Modul Praktikum

STRUKTUR DATA

Program Studi Matematika
Fakultas Sains dan Teknologi
UIN Maulana Malik Ibrahim Malang

2022

Modul 1 Struktur Data: Pengantar OOP

Pada praktikum kali ini, kita akan membahas tentang `class`, yang nantinya akan kita gunakan untuk membuat berbagai jenis struktur data. Sekaligus, kita juga akan membahas tentang *object-oriented programming* atau OOP (pemrograman berorientasi objek atau PBO), yaitu semacam "paradigma pemrograman" (gaya pemrograman) di mana kita sering berurusan dengan `class`.

Intinya, hari ini kita akan membahas tentang `class` dan serba-serbi (filosofi) penggunaannya.

Apa itu `class`? Apa itu OOP?

Di pertemuan sebelumnya, ketika belajar tentang tipe data di Python, kita sering menjumpai nama tipe data disertai istilah `class`. Sebelum memahami apa itu `class`, kita bisa paham dulu tentang konsep "objek".

Di Python (dan banyak bahasa pemrograman lainnya yang "mendukung OOP"), sebuah "objek" adalah sesuatu yang bisa memiliki variabel-variabel tersendiri (disebut atribut) serta fungsi-fungsi tersendiri (disebut *method*) di bawah satu nama yang sama (yaitu objek tersebut).

Kemudian, sebuah `class` adalah semacam *blueprint* untuk membuat objek. Ketika kita ingin membuat objek, kita harus membuat definisi `class` nya terlebih dahulu sebagai *blueprint* untuk objek tersebut. Barulah, setelah definisi `class` nya ada, kita bisa membuat objek sebanyak-banyaknya dari `class` yang sama.

Sebagai *blueprint* untuk membuat objek, suatu definisi `class` mencakupi atribut serta *method* yang akan terdefinisi untuk objek yang akan dibuat. Artinya, semua objek yang dibuat dari `class` yang sama itu akan memiliki "struktur" yang sama, baik variabel-variabel maupun fungsi-fungsi yang terkandung di dalam tiap objek.

(Itulah mengapa tipe data dianggap sebagai `class` di Python. Misalnya, untuk tipe data `str`, yaitu `<class 'str'>`, semua *string* di Python tentunya "memiliki sifat yang sama", seperti bisa di-*format* dengan *method* `.format`)

Agar lebih paham, mari kita coba membuat `class` pertama kita, yaitu `class Orang`, untuk menyimpan data orang yang terdiri dari nama dan umur. Kemudian, kita akan membuat beberapa objek, yaitu beberapa `Orang`, yang masing-masing bisa memiliki data nama dan umur tersendiri.

```
class Orang:
    def __init__(self, nama, umur):
        self.nama = nama
        self.umur = umur
```

Pada definisi `class Orang` di atas, kita baru merancang atribut apa saja yang akan terkandung dalam objek, yaitu `nama` dan `umur`.

- Pada baris pertama, kita menuliskan kata `class` untuk memulai suatu definisi `class` baru, diikuti dengan nama `class` nya (di sini namanya `Orang`).
- Pada baris kedua, kita memulai definisi suatu *method* istimewa yang bernama `__init__` yang dimulai dan diakhiri dengan dua garis bawah. *Method* yang satu ini harus selalu ada di tiap definisi `class`, dan istilahnya adalah *constructor*. Argumen yang masuk ke dalam *method* ini adalah `self` yang merujuk ke "diri sendiri" (objek yang bersangkutan), kemudian dua atribut yang bisa ditentukan ketika objek dibuat, yaitu `nama` dan `umur`
- Di dalam definisi `__init__` di atas (baris ketiga dan keempat), nilai `self.nama` dan `self.umur` akan dipasangkan menjadi `nama` dan `umur` yang "masuk ke dalam *method*" (yaitu ditentukan ketika objek dibuat).

Kalau baru pertama kali lihat, mungkin *syntax* definisi `class` rasanya sangat aneh dan asing. Tidak masalah, itu normal. Ketiknya pelan-pelan saja. Kalau belum begitu paham, juga tidak masalah, ikuti saja. Perlahan, kita akan terus-menerus memberi tambahan ke definisi `class Orang` tersebut agar lebih paham.

Semoga menjadi lebih jelas setelah melihat *syntax* pembuatan objek:

```
        orang1 = Orang("Bisma", 19)
        orang2 = Orang("Vero", 20)
```

Kemudian, kita bisa melihat atribut objek seperti berikut:

```
        print(orang1.nama)
        print(orang1.umur)
```

```
Bisma
19
```

```
        print(orang2.nama)
        print(orang2.umur)
```

```
Vero
20
```

Perhatikan bahwa masing-masing atribut diakses melalui objek yang bersangkutan. Terlihat kegunaan objek sebagai penampung beberapa variabel (atribut) di bawah satu nama yang sama.

Selain melihat, tentunya kita juga bisa melakukan *assignment*:

```
        orang1.umur = 21
        print(orang1.umur)
```

```
21
```

Bahkan, kita bisa melakukan variasi *assignment* lainnya seperti biasa, misalnya `+=`

```
        orang1.umur += 3
        print(orang1.umur)
```

```
24
```

Kalau dirasa perlu, kita dapat membuat fungsi yang akan menerima suatu objek `Orang` lalu akan mengubah data `umur`.`

```
        def ulangtahun(orang):
            orang.umur += 1
```

Sehingga, bisa digunakan seperti berikut:

```
        ulangtahun(orang1)
        print(orang1.umur)
```

```
25
```

Perhatikan bahwa objek di Python bersifat *pass-by-reference*! Artinya, apabila suatu objek dimasukkan ke dalam fungsi, kemudian dimodifikasi di dalam fungsi tersebut, maka modifikasi tersebut juga berdampak hingga di luar fungsi.

Definisi fungsi `ulangtahun` yang telah kita buat di atas sebenarnya bisa dimasukkan ke dalam definisi `class Orang` sebagai suatu *method*.

```
        class Orang:
            def __init__(self, nama, umur):
                self.nama = nama
                self.umur = umur
            def ulangtahun(self):
                self.umur += 1
```

Perhatikan, ini adalah pendefinisian ulang! Ini adalah definisi baru untuk `class Orang`. Sedangkan, objek-objek yang sudah kita buat sebelumnya masih menganut definisi yang lama. Sehingga, setelah ini, kita harus membuat ulang objek agar mengikuti definisi `class Orang` yang baru.

Perhatikan juga, ada sedikit perbedaan istilah pada fungsi `ulangtahun`: tadinya, objek yang masuk itu kita sebut `orang`, sekarang kita sebut `self`. Istilah `self` ini memang sudah menjadi kebiasaan di Python untuk merujuk ke diri sendiri, yaitu objek yang bersangkutan. Tiap definisi *method* selalu harus diawali dengan masuknya objek yang bersangkutan (yang biasa disebut `self`), sudah menjadi formalitas di Python.

Itulah mengapa, di definisi `__init__` seolah-olah ada tiga variabel yang masuk yaitu `self`, `nama`, dan `umur`, meskipun yang diperlukan ketika membuat objeknya hanyalah `nama` dan `umur`.

