhubung satu sama lain melalui pointer. Berbeda dengan struktur data seperti daftar bawaan (*list*) dalam Python yang berbasis array dinamis, linked list tidak memiliki alokasi memori yang berurutan, sehingga memungkinkan fleksibilitas dalam penambahan dan penghapusan elemen tanpa perlu memindahkan elemen lainnya. Linked list memungkinkan elemen-elemennya tersebar di berbagai lokasi memori, dengan setiap node menunjuk ke node selanjutnya.



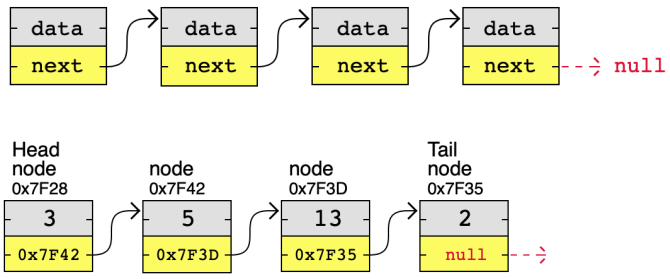
A Node Object Contains the Item and a Reference to the Next Node

A Typical Representation for a Node

2. Struktur Memori Linked List

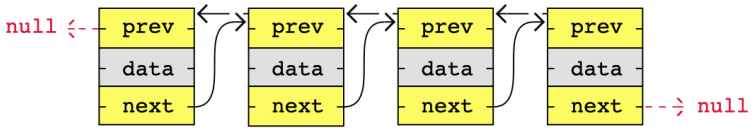
Setiap node dalam linked list terdiri dari dua bagian:

- Data : Menyimpan nilai yang akan disimpan.
- Pointer (next) : Menunjuk ke node berikutnya dalam daftar.



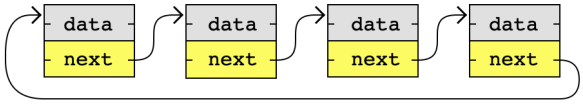
Jenis linked list berdasarkan strukturnya:

- Singly Linked List : Setiap node memiliki satu pointer ke node berikutnya. (**lihat contoh gambar dan kode diatas)**
- Doubly Linked List : Setiap node memiliki pointer ke node berikutnya dan sebelumnya.

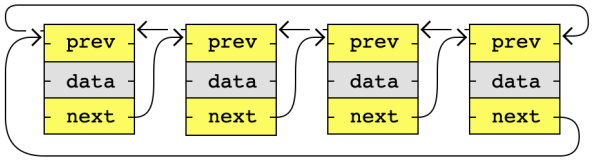


- Circular Linked List : Node terakhir mengarah ke node pertama, membentuk lingkaran.

The image below is an example of a singly circular linked list:



The image below is an example of a doubly circular linked list:



3. Kode Program dalam Python

Single Linkedlist dasar pada pyhthon :

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

class LinkedList:
    def __init__(self):
        self.head = None
        self.tail = None  # Tambahkan pointer tail

    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:  # Jika linked list kosong
            self.head = new_node
            self.tail = new_node  # Tail juga menunjuk ke node pertama
        else:
```

```
            self.tail.next = new_node  # Sambungkan tail ke node baru
            self.tail = new_node  # Update tail ke node baru

    def display(self):
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("null")

# Contoh Penggunaan
ll = LinkedList()
ll.insert_at_end(3)
ll.insert_at_end(5)
ll.insert_at_end(13)
ll.insert_at_end(2)
ll.display()
```

Pahami, ketik dan jalankan kode program tersebut

Double LinkedList pada pyhthon

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

      class DoublyLinkedList:
          def __init__(self):
              self.head = None
              self.tail = None  # Menyimpan node terakhir untuk traversing
      mundur

          def insert_at_end(self, data):
              new_node = Node(data)
              if not self.head:
                  self.head = new_node
                  self.tail = new_node
              else:
                  self.tail.next = new_node
                  new_node.prev = self.tail
                  self.tail = new_node

          def display_forward(self):
              print("\nTraversing forward:")
              temp = self.head
              while temp:
                  print(temp.data, end=" -> ")
```

```
                temp = temp.next
            print("null")

        def display_backward(self):
            print("\nTraversing backward:")
            temp = self.tail
            while temp:
                print(temp.data, end=" -> ")
                temp = temp.prev
            print("null")

# Contoh Penggunaan
dll = DoublyLinkedList()
dll.insert_at_end(3)
dll.insert_at_end(5)
dll.insert_at_end(13)
dll.insert_at_end(2)

dll.display_forward()
dll.display_backward()
```

Pahami, ketik dan jalankan kode program tersebut !