Mari kita buat ulang `orang1`:

```
        orang1 = Orang("Bisma", 19)
```

Kita bisa melihat atributnya:

```
        print(orang1.nama)
        print(orang1.umur)
```

```
Bisma
19
```

Kemudian, kita bisa menggunakan *method* `ulangtahun` yang telah kita buat, lalu melihat data umur terbaru:

```
orang1.ulangtahun()
print(orang1.umur)
```

```
20
```

Penggunaan *method* memang seperti itu, sangat mirip dengan mengakses atribut, bedanya adalah bahwa *method* berupa fungsi. Di sini, kita bisa melihat, baik atribut maupun *method* suatu objek itu sama-sama berada di bawah satu nama yang sama, yaitu objek yang bersangkutan (di sini, baik atribut `umur` maupun *method* `ulangtahun` diakses melalui `orang1`).

Kalau mau, kita bisa melakukannya lagi:

```
orang1.ulangtahun()
print(orang1.umur)
```

```
21
```

Tentu saja, kegunaan `class` tidak sebatas itu. Bahkan, ada semacam "paradigma pemrograman" (gaya pemrograman) di mana kita sering berurusan dengan `class`, yang disebut OOP. Agar lebih paham juga tentang `class` dan kegunaannya, kita akan mempelajari dasar-dasar OOP, yang tercakup oleh empat pilar (tiang) OOP.

Empat pilar OOP

Empat pilar OOP adalah:

1. *Encapsulation* (pembungkusan)
2. *Abstraction* (abstraksi; kebalikan dari "mendetail")
3. *Inheritance* (pewarisan sifat)
4. *Polymorphism* ("banyak bentuk")

Istilah prinsip *polymorphism* memang sulit diterjemahkan. Kita akan membahas masing-masing keempat prinsip OOP tersebut.

Encapsulation dan *Abstraction*

Sejauh ini, kita sudah merasakan bagaimana variabel (atribut) dan fungsi (*method*) sama-sama berada di bawah satu nama yang sama, yaitu objek yang bersangkutan. Seolah-olah, atribut dan *method* tersebut dibungkus ke dalam objek tersebut. Inilah yang dinamakan prinsip ***encapsulation*** atau pembungkusan.

Namun, ada juga konsep *data hiding*, di mana atribut objek sebaiknya diakses dan dimodifikasi melalui *method* saja. *Method* untuk memperoleh (mengakses) nilai atribut tertentu disebut *getter*, dan *method* untuk memasang nilai baru untuk atribut tertentu disebut *setter*.

Prinsip *data hiding* seringkali dianggap bagian dari prinsip *encapsulation* (tetapi terkadang dianggap bagian dari *abstraction* yang akan kita bahas selanjutnya).

Kita akan mendefinisikan ulang `class Orang` agar memiliki *getter* dan *setter* untuk atribut `umur`.

```
class Orang:
    def __init__(self, nama, umur):
        self.nama = nama
        self.umur = umur
```

```
    def ulangtahun(self):
        self.umur += 1
    def get_umur(self):
        return self.umur
    def set_umur(self, baru):
        self.umur = baru
```

Perhatikan bahwa *method* `get_umur` melakukan `return`. Penggunaannya akan mirip dengan fungsi seperti biasanya. Kemudian, *method* `set_umur` akan menerima satu input di dalam kurungnya (sedangkan `self` hanya untuk formalitas).

Kita bisa membuat objek seperti biasa...

```
orang1 = Orang("Bisma", 19)
```

Lalu kita bisa melihat umurnya seperti ini:

```
print(orang1.get_umur())
```

```
19
```

Atau bahkan kita bisa membuat variabel baru yang menyimpan umur yang diperoleh:

```
berapa_tahun = orang1.get_umur()
print(berapa_tahun)
```

```
19
```

Kemudian, kita bisa memasang nilai baru untuk atribut umur:

```
orang1.set_umur(30)
```

Lalu memperoleh kembali umur yang baru:

```
orang1.get_umur()
```

```
30
```

Sebenarnya, tujuan *getter* dan *setter* adalah untuk berjaga-jaga agar tidak terjadi hal yang aneh. Misalnya, saat ini, kita masih bisa memasang umur menjadi negatif:

```
orang1.umur = -5
print(orang1.umur)
```

```
-5
```

Kita dapat menambahkan *if statement* pada definisi *method* `set_umur` di definisi `class Orang` untuk mencegah umur dipasang menjadi negatif:

```
class Orang:
    def __init__(self, nama, umur):
        self.nama = nama
        self.umur = umur
    def ulangtahun(self):
```

```
            self.umur += 1
        def get_umur(self):
            return self.umur
        def set_umur(self, baru):
            if baru >= 0:
                self.umur = baru
            else:
                print("error: umur tidak bisa negatif")
```

Sehingga, setelah membuat objek, kita bisa mencoba:

```
orang1 = Orang("Bisma", 19)
```

```
orang1.set_umur(-5)
```

```
error: umur tidak bisa negatif
```

Dengan begitu, data `umur` masih aman:

```
orang1.get_umur()
```

```
19
```

Sedangkan, pemasangan umur menjadi bilangan yang tidak negatif tetap berjalan dengan lancar:

```
orang1.set_umur(25)
print(orang1.get_umur())
```

```
25
```

Apakah kemudian kita masih bisa menuliskan misalnya `orang1.umur = -5`? Masih bisa, tetapi setidaknya, sekarang dengan adanya *getter* dan *setter* untuk atribut `umur`, kita bisa menjadikan kebiasaan agar selalu menggunakan `get_umur` dan `set_umur` ketika ingin berurusan dengan data umur, tidak lagi melalui `self.umur`, agar terjamin tidak akan terjadi keanehan seperti itu. Biasanya, istilahnya, atribut `umur` disebut *private*, karena diharapkan tidak bisa diakses dari luar secara langsung, hanya boleh melalui *method*.

Bahkan, kita dapat menggunakan *getter* dan *setter* di dalam definisi *method* lainnya. Contohnya, yang tadinya *method* `ulangtahun` didefinisikan sebagai `self.umur += 1`, kita bisa menggantikannya dengan `get_umur` dan `set_umur`:

```
class Orang:
    def __init__(self, nama, umur):
        self.nama = nama
        self.umur = umur
    def ulangtahun(self):
        self.set_umur(self.get_umur() + 1)
    def get_umur(self):
        return self.umur
    def set_umur(self, baru):
        if baru >= 0:
            self.umur = baru
        else:
            print("error: umur tidak bisa negatif")
```

Pada definisi baru di atas untuk *method* `ulangtahun`, konsepnya sebagai berikut:

1. Peroleh umur saat ini dengan `self.get_umur`
2. Tambah satu
3. Hasil yang baru itu dijadikan umur yang baru menggunakan `self.set_umur`

Saat ini, `orang1` masih menggunakan definisi *method* `ulangtahun` yang lama. Mari kita buat objek baru dari definisi `class Orang` yang baru bernama `orang3`, agar bisa dibandingkan:

```
        orang3 = Orang("Bisma", 19)
        orang1.set_umur(19) # kita samakan dulu umurnya
```

Kemudian, kita gunakan *method* `ulangtahun` pada keduanya:

```
        orang1.ulangtahun()
        orang3.ulangtahun()
```

Kita bisa melihat umur baru masing-masing:

```
        print(orang1.get_umur())
        print(orang3.get_umur())
```

```
20
20
```

Ternyata hasilnya sama. Artinya, kedua cara mendefinisikan *method* `ulangtahun` itu memberikan hasil yang sama.