Circular singly linked list pada Python

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

class CircularSinglyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None  # Tambahkan pointer tail

    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:  # Jika linked list kosong
            self.head = new_node
            self.tail = new_node
            self.tail.next = self.head  # Circular link ke dirinya sendiri
        else:
            self.tail.next = new_node  # Sambungkan tail ke node baru
            self.tail = new_node  # Update tail ke node baru
            self.tail.next = self.head  # Circular link kembali ke head

    def display(self):
```

```
        if not self.head:
            print("List is empty")
            return

        print("Circular Linked List Traversal:")
        temp = self.head
        print(temp.data, end=" -> ")
        temp = temp.next

        while temp != self.head:
            print(temp.data, end=" -> ")
            temp = temp.next

        print("... (back to head)")

# Contoh Penggunaan
cll = CircularSinglyLinkedList()
cll.insert_at_end(3)
cll.insert_at_end(5)
cll.insert_at_end(13)
cll.insert_at_end(2)

cll.display()
```

Pahami, ketik dan jalankan kode program tersebut !

Circular doubly linked list in Python

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

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

    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            self.head.next = self.head
            self.head.prev = self.head  # Circular link ke dirinya
sendiri
        else:
            tail = self.head.prev  # Node terakhir dalam circular list
```

```
                tail.next = new_node
                new_node.prev = tail
                new_node.next = self.head  # Menghubungkan kembali ke head
                self.head.prev = new_node  # Menghubungkan head ke node baru

        def display_forward(self):
            if not self.head:
                print("List is empty")
                return

            print("\nTraversing forward:")
            temp = self.head
            print(temp.data, end=" -> ")
            temp = temp.next

            while temp != self.head:
                print(temp.data, end=" -> ")
                temp = temp.next
            print("... (back to head)")

        def display_backward(self):
            if not self.head:
                print("List is empty")
                return

            print("\nTraversing backward:")
            tail = self.head.prev  # Node terakhir
            temp = tail
            print(temp.data, end=" -> ")
            temp = temp.prev

            while temp != tail:
                print(temp.data, end=" -> ")
                temp = temp.prev
            print("... (back to tail)")

    # Contoh Penggunaan
    cdll = CircularDoublyLinkedList()
    cdll.insert_at_end(3)
    cdll.insert_at_end(5)
    cdll.insert_at_end(13)
    cdll.insert_at_end(2)

    cdll.display_forward()
    cdll.display_backward()
```

Pahami, ketik dan jalankan kode program tersebut !

4. Operasi pada Linked List
   - Penambahan (Insert) : Menambahkan node di awal, tengah, atau akhir linked list.
   - Penghapusan (Delete) : Menghapus node tertentu dari linked list.
   - Pencarian (Search) : Mencari nilai tertentu dalam linked list.

Cara Kerja Operasi pada Singly Linked List

1. Penambahan (Insert)

Pada Singly Linked List, kita dapat menambahkan node di tiga posisi berbeda:

- Di awal (insert at beginning)
- Di tengah (insert after a specific node)
- Di akhir (insert at end)

a. Menambahkan Node di Awal

Ketika kita menambahkan node di awal, langkah-langkahnya adalah:

1. Buat node baru.
2. Arahkan next dari node baru ke head saat ini.
3. Perbarui head agar menunjuk ke node baru.

Implementasi:

```
def insert_at_beginning(self, data):
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node
```

b. Menambahkan Node di Tengah (Setelah Node Tertentu)

Ketika kita menambahkan node setelah node tertentu:

1. Temukan node yang ingin ditambahkan setelahnya.
2. Buat node baru.
3. Arahkan next dari node baru ke next dari node sebelumnya.
4. Arahkan next dari node sebelumnya ke node baru.

Implementasi:

```
def insert_after(self, prev_node, data):
    if not prev_node:
        print("Node sebelumnya tidak boleh kosong")
        return
    new_node = Node(data)
    new_node.next = prev_node.next
    prev_node.next = new_node
```

c. Menambahkan Node di Akhir

Ketika kita menambahkan node di akhir:

1. Buat node baru.
2. Jika linked list kosong, jadikan node baru sebagai head.
3. Jika tidak kosong, cari node terakhir (next == None).
4. Arahkan next dari node terakhir ke node baru.

Implementasi:

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

2. Penghapusan (Delete)

Dalam Singly Linked List, penghapusan node bisa dilakukan dengan beberapa cara:

1. Menghapus node pertama
2. Menghapus node berdasarkan nilai tertentu
3. Menghapus node terakhir

a. Menghapus Node Pertama

1. Periksa apakah head kosong.
2. Jika tidak kosong, arahkan head ke node berikutnya.
3. Hapus node lama dari memori.

Implementasi:

```
def delete_first(self):
    if self.head:
        self.head = self.head.next
```

b. Menghapus Node Berdasarkan Nilai

1. Jika node pertama yang memiliki nilai tersebut, perbarui head.
2. Jika bukan node pertama, cari node yang memiliki nilai yang dihapus.
3. Arahkan next dari node sebelumnya ke next dari node yang dihapus.

Implementasi:

```
def delete_node(self, key):
    temp = self.head
    if temp and temp.data == key:
        self.head = temp.next
        temp = None
        return
    prev = None
    while temp and temp.data != key:
        prev = temp
        temp = temp.next
    if temp is None:
        return
    prev.next = temp.next
    temp = None
```

c. Menghapus Node Terakhir

1. Jika hanya ada satu node, perbarui head ke None.
2. Jika lebih dari satu node, cari node kedua terakhir.
3. Atur next dari node kedua terakhir ke None.

Implementasi:

```
def delete_last(self):
    if not self.head:
        return
    if not self.head.next:
        self.head = None
        return
    temp = self.head
    while temp.next.next:
        temp = temp.next
    temp.next = None
```

3. Pencarian (Search)

Untuk mencari nilai dalam Singly Linked List:

1. Mulai dari head.
2. Periksa apakah data pada node saat ini sesuai dengan yang dicari.
3. Jika ditemukan, kembalikan True.
4. Jika tidak, pindah ke node berikutnya sampai akhir daftar.
5. Jika tidak ditemukan, kembalikan False.

Implementasi:

```
def search(self, key):
    temp = self.head
    while temp:
        if temp.data == key:
            return True
        temp = temp.next
    return False
```

5. Latihan Praktikum

- Latihan 1: Implementasikan fungsi untuk menghapus node dengan nilai tertentu.

```
def delete_node(self, key):
  temp = self.head
  if temp and temp.data == key:
    self.head = temp.next
    temp = None
    return
  prev = None
  while temp and temp.data != key:
    prev = temp
    temp = temp.next
  if temp is None:
    return
  prev.next = temp.next
  temp = None
```

- Latihan 2: Buat kode Implementasikan Pencarian pada node tertentu Single Circular Linked List.
  Contoh Tampilan #1 :
  Masukkan elemen ke dalam Circular Linked List: 3, 7, 12, 19, 25
  Masukkan elemen yang ingin dicari: 12
  Elemen 12 ditemukan dalam Circular Linked List.

  Contoh Tampilan #2 :
  Masukkan elemen ke dalam Circular Linked List: 5, 10, 15, 20, 30
  Masukkan elemen yang ingin dicari: 25
  Elemen 25 tidak ditemukan dalam Circular Linked List.

  Contoh Tampilan #3 :
  Masukkan elemen ke dalam Circular Linked List: (Tidak ada elemen)
  Masukkan elemen yang ingin dicari: 10
  Circular Linked List kosong. Tidak ada elemen yang bisa dicari.

- Latihan 3: Implementasikan Pencarian pada node tertentu Double Linked List.
  Contoh Tampilan #1 :
  Masukkan elemen ke dalam Doubly Linked List: 2, 6, 9, 14, 20
  Masukkan elemen yang ingin dicari: 9
  Elemen 9 ditemukan dalam Doubly Linked List.

  *tampilan lain pada kasus lain, sama seperti pada tertentu Single Circular Linked List

- Latihan 4: Buat metode untuk menggabungkan dua single linked list menjadi satu linked list baru.
  Contoh Tampilan #1 :
  Masukkan elemen untuk Linked List 1: 1, 3, 5, 7
  Masukkan elemen untuk Linked List 2: 2, 4, 6, 8

  Linked List 1: 1 -> 3 -> 5 -> 7 -> null
  Linked List 2: 2 -> 4 -> 6 -> 8 -> null
  Linked List setelah digabungkan: 1 -> 3 -> 5 -> 7 -> 2 -> 4 -> 6 -> 8 -> null

  Contoh Tampilan #2 :
  Masukkan elemen untuk Linked List 1: 5, 15, 25
  Masukkan elemen untuk Linked List 2: (Tidak ada elemen)

  Linked List 1: 5 -> 15 -> 25 -> null
  Linked List 2: kosong
  Linked List setelah digabungkan: 5 -> 15 -> 25 -> null

- Latihan 5: Tambahkan metode untuk membalik (reverse) sebuah single linked list tanpa membuat linked list baru.