Perhatikan bahwa, dari segi penggunaan, untuk menambahkan satu ke data `umur`, kita tinggal memanggil *method* `ulangtahun`. Kita tidak perlu memikirkan internalnya seperti apa. Bahkan, kita bisa mengubah definisinya secara internal, tetapi cara penggunaannya dari luar tetap sama.

Selain itu, untuk memasang data umur baru tanpa pusing, kita bisa langsung menggunakan `set_umur`. Bahkan, kita tidak perlu mengkhawatirkan kasus umur negatif; *method* tersebut bisa langsung menanganinya. Sehingga, kapanpun kita ingin memasang data umur yang baru, kita tidak perlu lagi membuat *if statement* untuk memastikan umurnya tidak negatif, karena sudah ditangani oleh `set_umur`.

Kedua contoh *method* di atas menggambarkan bagaimana *method* bisa sangat mempermudah proses pemrograman kita dengan objek. Prinsip ***abstraction*** menekankan penggunaan *method* dengan cara seperti itu agar kita tidak perlu terlalu memusingkan detailnya. Misalnya, kita tidak perlu memusingkan cara mendefinisikan *method* `ulangtahun`, dan kita tidak perlu memusingkan kasus umur negatif berkat adanya *method* `set_umur`, pokoknya tinggal pakai. Lagipula, maksudnya "abstraksi" adalah kebalikan dari "mendetail".

Selain tidak pusing, manfaat lain dari *abstraction* adalah, kapanpun kita mau, kita bisa memodifikasi definisi *method* di definisi `class` nya saja, tanpa harus mengubah kode yang menggunakan *method* tersebut.

Bayangkan apabila tidak ada *method* `ulangtahun`, sehingga kita menjadi harus mengubah `self.umur += 1` menjadi `self.set_umur(self.get_umur() + 1)` di mana-mana. Betapa ribetnya.

Inheritance (pewarisan sifat)

Sebelum belajar tentang *inheritance*, mari kita buat satu *method* lagi yaitu `perkenalan`:

```
class Orang:
    def __init__(self, nama, umur):
        self.nama = nama
        self.umur = umur
    def ulangtahun(self):
        self.set_umur(self.get_umur() + 1)
    def get_umur(self):
        return self.umur
    def set_umur(self, baru):
        if baru >= 0:
            self.umur = baru
        else:
            print("error: umur tidak bisa negatif")
    def perkenalan(self):
        print("Halo, nama saya " + self.nama + " dan umur saya " + str(self.um
```

Seperti biasa, kita bisa membuat objek:

```
orang1 = Orang("Bisma", 19)
orang2 = Orang("Vero", 20)
```

Kemudian, kita bisa memanggil *method* `perkenalan`

```
orang1.perkenalan()
orang2.perkenalan()
```

```
Halo, nama saya Bisma dan umur saya 19 tahun.
Halo, nama saya Vero dan umur saya 20 tahun.
```

Lalu, misalnya, kita ingin membuat `class` baru yaitu `class Mahasiswa`, yang akan memiliki atribut tambahan yaitu NPM.

Tentunya, mahasiswa adalah orang, sehingga kita harapkan bahwa semua yang bisa dilakukan oleh objek dari `class Orang` juga bisa dilakukan oleh objek dari `class Mahasiswa`.

Untungnya, daripada harus *copy-paste* semua *method* yang ada di `class Orang` ke dalam definisi `class Mahasiswa`, kita tinggal memanfaatkan ***inheritance*** (pewarisan sifat), dengan *syntax* yang bisa dilihat di baris pertama di kode berikut:

```
class Mahasiswa(Orang):
    def __init__(self, nama, umur, NPM):
        self.nama = nama
        self.umur = umur
        self.NPM = NPM
```

Sesingkat itu! Kita tinggal menyediakan *constructor* `__init__` yang baru yang lebih sesuai untuk `class Mahasiswa`, karena adanya atribut baru yaitu NPM. Semua *method* lainnya akan tetap dimiliki oleh objek dari `class Mahasiswa` karena sudah diwariskan dari `class Orang`, hanya dengan menuliskan `class Mahasiswa(Orang)` pada baris pertama definisi `class Mahasiswa`.

`class` yang asli (di sini `class Orang`) biasa disebut *parent class*, *base class*, atau *superclass*, sedangkan `class` yang mewariskan (di sini `class Mahasiswa`) biasa disebut *child class*, *derived class*, atau *subclass*.

Kemudian, pembuatan objek dari `class Mahasiswa` dilakukan seperti biasa (jangan lupa, kali ini ada tiga atribut):

```
mhs1 = Mahasiswa("Bisma", 19, 2106635581)
```

Seperti biasa, kita bisa lihat isi atributnya satu per satu:

```
print(mhs1.nama)
print(mhs1.umur)
print(mhs1.NPM)
```

```
Bisma
19
2106635581
```

Semua *method* yang dimiliki oleh objek `Orang` itu juga dimiliki oleh objek `Mahasiswa`. Misalnya, kita bisa menggunakan *method* `ulangtahun` dan `get_umur`:

```
mhs1.ulangtahun()
print(mhs1.get_umur())
```

```
20
```

Kita juga bisa melakukan `perkenalan`

```
mhs1.perkenalan()
```

```
Halo, nama saya Bisma dan umur saya 20 tahun.
```

Namun, isi perkenalannya sama persis seperti objek `Orang`, bahkan tidak ada keterangan NPM. Bagaimana kalau kita mau mahasiswa melakukan perkenalan dengan NPM juga? Apakah kita bisa memodifikasi *method* ini khusus untuk `class Mahasiswa`? Jawabannya adalah bisa, berkat prinsip *polymorphism*.

Polymorphism ("banyak bentuk")

Setelah melakukan *inheritance*, seandainya ada *method* yang diwaris yang dirasa perlu diubah atau dibedakan dari *parent class*, kita tinggal mendefinisikan ulang *method* tersebut di dalam definisi *child class* yang bersangkutan.

Misalnya, kita bisa mendefinisikan ulang *method* `perkenalan` di dalam definisi `class Mahasiswa` agar berbeda dengan `perkenalan` di `class Orang`:

```
class Mahasiswa(Orang):
    def __init__(self, nama, umur, NPM):
        self.nama = nama
        self.umur = umur
        self.NPM = NPM
    def perkenalan(self):
        print("Perkenalkan, saya " + self.nama + " dengan NPM " + str(self.NPM
```

Kita sudah memiliki `orang1` sebagai objek dari `class Orang`, sehingga bisa kita bandingkan dengan objek dari `class Mahasiswa` yang perlu kita buat ulang:

```
mhs1 = Mahasiswa("Bisma", 19, 2106635581)
```

Sekarang kita lakukan `perkenalan` untuk masing-masing:

```
orang1.perkenalan()
mhs1.perkenalan()
```

```
Halo, nama saya Bisma dan umur saya 19 tahun.
Perkenalkan, saya Bisma dengan NPM 2106635581
```

Hasilnya berbeda, sesuai harapan. Namun, nama *method* nya tetap sama, yaitu `perkenalan`. Seolah-olah, *method* `perkenalan` ini adalah "*method* yang sama" tetapi "memiliki bentuk yang berbeda-beda", yaitu berbeda antara di `class Orang` dengan `class Mahasiswa`.

Bahkan, kalau mau, kita bisa membuat *child class* yang baru lagi dari `class Orang`, dan mendefinisikan ulang atau "menimpa" lagi *method* `perkenalan` untuk *child class* tersebut. Sehingga, *method* `perkenalan` ini seperti memiliki banyak bentuk.

"Banyak bentuk" itulah yang dimaksud dengan ***polymorphism***. Kita bisa melakukan *inheritance* berkali-kali, kemudian "menimpa" suatu *method* pada *child class* dengan definisi yang berbeda daripada di *parent class*.

Penerapan lain dari prinsip *polymorphism* adalah fitur yang bernama *operator overloading*, yang kebetulan dimiliki oleh Python dan sejumlah "bahasa OOP" lainnya (bahasa yang "mendukung OOP", yaitu memiliki fitur `class`, *inheritance* dan sebagainya sesuai dengan empat pilar OOP).

Operator overloading

Misalnya kita membuat `class Pecahan` yang terdiri dari atribut `pembilang` dan `penyebut`:

```
class Pecahan:
    def __init__(self, pembilang, penyebut):
        self.pembilang = pembilang
        self.penyebut = penyebut
```

Kita bisa membuat pecahan setengah seperti berikut:

```
frac1 = Pecahan(1, 2)
```

Kita bisa melihat isi atribut `pembilang` dan `penyebut`:

```
print(frac1.pembilang)
print(frac1.penyebut)
```

```
1
2
```

Misalnya kita ada pecahan lain...

```
frac2 = Pecahan(3, 5)
```

... alangkah indahnya kalau kita bisa menjumlahkannya begitu saja...

```
frac1 + frac2
```

```
TypeError: unsupported operand type(s) for +: 'Pecahan' and 'Pecahan'
```

Terjadi *error*, karena saat ini, operator + belum ada artinya untuk objek `Pecahan`.

Akan tetapi, ada *method* istimewa yang bisa kita definisikan agar operator + menjadi terdefinisi, lho! Namanya adalah `__add__`.

Secara matematis, penjumlahan pecahan bisa dituliskan seperti berikut:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

Sehingga, kita bisa mendefinisikan *method* `__add__` sebagai berikut:

```
class Pecahan:
    def __init__(self, pembilang, penyebut):
        self.pembilang = pembilang
        self.penyebut = penyebut
    def __add__(self, pecahan2):
        a = self.pembilang
        b = self.penyebut
        c = pecahan2.pembilang
        d = pecahan2.penyebut
        atas = a*d + b*c
        bawah = b*d
        hasil = Pecahan(atas, bawah)
        return hasil
```

Lalu, kita bisa membuat ulang kedua pecahan yang tadi, mencoba menjumlahkannya, dan melihat data atribut `pembilang` dan `penyebut` di hasil jumlahannya:

```
frac1 = Pecahan(1, 2)
frac2 = Pecahan(3, 5)
```

```
frac3 = frac1 + frac2
print(frac3.pembilang)
print(frac3.penyebut)
```

```
11
10
```

Wow, keren! Hasilnya benar ya!

Selain penjumlahan, kita bisa mendefinisikan banyak operator lainnya untuk `class`. Pendefinisian operator untuk `class` disebut *operator overloading* ("menimpa operator"), dan selalu melibatkan *method* istimewa atau ***magic methods*** (juga disebut *dunder methods* atau *double underscore methods*) yang sudah memiliki nama tertentu. Kebetulan, *constructor* yang dinamakan `__init__` juga termasuk *magic method*.

Kalian bisa membaca lebih lanjut tentang *operator overloading* dan *magic method* lainnya di link berikut:

https://www.geeksforgeeks.org/operator-overloading-in-python/

Modul 2 Struktur Data: Array, Searching, Sorting

Pada pertemuan ini, kita akan membahas tentang operasi pada array, termasuk melihat beberapa algoritma-algoritma searching dan sorting pada array.

Operasi pada array

Sebagian besar pembahasan di praktikum kali ini bisa menggunakan `list` biasa atau menggunakan *array* dari `numpy`, terutama materi searching dan sorting. Namun, untuk materi operasi pada *array*, kita akan menggunakan *array* dari `numpy`.

```
import numpy as np
```

Traversal

Traversal pada *array* adalah "mengunjungi" elemen *array* satu per satu, dari awal sampai akhir. Tujuannya bisa untuk print saja, atau untuk menjumlahkan, atau yang lain. Apapun tujuannya, kalau itu melibatkan mengunjungi elemen *array* satu per satu, maka itu termasuk traversal.

Kita bisa mendeklarasikan suatu *array* dengan ukurannya saja, kemudian mengisi elemennya satu-per-satu.

```
A = np.empty(5)
```

```
print(A) # isinya masih garbage value
```

```
[0.  0.5 1.  1.5 2. ]
```

```
A[0] = 5
A[1] = 20
A[2] = -3
A[3] = 7
A[4] = -11
```

```
print(A)
```

```
[  5.  20.  -3.   7. -11.]
```

Alternatifnya, kita bisa langsung saja menentukan elemen *array* sejak awal dibuat.

```
A = np.array([5, 20, -3, 7, -11])
print(A)
```

```
[  5  20  -3   7 -11]
```

Berikut beberapa contoh traversal pada *array*.

```
        for i in range(0, len(A)):
            print(A[i])
```

```
5
20
-3
7
-11
```

```
        sum = 0
        for i in range(0, len(A)):
            sum += A[i]
        print(sum)
```

```
18
```

“Insertion”

Array memiliki ukuran yang tetap. Terkadang, ketika kita membuat *array*, belum tentu keseluruhan *array* itu langsung kita gunakan semua. Bisa jadi, di awal kita hanya menggunakan sebagian saja, namun nantinya akan kita gunakan seutuhnya. Sehingga, untuk mengelola data yang kita simpan di dalam *array* (sebagai struktur data), perlu ada mekanisme “memasukkan” dan “menghapus” data pada *array*.

(Pembahasan “insertion” dan “deletion” pada *array* mungkin agak aneh, tetapi sangat masuk akal untuk berbagai struktur data yang akan kita pelajari ke depannya, sehingga kita bahas terlebih dahulu untuk *array*.