  Contoh Tampilan #1 :
  Masukkan elemen untuk Linked List: 1, 2, 3, 4, 5

  Linked List sebelum dibalik: 1 -> 2 -> 3 -> 4 -> 5 -> null
  Linked List setelah dibalik: 5 -> 4 -> 3 -> 2 -> 1 -> null

  Contoh Tampilan #2 :
  Masukkan elemen untuk Linked List: 10, 20, 30, 40

  Linked List sebelum dibalik: 10 -> 20 -> 30 -> 40 -> null
  Linked List setelah dibalik: 40 -> 30 -> 20 -> 10 -> null

  Contoh Tampilan #3 :
  Masukkan elemen untuk Linked List: 7

  Linked List sebelum dibalik: 7 -> null
  Linked List setelah dibalik: 7 -> null

- **Kumpulkan dalam 1 dokumen pdf, kode dan screenshoot program :**
  - **NIM berakhiran angka ganjil mengerjakan Latihan 1,3,5**
  - **NIM berakhiran angka genap mengerjakan Latihan 1,2,4**

6. Berikut adalah beberapa implementasi dan penggunaan linked list di dunia nyata:
   a. Manajemen Memori

   Sistem operasi menggunakan linked list untuk mengelola memori yang tersedia, terutama dalam manajemen memori yang dialokasikan secara dinamis. Linked list digunakan untuk melacak blok memori yang bebas atau yang telah dialokasikan, memungkinkan penambahan dan penghapusan blok memori dengan efisien tanpa perlu memindahkan data yang sudah ada.

   b. Pengelolaan Tab Browser

   Browser web menggunakan linked list untuk mengelola tab yang terbuka. Setiap tab diwakili sebagai node dalam linked list, memungkinkan pengguna untuk dengan mudah menambahkan, menghapus, dan berpindah antar tab tanpa mempengaruhi performa.

   c. Undo Functionality dalam Aplikasi

   Dalam aplikasi pengolah kata atau program desain grafis, linked list digunakan untuk implementasi fitur undo. Setiap operasi yang dilakukan oleh pengguna disimpan sebagai node dalam linked list. Ketika pengguna melakukan undo, aplikasi dapat kembali ke state sebelumnya dengan menelusuri linked list.

d. Pemutar Musik

Pemutar musik sering menggunakan linked list untuk mengelola daftar lagu dalam playlist. Linked list memudahkan penambahan, penghapusan, dan pemindahan lagu dalam daftar, serta memungkinkan navigasi mudah ke lagu sebelumnya atau berikutnya.

e. Alokasi Dinamis dalam Game Development

Dalam pengembangan game, linked list digunakan untuk mengelola objek-objek seperti karakter, peluru, atau musuh yang dibuat dan dihancurkan secara dinamis selama permainan berlangsung. Penggunaan linked list memungkinkan alokasi dan dealokasi memori yang efisien untuk objek-objek tersebut.

f. Implementasi Struktur Data Lain

Linked list digunakan sebagai blok bangunan untuk struktur data yang lebih kompleks seperti stack, queue, dan graph. Misalnya, linked list dapat digunakan untuk mengimplementasikan stack dan queue dengan operasi penambahan dan penghapusan elemen yang efisien.

g. Pengelolaan Database

Dalam beberapa implementasi database, linked list digunakan untuk mengelola catatan data yang tersimpan secara dinamis. Ini memungkinkan penambahan, penghapusan, dan pembaruan catatan dengan efisien tanpa perlu reorganisasi seluruh struktur data.

7. Kesimpulan

   Linked List adalah struktur data yang fleksibel dan efisien untuk manipulasi data dibandingkan dengan array. Pemahaman berbagai operasi dalam linked list, seperti insert, delete, dan traversal, akan membantu dalam implementasi algoritma yang lebih kompleks.