Misalkan kita hanya mendeklarasikan suatu *array*. Belum ada data yang dimasukkan, sehingga kita bisa menyimpan variabel untuk “ukuran” *array* saat ini adalah nol.

```
        B = np.empty(5)
        B_size = 0
```

Saat ini, *array* tersebut masih sepenuhnya berisi *garbage value*.

```
        print(B)
```

```
[13. 20.  3.  7. 11.]
```

Kita bisa memasukkan elemen, misalnya 13, seperti berikut.

```
        # insert 97
        B[B_size] = 97

        # update data "ukuran" array,
        # bertambah satu karena memasukkan satu elemen baru
        B_size += 1
```

Dengan begitu, *array* menjadi seperti ini:

```
        print(B)
```

```
[97. 20.  3.  7. 11.]
```

Perhatikan nilai variabel “ukuran” yang kita simpan:

```
        print(B_size)
```

```
1
```

Saat ini, baru satu elemen yang kita masukkan ke dalam *array*. Sehingga, semua elemen lainnya itu tidak kita anggap, karena masih berupa *garbage value* (data sampah).

```
        # insert -17
        B[B_size] = -17
        B_size += 1
```

```
        print(B)
```

```
[ 97. -17.   3.   7.  11.]
```

```
        print(B_size)
```

```
2
```

```
        # insert 43
        B[B_size] = 43
        B_size += 1
```

```
        print(B)
```

```
[ 97. -17.  43.   7.  11.]
```

```
        print(B_size)
```

```
3
```

“Deletion”

Selain memasukkan data, kita juga bisa menghapus data. Kalau kita hanya ingin menghapus elemen “terakhir” (di data kita yaitu 43), maka kita tinggal “melupakan” elemen tersebut (sehingga statusnya menjadi *garbage value*) dengan mengurangi variabel “ukuran”:

```
        # delete elemen "terakhir" (dari yang sudah kita isi)
        B_size = B_size - 1
```

```
        print(B)
```

```
[ 97. -17.  43.   7.  11.]
```

```
        print(B_size)
```

```
2
```

Memang *array* nya tidak berubah sama sekali, tapi ini masalah mindset (hehe). Tadinya, kita mengakui bahwa *array* berisi tiga buah data yang kita simpan, tetapi sekarang kita menganggap hanya berisi dua buah data. Sehingga, data ketiga yang tadi kita anggap data, itu sekarang menjadi *garbage value* yang bukan tanggung jawab kita.

Mari kita coba insert beberapa elemen lagi.

```
        # insert 53, -98, 71

        B[B_size] = 53
        B_size += 1

        B[B_size] = -98
        B_size += 1

        B[B_size] = 71
        B_size += 1
```

```
        print(B)
```

```
[ 97. -17.  53. -98.  71.]
```

```
        print(B_size)
```

```
5
```

Sekarang *array* sudah penuh. Bagaimana kalau misalnya kita ingin menghapus elemen pada indeks 2 (yaitu 53)? Kita perlu menggeser elemen indeks 3 menjadi indeks 2, kemudian indeks 4 menjadi indeks 3, sehingga "ukuran" *array* menjadi berkurang satu (elemen terakhir menjadi *garbage value*).

```
        # delete elemen pada indeks 2
        for i in range(2, len(B)-1):
            B[i] = B[i+1]
        B_size = B_size - 1
```

```
        print(B)
```

```
[ 97. -17. -98.  71.  71.]
```

```
        print(B_size)
```

```
4
```

Jangan lupa, sekarang "ukuran" data kita hanya empat buah data, sehingga elemen terakhir di situ (yang kebetulan juga 71) adalah *garbage value* yang tidak kita anggap.

Searching

Algoritma searching, seperti namanya, adalah algoritma yang digunakan untuk mencari sesuatu dalam suatu list. Umumnya, algoritma semacam ini memiliki 2 input, yaitu suatu "key" atau elemen yang ingin dicari, dan suatu *array* atau list tempat pencarian key tersebut.

Terdapat 2 algoritma umum untuk searching, yaitu:

- Linear Search
- Binary Search

Linear Search

Linear search adalah algoritma searching di mana setiap elemen pada list dibandingkan satu per satu dengan key. Pada algoritma ini, kita akan mencoba untuk mencari keberadaan key pada list, serta index dari key tersebut (jika ada). Kalau key tidak ditemukan, kita bisa return -1 (memang sudah tradisi untuk menandakan ketiadaan elemen pada *array*, lagipula mustahil ada indeks -1 pada *array*).

```
        def linear_search(arr, key):
            for i in range(0, len(arr)):
                if arr[i] == key:
                    return i

            # sampai sini, berarti elemen tidak ditemukan
            return -1
```

```
        A = [1, 5, 2, 3, 4, 8, 7, 6, 10, 9]
        linear_search(A, 8)
```

```
5
```

Binary Search

Binary search adalah algoritma searching dimana suatu list dicek apakah nilai tengahnya adalah key. Jika tidak, list dipecah dua dan searching dilanjut tergantung posisi key relatif dari nilai tengah tersebut (apakah lebih kecil atau lebih besar).

```
        def binary_search(arr, key):
            left_idx = 0
            right_idx = len(A)
            found = False
            while (not found) and (left_idx <= right_idx):
                center_idx = int( (left_idx + right_idx) / 2 )
                if arr[center_idx] == key:
                    return center_idx
                elif arr[center_idx] > key:
                    right_idx = center_idx - 1
                else:
                    left_idx = center_idx + 1
            # keluar loop berarti tidak ditemukan
            return -1
```

```
        A = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
        binary_search(A, 14)
```

Sorting

Terdapat 5 algoritma umum dalam sorting yang akan dijelaskan, yaitu:

- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort
- Merge Sort

Bubble Sort

Bubble sort adalah algoritma sorting yang cara kerjanya adalah dengan membandingkan elemen yang bersebelahan secara berurutan, lalu ditukar jika urutannya salah. Bubble sort melibatkan beberapa kali "pass", yaitu beberapa kali melihat *array* dari awal sampai akhir.

Tentunya, bubble sort akan berhenti ketika *array* sudah terurut. Namun, bagaimana cara mengetahui apakah *array* sudah terurut? Salah satu caranya, di tiap *pass*, kita bisa menganggap *array* sudah terurut (ditandai dengan variabel *boolean*), lalu melakukan *bubble sort*, dan apabila ada elemen yang masih belum terurut, maka ketika ditukar, kita menandai *array* tersebut belum terurut. Sedangkan, apabila semua elemen sudah terurut (tidak terjadi pertukaran), variabel *boolean* tetap bernilai `True`, sehingga *array* sudah terurut dan bubble sort sudah selesai. Untuk itu, digunakan while loop.

```
def bubble_sort_while(A):
    n = len(A)
    # di awal, array belum terurut
    selesai = False
    while (not selesai):
        # di awal pass, asumsi array sudah terurut
        selesai = True
        for i in range(0, n-1):
            # jika ada elemen yang belum terurut (perlu ditukar),
            if A[i] > A[i+1]:
                # tandai array belum terurut
                selesai = False
                # lalu tukar
                A[i], A[i+1] = A[i+1], A[i]
        # pass selesai
```

```
A = [1, 5, 2, 3, 4, 8, 7, 6, 10, 9]
bubble_sort_while(A)
print(A)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Sebenarnya, banyaknya pass tidak akan melebihi $(n-1)$. Sehingga, daripada menggunakan while loop dan menandai *array*, kita bisa menggunakan for loop saja, untuk pass ke-i.