8. Referensi:

1. [W3Schools - Linked List](https://www.w3schools.com/dsa/dsa_theory_linkedlists.php)
2. [W3Schools - Linked List Memory](https://www.w3schools.com/dsa/dsa_theory_linkedlists_memory.php)
3. [W3Schools - Types of Linked List](https://www.w3schools.com/dsa/dsa_data_linkedlists_types.php)
4. [W3Schools - Linked List Operations](https://www.w3schools.com/dsa/dsa_algo_linkedlists_operations.php)
5. [Runestone Academy - Unordered List](https://runestone.academy/ns/books/published/pythonds/BasicDS/ImplementinganUnorderedListLinkedLists.html)

SUPLEMEN #1: Penjelasan Kode: Single Linked List Dasar dalam Python

1. Definisi Node

```
class Node:
  def __init__(self, data):
    self.data = data  # Menyimpan nilai dari node
    self.next = None  # Pointer ke node berikutnya
```

- Setiap node dalam linked list terdiri dari:
  - data: menyimpan nilai.
  - next: menunjuk ke node berikutnya (atau None jika node adalah yang terakhir).

2. Definisi Linked List

```
class LinkedList:
  def __init__(self):
    self.head = None  # Menyimpan node pertama dalam linked list
    self.tail = None  # Menyimpan node terakhir dalam linked list
```

- head menyimpan referensi ke node pertama dalam linked list.
- tail → Menunjuk ke node terakhir, mengurangi kebutuhan traversal saat menambahkan elemen baru.
- Saat pertama kali dibuat, linked list kosong (head = None).

3. Menambahkan Node di Akhir

```
def insert_at_end(self, data):
  new_node = Node(data)  # Membuat node baru
  if not self.head:  # Jika linked list kosong
    self.head = new_node
    self.tail = new_node  # Tail juga menunjuk ke node pertama
  else:
    self.tail.next = new_node  # Sambungkan tail ke node baru
    self.tail = new_node  # Update tail ke node baru
```

Cara Kerja:

1. Buat node baru dengan data yang diberikan.
2. Cek apakah linked list kosong (if not self.head):
   - Jika kosong, node baru menjadi head dan tail.
3. Jika linked list sudah berisi node:
   - Langsung sambungkan node baru ke tail.next (karena kita tahu posisi terakhirnya).
   - Perbarui tail agar menunjuk ke node baru.
4. Tanpa tail, kita harus melakukan traversal (while temp.next) untuk menemukan node terakhir

4. Menampilkan Isi Linked List

```
def display(self):
  temp = self.head  # Mulai dari node pertama
  while temp:  # Iterasi selama masih ada node
    print(temp.data, end=" -> ")  # Cetak data node
    temp = temp.next  # Pindah ke node berikutnya
```

```
    print("null")  # Menandakan akhir linked list
```

Cara Kerja:

1. Mulai dari head.
2. Iterasi melalui setiap node dalam linked list.
3. Cetak data node diikuti panah (->).
4. Ketika sampai akhir, cetak "null" untuk menunjukkan akhir dari linked list.

5. Contoh Penggunaan

```
# Membuat objek LinkedList
ll = LinkedList()

# Menambahkan elemen ke dalam linked list
ll.insert_at_end(3)
ll.insert_at_end(5)
ll.insert_at_end(13)
ll.insert_at_end(2)

# Menampilkan isi linked list
ll.display()
```

Hasil Output:

```
3 -> 5 -> 13 -> 2 -> null
```

Urutan Eksekusi

1. insert_at_end(3)
- head dan tail menunjuk ke node 3.
2. insert_at_end(5)
- tail.next = 5, tail diperbarui menjadi 5.
3. insert_at_end(13)
- tail.next = 13, tail diperbarui menjadi 13.
4. insert_at_end(2)
- tail.next = 2, tail diperbarui menjadi 2.
5. Traversal (display()) → Cetak 3 -> 5 -> 13 -> 2 -> null.

Kesimpulan :

✅ if not self.head: digunakan untuk mengecek apakah linked list kosong.
✅ Jika kosong, node pertama yang ditambahkan menjadi head.
✅ return digunakan untuk menghentikan eksekusi kode lebih lanjut setelah menambahkan node pertama.

SUPLEMEN #2: Penjelasan Kode: Doubly Linked List dalam Python

Kode ini mengimplementasikan Doubly Linked List (DLL), yang memungkinkan traversal maju dan mundur. Setiap node memiliki dua pointer:

- next → menunjuk ke node berikutnya.
- prev → menunjuk ke node sebelumnya.

1. Definisi Node

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

Penjelasan:

1. data → menyimpan nilai dalam node.
2. next → menunjuk ke node berikutnya dalam linked list.
3. prev → menunjuk ke node sebelumnya dalam linked list.

2. Definisi Doubly Linked List

```
class DoublyLinkedList:
  def __init__(self):
    self.head = None
    self.tail = None  # Menyimpan node terakhir untuk traversing mundur
```

Penjelasan:

4. head → menunjuk ke node pertama dalam linked list.
5. tail → menunjuk ke node terakhir, digunakan untuk traversal mundur.
6. Saat pertama kali dibuat, linked list masih kosong (head = None, tail = None).

3. Menambahkan Node di Akhir

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

Penjelasan:

- Buat node baru dengan data yang diberikan.
- Cek apakah linked list kosong (if not self.head):
  - Jika kosong, node pertama menjadi head sekaligus tail.
- Jika linked list sudah berisi node:
  - Tambahkan node baru setelah tail.
  - Atur prev dari node baru menunjuk ke tail lama.
  - Perbarui tail agar menunjuk ke node baru.

4. Menampilkan Isi Linked List (Maju)

```
def display_forward(self):
   print("\nTraversing forward:")
   temp = self.head
   while temp:
      print(temp.data, end=" -> ")
      temp = temp.next
   print("null")
```

Cara Kerja:

- Mulai dari head.
- Iterasi melalui setiap node menggunakan next.
- Cetak nilai data hingga mencapai node terakhir (None).
- Cetak "null" untuk menandakan akhir linked list.

Contoh Output Traversal Maju

Traversing forward:
3 -> 5 -> 13 -> 2 -> null

5. Menampilkan Isi Linked List (Mundur)

```
def display_backward(self):
   print("\nTraversing backward:")
   temp = self.tail
   while temp:
      print(temp.data, end=" -> ")
      temp = temp.prev
   print("null")
```

Cara Kerja:

- Mulai dari tail.
- Iterasi mundur melalui prev sampai mencapai None.
- Cetak nilai data hingga mencapai node pertama (head).
- Cetak "null" untuk menandakan akhir traversal.

Contoh Output Traversal Mundur

Traversing backward:
2 -> 13 -> 5 -> 3 -> null

6. Contoh Penggunaan

```
# Membuat objek DoublyLinkedList
dll = DoublyLinkedList()

# Menambahkan elemen ke dalam linked list
dll.insert_at_end(3)
dll.insert_at_end(5)
dll.insert_at_end(13)
dll.insert_at_end(2)

# Menampilkan isi linked list maju & mundur
dll.display_forward()
dll.display_backward()
```

Urutan Eksekusi

- insert_at_end(3) → head dan tail menunjuk ke node 3.
- insert_at_end(5) → 3 -> 5, tail pindah ke 5.
- insert_at_end(13) → 3 -> 5 -> 13, tail pindah ke 13.
- insert_at_end(2) → 3 -> 5 -> 13 -> 2, tail pindah ke 2.
- Traversal maju (display_forward()) → Cetak 3 -> 5 -> 13 -> 2 -> null.
- Traversal mundur (display_backward()) → Cetak 2 -> 13 -> 5 -> 3 -> null.

Kesimpulan :

✅ Doubly Linked List (DLL) memungkinkan traversal maju dan mundur menggunakan next dan prev.
✅ Metode insert_at_end() menambahkan node di akhir sambil menjaga hubungan antar node.
✅ Metode display_forward() & display_backward() memungkinkan navigasi dua arah dalam linked list.

SUPLEMEN #3: Penjelasan Kode: Circular Singly Linked List dalam Python

Kode ini mengimplementasikan Circular Singly Linked List (CSLL), yang berbeda dari Singly Linked List (SLL) karena node terakhir menunjuk kembali ke node pertama (head), sehingga membentuk lingkaran.

1. Definisi Node

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

Penjelasan:

- data → Menyimpan nilai dalam node.
- next → Pointer yang menunjuk ke node berikutnya.
- Sama seperti pada Singly Linked List, tetapi di dalam CSLL node terakhir akan menunjuk kembali ke head.

2. Definisi Circular Singly Linked List

```
class CircularSinglyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None  # Tambahkan pointer tail
```

Penjelasan:

- head → Menunjuk ke node pertama dalam linked list.
- tail → Menunjuk ke node terakhir, yang digunakan untuk menambahkan node dengan cepat tanpa traversal.
    - Jika tidak menggunakan tail, setiap kali ingin menambahkan node di akhir, kita harus melakukan traversal dari head ke node terakhir, yang menyebabkan O(n) kompleksitas waktu.
- Saat pertama kali dibuat, linked list masih kosong (head = None).

3. Menambahkan Node di Akhir

```
def insert_at_end(self, data):
    new_node = Node(data)
    if not self.head:  # Jika linked list kosong
        self.head = new_node
        self.tail = new_node
        self.tail.next = self.head  # Circular link ke dirinya sendiri
    else:
        self.tail.next = new_node  # Sambungkan tail ke node baru
        self.tail = new_node  # Update tail ke node baru
        self.tail.next = self.head  # Circular link kembali ke head
```

Penjelasan Cara Kerja:

1. Buat node baru dengan data yang diberikan.
2. Cek apakah linked list kosong (if not self.head):
   - Jika kosong, buat node pertama sebagai head dan tail.
   - Node pertama menunjuk ke dirinya sendiri (self.tail.next = self.head).
3. Jika linked list sudah memiliki node:
   - Langsung sambungkan node baru ke tail.next.
   - Perbarui tail agar menunjuk ke node baru.
   - Buat circular link dengan self.tail.next = self.head.
4. Keuntungan utama menggunakan tail → tidak perlu traversal untuk menemukan node terakhir saat menambahkan node baru.

4. Menampilkan Isi Circular Singly Linked List

```
def display(self):
    if not self.head:
        print("List is empty")
        return

    print("Circular Linked List Traversal:")
    temp = self.head
    print(temp.data, end=" -> ")
    temp = temp.next

    while temp != self.head:
        print(temp.data, end=" -> ")
        temp = temp.next

    print("... (back to head)")
```

Penjelasan Cara Kerja:

1. Cek apakah linked list kosong (if not self.head):
1. Jika kosong, tampilkan "List is empty".
2. Traversal mulai dari head:
2. Cetak data node pertama (head).
3. Lanjutkan ke node berikutnya (temp.next).
4. Loop berhenti ketika kembali ke head, karena ini adalah circular list.
1. Cetak "..." untuk menunjukkan bahwa daftar bersifat sirkular.

5. Contoh Penggunaan

```
# Membuat objek CircularSinglyLinkedList
cll = CircularSinglyLinkedList()

# Menambahkan elemen ke dalam linked list
cll.insert_at_end(3)
cll.insert_at_end(5)
cll.insert_at_end(13)
cll.insert_at_end(2)
```

```
# Menampilkan isi linked list
cll.display()
```

Urutan Eksekusi

1. insert_at_end(3)
- head dan tail menunjuk ke node 3.
- tail.next menunjuk ke head, membentuk circular link.
2. insert_at_end(5)
- tail.next = 5, tail diperbarui menjadi 5.
- tail.next menunjuk kembali ke head.
3. insert_at_end(13)
- tail.next = 13, tail diperbarui menjadi 13.
- tail.next menunjuk kembali ke head.
4. insert_at_end(2)
- tail.next = 2, tail diperbarui menjadi 2.
- tail.next menunjuk kembali ke head.
5. Traversal (display())
- Cetak 3 -> 5 -> 13 -> 2 -> ... (back to head).

Kesimpulan :

✅ Circular Singly Linked List (CSLL) memiliki struktur melingkar, di mana node terakhir menunjuk kembali ke head.
✅ Metode insert_at_end() menambahkan node di akhir sambil menjaga hubungan circular.
✅ Metode display() memastikan traversal berhenti setelah satu putaran penuh.

SUPLEMEN #4: Penjelasan Kode: Circular Doubly Linked List dalam Python

Kode ini mengimplementasikan Circular Doubly Linked List (CDLL), yang memungkinkan traversal maju dan mundur dalam daftar sirkular.
Dalam CDLL:

- Node terakhir terhubung kembali ke head.
- Node pertama (head) memiliki prev yang menunjuk ke node terakhir.
- Node terakhir (tail) memiliki next yang menunjuk kembali ke head.

1. Definisi Node

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

Penjelasan:

- data → Menyimpan nilai dalam node.
- next → Menunjuk ke node berikutnya dalam linked list.
- prev → Menunjuk ke node sebelumnya dalam linked list.

2. Definisi Circular Doubly Linked List

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

Penjelasan:

- head → Menunjuk ke node pertama dalam linked list.
- Saat pertama kali dibuat, linked list masih kosong (head = None).

3. Menambahkan Node di Akhir

```
def insert_at_end(self, data):
    new_node = Node(data)
    if not self.head:
        self.head = new_node
        self.head.next = self.head
        self.head.prev = self.head  # Circular link ke dirinya sendiri
    else:
        tail = self.head.prev  # Node terakhir dalam circular list
        tail.next = new_node
        new_node.prev = tail
        new_node.next = self.head  # Menghubungkan kembali ke head
        self.head.prev = new_node  # Menghubungkan head ke node baru
```

Cara Kerja:

1. Cek apakah linked list kosong (if not self.head):
    - Jika kosong, buat node baru sebagai head.
    - Buat circular link ke dirinya sendiri dengan:

        - self.head.next = self.head
        - self.head.prev = self.head
    2. Jika linked list sudah ada node:
        - Temukan node terakhir (tail = self.head.prev)
        - Tambahkan node baru setelah tail.
        - Buat circular link:
            - tail.next = new_node → Sambungkan tail ke node baru.
            - new_node.prev = tail → Sambungkan node baru ke tail.
            - new_node.next = self.head → Sambungkan node baru ke head.
            - self.head.prev = new_node → Sambungkan head ke node baru.

4. Traversal Maju (Forward Traversal)

```
def display_forward(self):
    if not self.head:
        print("List is empty")
        return

    print("\nTraversing forward:")
    temp = self.head
    print(temp.data, end=" -> ")
    temp = temp.next

    while temp != self.head:
        print(temp.data, end=" -> ")
        temp = temp.next
    print("... (back to head)")
```

Cara Kerja:

5. Cek apakah linked list kosong (if not self.head):
    - Jika kosong, tampilkan "List is empty".
6. Traversal mulai dari head:
    - Cetak data node pertama (head).
    - Lanjutkan ke node berikutnya (temp.next).
    - Loop berhenti ketika kembali ke head, karena ini adalah circular list.
7. Cetak "..." untuk menunjukkan bahwa daftar bersifat sirkular.

Contoh Output Traversal Maju
Traversing forward:
3 -> 5 -> 13 -> 2 -> ... (back to head)

5. Traversal Mundur (Backward Traversal)

```
def display_backward(self):
    if not self.head:
        print("List is empty")
        return
```

```
    print("\nTraversing backward:")
    tail = self.head.prev  # Node terakhir
    temp = tail
    print(temp.data, end=" -> ")
    temp = temp.prev

    while temp != tail:
        print(temp.data, end=" -> ")
        temp = temp.prev
    print("... (back to tail)")
```

Cara Kerja:

1. Cek apakah linked list kosong (if not self.head):
   - Jika kosong, tampilkan "List is empty".
2. Mulai traversal dari tail (self.head.prev).
3. Lanjutkan ke prev (mundur) sampai kembali ke tail.
4. Cetak "..." untuk menunjukkan bahwa daftar bersifat sirkular.

Contoh Output Traversal Mundur
Traversing backward:
2 -> 13 -> 5 -> 3 -> ... (back to tail)

6. Contoh Penggunaan

```
# Membuat objek CircularDoublyLinkedList
cdll = CircularDoublyLinkedList()

# Menambahkan elemen ke dalam linked list
cdll.insert_at_end(3)
cdll.insert_at_end(5)
cdll.insert_at_end(13)
cdll.insert_at_end(2)

# Menampilkan isi linked list maju & mundur
cdll.display_forward()
cdll.display_backward()
```

Urutan Eksekusi

1. insert_at_end(3) → head menunjuk ke node 3, yang menunjuk ke dirinya sendiri.
2. insert_at_end(5) → 3 <-> 5, 5.next menunjuk ke 3, 3.prev menunjuk ke 5.
3. insert_at_end(13) → 3 <-> 5 <-> 13, 13.next menunjuk ke 3, 3.prev menunjuk ke 13.
4. insert_at_end(2) → 3 <-> 5 <-> 13 <-> 2, 2.next menunjuk ke 3, 3.prev menunjuk ke 2.
5. Traversal maju (display_forward()) → Cetak 3 -> 5 -> 13 -> 2 -> ... (back to head).
6. Traversal mundur (display_backward()) → Cetak 2 -> 13 -> 5 -> 3 -> ... (back to tail).
7. Output Program
   Traversing forward:
   3 -> 5 -> 13 -> 2 -> ... (back to head)
   Traversing backward:
   2 -> 13 -> 5 -> 3 -> ... (back to tail)

Kesimpulan :

✅ Circular Doubly Linked List (CDLL) memiliki struktur melingkar dengan dua arah traversal.
✅ Node terakhir menunjuk ke head, dan head menunjuk ke tail, memungkinkan navigasi tanpa ujung.
✅ Metode insert_at_end() menambahkan node di akhir sambil menjaga hubungan circular.
✅ Metode display_forward() & display_backward() memastikan traversal dalam dua arah tanpa infinite loop.