```
def bubble_sort_for(A):
    n = len(A)
    # Lakukan pass sebanyak (n-1) kali, yaitu pass ke-i, i=0, 1, ..., (n-2)
```

```
    for i in range(n-1):
        # Iterasi untuk tiap elemen ke-j, j=0, 1, ..., (n-2)
        for j in range(n-1):
            # Apabila elemen ke-j ternyata lebih besar daripada yang setelahny
            if A[j] > A[j+1]:
                # Maka tukar kedua elemen agar urutannya benar
                A[j], A[j+1] = A[j+1], A[j]
```

```
A = [1, 5, 2, 3, 4, 8, 7, 6, 10, 9]
bubble_sort_for(A)
print(A)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Insertion Sort

Cara kerja dari insertion sort adalah dengan membandingkan elemen baru dengan elemen sebelumnya dan ditempatkan di tempat yang sesuai. Insertion sort mulai dari indeks ke-1, yang mana elemen pada indeks tersebut dibandingkan dengan indeks sebelumnya. Jika posisinya tidak sesuai, maka elemen ditukar, dan seterusnya hingga posisinya sesuai. Lalu iterasi dilanjutkan dengan elemen indeks ke-2, hingga elemen telah diiterasi semua.

```
def insertion_sort(A):
    n = len(A)
    # Untuk tiap elemen di array... (kecuali elemen paling pertama, indeks 0)
    for i in range(1, n):
        j = i
        # Selama elemen itu lebih kecil daripada elemen di sebelah kirinya,
        # tukar (geser elemen itu ke sebelah kirinya) agar menjadi terurut
        while A[j] < A[j-1]:
            A[j], A[j-1] = A[j-1], A[j]
            j -= 1 # j berkurang karena bergeser ke kiri
            # Kalau elemen sudah di ujung kiri array,
            # udah ga ada elemen di sebelah kirinya lagi, jadi keluar aja
            if j == 0:
                break
```

```
A = [1, 5, 2, 3, 4, 8, 7, 6, 10, 9]
insertion_sort(A)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Selection Sort

Selection sort melakukan sorting dengan memasukkan nilai minimum dari suatu list. Jika diberikan suatu list $A[0..(n-1)]$, maka algoritma mencari nilai minimum dari $A[0..(n-1)]$, lalu ditukar dengan elemen $A[0]$. Selanjutnya algoritma mencari nilai minimum dari $A[1..(n-1)]$, lalu ditukar dengan elemen $A[1]$, dan seterusnya.

```
def selection_sort(A):
    n = len(A)
    # Untuk tiap elemen ke-i, akan ditukarkan dengan elemen minimum yang
    # ada di sebelah kanannya
```

```
            for i in range(n-1):
                # Asumsi awal: elemen yang sedang dilihat (elemen ke-i) adalah minimum
                min_idx = i
                min_val = A[min_idx]

                # Periksa masing-masing elemen selanjutnya...
                for j in range(i+1, n):
                    # Kalau ternyata ketemu elemen yang lebih kecil lagi...
                    if A[j] < min_val:
                        # ... maka itu menjadi minimum yang terbaru
                        min_val = A[j]
                        min_idx = j
                # Ketika keluar for loop, sudah diperoleh elemen minimum sesungguhnya
                # Tukar elemen minimum dengan elemen ke-i
                A[i], A[min_idx] = A[min_idx], A[i]
```

```
        A = [1, 5, 2, 3, 4, 8, 7, 6, 10, 9]
        selection_sort(A)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Merge Sort

Merge sort melakukan sort dengan memecah list menjadi dua secara rekursif. Lalu sorting dilakukan dengan melakukan merge pada hasil pecahan list. Merge adalah proses pada dua list yang menyatukan dua list terurut menjadi satu list terurut. Merge dilakukan hingga list utuh kembali.

```
        def merge_sort(A):
            n = len(A)
            # Seandainya hanya berisi satu elemen, tidak perlu dilakukan apa-apa
            if len(A) > 1:
                # indeks middle (elemen tengah)
                m = int(n/2)
                # Array A dipisah menjadi A1 (sebelah kiri) dan A2 (sebelah kanan)
                A1 = A[:m]
                A2 = A[m:]
                # Lakukan merge sort pada keduanya
                merge_sort(A1)
                merge_sort(A2)

                # Di bawah ini adalah proses penggabungan dari A1 dan A2 yang
                # masing-masing sudah terurut

                i = 0 # indeks untuk A1
                j = 0 # indeks untuk A2
                k = 0 # indeks untuk array/list baru yang nantinya sudah terurut

                # Loop selama kedua array masih punya elemen yang
                # belum dimasukkan ke array/list baru
                while i < len(A1) and j < len(A2):
                    # Kalau ternyata elemen pada A1 yang lebih kecil...
                    if A1[i] <= A2[j]:
                        # ... maka itulah yang dimasukkan ke array/list baru
                        A[k] = A1[i]
                        i += 1 # lanjut ke elemen berikutnya untuk A1
```

```
                    # Selain itu, berarti elemen pada A2 yang lebih kecil...
                    else:
                        # ... maka itulah yang dimasukkan
                        A[k] = A2[j]
                        j += 1 # lanjut ke elemen berikutnya untuk A2
                    # Ukuran array baru sudah bertambah satu
                    k += 1
                # Keluar loop, berarti salah satu array sudah habis
                # Ada dua kemungkinan, yaitu A1 yang belum habis, atau A2 yang belum.
                # Sehingga keduanya perlu "dihabiskan"

                # Menghabiskan A1 kalau belum habis
                while i < len(A1):
                    A[k] = A1[i]
                    i += 1
                    k += 1

                # Menghabiskan A2 kalau belum habis
                while j < len(A2):
                    A[k] = A2[j]
                    j += 1
                    k += 1
```

```
A = [1, 5, 2, 3, 4, 8, 7, 6, 10, 9]
merge_sort(A)
print(A)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Quicksort

Secara keseluruhan, algoritma quicksort (yang bersifat rekursif) terdiri dari langkah berikut:

1. Apabila *array* kosong atau terdiri dari satu elemen, sorting selesai. Selain itu, lanjut ke langkah berikut.
2. Pilih salah satu elemen di *array* sebagai "pivot" (Bebas, yang penting konsisten. Biasanya elemen pertama. Kemungkinan lain: elemen tengah, elemen terakhir, dsb)
3. Lakukan "partisi", yaitu proses yang membuat kondisi *array* menjadi seperti berikut:

```
-------------------------------------------------------------
| semua elemen yang      | pivot | semua elemen yang      |
| lebih kecil dari pivot |       | lebih besar dari pivot |
-------------------------------------------------------------
```

4. Lakukan *quicksort* pada sebelah kiri pivot dan pada sebelah kanan pivot.

Untuk proses "partisi", ada dua cara utama untuk melakukannya (algoritma partisi), yaitu algoritma partisi Hoare dan algoritma partisi Lomuto.

Quicksort dengan partisi Hoare

```
def partition_hoare(A, left_idx, right_idx):
    # Buat "pointer" low dan high (simpan indeksnya saja)
    low_idx = left_idx
    high_idx = right_idx
```

```
# Diasumsikan array sudah terpartisi dengan baik (padahal belom hehe),
# - tugas low adalah memeriksa dari kiri (apakah benar sudah dipartisi),
# - tugas high adalah memeriksa dari kanan.
# Sudah terpartisi artinya:
# - sebelah kiri pivot adalah yang lebih kecil dari pivot
# - sebelah kanan pivot adalah yang lebih besar dari pivot

# Pilih indeks pivot, bebas, misal elemen paling pertama (paling kiri)
pivot_idx = left_idx
pivot_val = A[pivot_idx]

# Loop selama low belum melewati high
# (syarat ini sangat penting, hingga diperiksa berkali-kali)
while low_idx <= high_idx:

    # low lanjut ke kanan hingga menemukan elemen yang posisinya salah,
    # yaitu elemen yang nilainya lebih besar dari pivot
    while (low_idx <= high_idx) and not (A[low_idx] > pivot_val):
        low_idx += 1

    # high lanjut ke kiri hingga menemukan elemen yang posisinya salah,
    # yaitu elemen yang nilainya lebih kecil dari pivot
    while (low_idx <= high_idx) and not (A[high_idx] < pivot_val):
        high_idx -= 1

    # low dan high sama-sama menunjuk pada elemen yang posisinya salah,
    # keduanya akan menjadi benar kalau posisinya ditukar
    if low_idx <= high_idx:
        A[low_idx], A[high_idx] = A[high_idx], A[low_idx]

        # Apabila elemen pivot ternyata ikut ditukar,
        # pastikan data posisinya (pivot_idx) di-update.
        if pivot_idx == low_idx: # Apabila tadinya pivot di low,
            pivot_idx = high_idx # maka sekarang pivot di high.
        elif pivot_idx == high_idx: # Namun apabila tadinya pivot di high,
            pivot_idx = low_idx # maka sekarang pivot di low.

# Kalau sudah keluar loop, berarti low sudah melewati high;
# Sudah ketemu garis baginya, yaitu antara low dan high.
# Saat ini, sebelah kiri garis bagi sudah lebih kecil dari pivot,
# dan sebelah kanan garis bagi sudah lebih besar dari pivot.
# Sekarang kita tinggal menempatkan pivot pada garis bagi tersebut

# Tukar pivot dengan high kalau pivot di sebelah kiri high,
if pivot_idx <= high_idx:
    A[pivot_idx], A[high_idx] = A[high_idx], A[pivot_idx]
    pivot_idx = high_idx

# atau tukar pivot dengan low kalau pivot di sebelah kanan low
else:
    A[pivot_idx], A[low_idx] = A[low_idx], A[pivot_idx]
    pivot_idx = low_idx

# Partisi sudah selesai, return posisi pivot
# supaya jadi tahu di mana garis baginya
return pivot_idx
```

```
        def quicksort_hoare(A, left_idx=None, right_idx=None):
            # Kalau left_idx dan right_idx tidak diinput, otomatis menjadi None
            # dan kalau begitu, berarti sebenarnya quicksort mau dilakukan pada
            # keseluruhan array, sehingga ujung kiri adalah indeks 0 dan
            # ujung kanan adalah indeks terakhir (n-1 di mana n adalah panjang array)
            if left_idx == None:
                left_idx = 0
            if right_idx == None:
                right_idx = len(A) - 1

            # Ada if statement untuk memastikan ujung kiri dan ujung kanan masih wajar
            if left_idx < right_idx:
                pivot_idx = partition_hoare(A, left_idx, right_idx)
                quicksort_hoare(A, left_idx, pivot_idx-1)
                quicksort_hoare(A, pivot_idx+1, right_idx)
            # Kalau sewaktu-waktu menjadi tidak wajar, berarti array kosong, berarti
            # quicksort sudah selesai dan tidak perlu dilakukan apa-apa lagi
```

```
        A = [1, 5, 2, 3, 4, 8, 7, 6, 10, 9]
        quicksort_hoare(A)
        print(A)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Quicksort dengan partisi Lomuto

```
        def partition_lomuto(A, left_idx, right_idx):
            # Pilih elemen pivot, sepertinya untuk Lomuto harus elemen terakhir
            pivot_idx = right_idx
            pivot_val = A[pivot_idx]

            # Asumsi awal: semua elemen lebih besar dari nilai pivot,
            # sehingga "separator" atau "garis pemisah" ada di ujung kiri,
            # bahkan di sebelah kiri elemen pertama
            sep = left_idx - 1

            # Periksa tiap elemen...
            for j in range(left_idx, right_idx):
                # Kalau ternyata ada elemen yang tidak lebih besar dari pivot...
                if A[j] <= pivot_val:
                    # Majukan garis pemisah...
                    sep = sep + 1
                    # Lalu tukar elemen itu (yang seharusnya di sebelah kiri pivot),
                    # agar menjadi di (sebelah kiri) garis pemisah
                    A[sep], A[j] = A[j], A[sep]
                    # Nantinya, pivot akan diletakkan di posisi indeks sep+1.
                    # Data indeks "sep" menunjuk pada indeks terakhir yang
                    # elemennya lebih kecil dari pivot.

            # Keluar for loop, sekarang semua elemen sudah diperiksa,
            # indeks sep menunjuk pada elemen terakhir yang lebih kecil dari pivot.
            # Maka, pivot bisa diletakkan di posisi sep+1.
            # Tukar elemen pivot dengan elemen apapun yang sedang di sep+1.
            A[sep+1], A[pivot_idx] = A[pivot_idx], A[sep+1]
```

```
        # Sekarang, pivot ada di sep+1
        pivot_idx = sep+1

        # Partisi sudah selesai, return posisi pivot
        # supaya jadi tahu di mana garis baginya
        return pivot_idx
```

```
def quicksort_lomuto(A, left_idx=None, right_idx=None):
    if left_idx == None:
        left_idx = 0
    if right_idx == None:
        right_idx = len(A) - 1

    if left_idx < right_idx:
        pivot_idx = partition_lomuto(A, left_idx, right_idx)
        quicksort_lomuto(A, left_idx, pivot_idx - 1)
        quicksort_lomuto(A, pivot_idx + 1, right_idx)
```

```
A = [1, 5, 2, 3, 4, 8, 7, 6, 10, 9]
quicksort_lomuto(A)
print(A)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Perhatikan bahwa, meskipun algoritma partisi Hoare dan partisi Lomuto sangat berbeda, ketika di fungsi `quicksort` (`quicksort_hoare` dan `quicksort_lomuto`), kodenya sama, hanya berbeda di fungsi partisi yang digunakan.

Modul 3 Struktur Data: Graphviz, *Linked List*

Pada praktikum kali ini, kita akan membahas mengenai *linked list*, serta cara memvisualisasikannya menggunakan yang namanya Graphviz.

Sebelum mengikuti praktikum ini, ada baiknya kalian me-*review* kembali modul berikut:

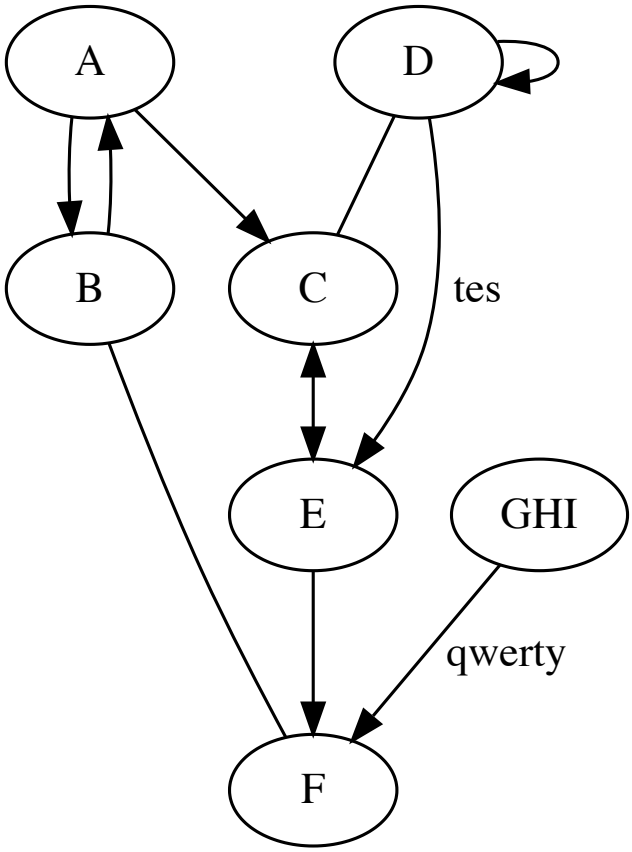
- Modul 1: Pengantar OOP

Untuk apa? Kita akan menyusun struktur data *linked list* menggunakan `class` :) semoga kalian sudah cukup paham tentang `class` yaa. Kalau belum pun, semoga kalian akan lebih paham setelah praktikum kali ini :D

Graphviz

Graphviz adalah semacam *software* yang bisa membuat visualisasi "graf" yang bagus. Mungkin di antara kalian belum semuanya kenal dengan graf, itu tidak masalah. Kurang lebih, suatu graf adalah kumpulan bulet-bulet (disebut simpul, *node*, atau *vertex*) yang disambung oleh "busur" (juga disebut *arc* atau *edge*), di mana tiap *edge* bisa berupa garis biasa atau berupa panah.

Berikut contoh graf yang digambar dengan Graphviz:



Lho, di mata kuliah Struktur Data kan ga ada graf. Untuk apa kita pelajari Graphviz?

Dengan Graphviz, kita bisa membuat visualisasi untuk berbagai struktur data nantinya, termasuk *linked list* hari ini. Kita bisa meminta Graphviz untuk membuat bentuk *node* yang tidak sederhana, termasuk bentuk *node* yang kita kenal di *linked list*, kemudian membuat *edge* yang berupa panah, sehingga kita benar-benar bisa menggambarkan suatu *linked list* :)

Instalasi Graphviz

Sebelum bisa menggunakan Graphviz, perlu di-*install* terlebih dahulu.

Di Google Colaboratory, kalian tinggal mengetik:

```
pip install graphviz
```

Sedangkan, apabila menggunakan Jupyter Notebook melalui Anaconda, buka Anaconda Prompt lalu ketik:

```
conda install graphviz
```

Tunggu instalasi selesai, barulah buka Jupyter Notebook dan ketik

```
pip install graphviz
```

Note:

- Apabila Anda menggunakan Jupyter Notebook tetapi tidak melalui Anaconda, langkah `conda install graphviz` bisa digantikan dengan menginstal Graphviz dari https://graphviz.gitlab.io/download/
- Untuk penulisan `pip`, ada kemungkinan kalian perlu mengetik `!pip` dengan tanda seru di awal. Biasanya tidak perlu, tapi kalau menjadi *error*, boleh dicoba dengan tanda seru.

Mengenal Graphviz: *node* dan *edge*

Setelah instalasi selesai, kita bisa import:

```
import graphviz as gv
```

Dengan Graphviz, ada dua jenis gambar graf yang bisa kita buat:

- `Digraph` (graf berarah, yaitu tiap *edge* bisa berupa panah maupun garis biasa)
- `Graph` (graf sederhana, yaitu tiap *edge* hanya bisa berupa garis biasa, bukan panah)

Karena `Digraph` lebih banyak fiturnya, kita akan membuat `Digraph` saja.

Sebagai contoh sederhana, kita bisa membuat `Digraph` yang terdiri dari dua *node* yaitu A dan B, dengan *edge* berupa panah yang menghubungkan A ke B. Kita buat objek `Digraph` terlebih dahulu:

```
graf1 = gv.Digraph()
```

Kemudian, kita bisa menambahkan *node* A dan B sebagai berikut:

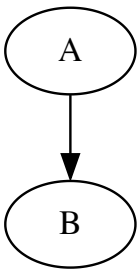
```
graf1.node("A")
graf1.node("B")
```

Selanjutnya, kita bisa membuat/menambahkan suatu *edge* dari A ke B, seperti berikut:

```
graf1.edge("A", "B")
```

Sekarang kita bisa lihat grafnya:

```
display(graf1)
```



Note: apabila fungsi `display` tidak dikenal, silakan import:

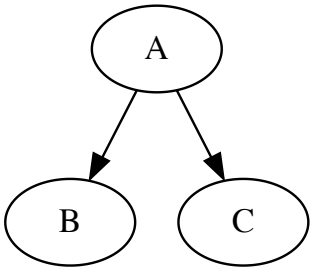
```
from IPython.display import display
```

Sebenarnya, kita bisa saja menambahkan *edge* baru tanpa membuat *node* terlebih dahulu. Contohnya, menambahkan *edge* dari A ke C (suatu *node* baru):

```
graf1.edge("A", "C")
```

Kita bisa lihat lagi:

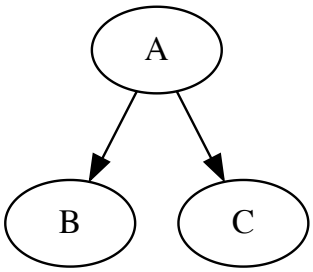
```
display(graf1)
```



Bahkan, kita bisa membuat ulang graf di atas dengan cara seperti berikut:

```
graf2 = gv.Digraph()
graf2.edge("A", "B")
graf2.edge("A", "C")
```

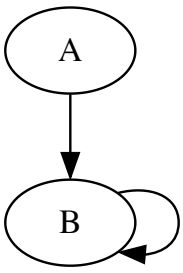
```
display(graf2)
```



Menariknya, kita bisa saja membuat panah yang menunjuk ke dirinya sendiri.

```
graf3 = gv.Digraph()
graf3.edge("A", "B")
graf3.edge("B", "B")
```

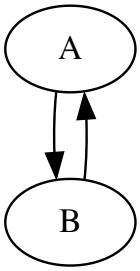
```
display(graf3)
```



Kita juga bisa membuat dua panah berlawanan arah di antara dua *node* seperti berikut:

```
graf4 = gv.Digraph()
graf4.edge("A", "B")
graf4.edge("B", "A")
```

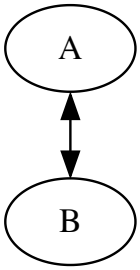
```
display(graf4)
```



Membuat satu panah yang dua arah juga bisa, dengan menentukan `dir` atau *direction* dari *edge* tersebut menjadi `"both"` seperti berikut:

```
graf5 = gv.Digraph()
graf5.edge("A", "B", dir="both")
```

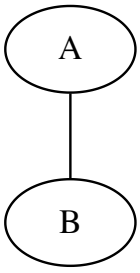
```
display(graf5)
```



Daripada panah, kita juga bisa membuat *edge* berupa garis biasa, dengan `dir="none"` (bukan `None` ya!)

```
graf6 = gv.Digraph()
graf6.edge("A", "B", dir="none")
```

```
display(graf6)
```

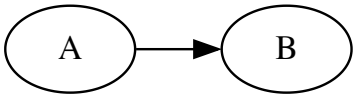


Sejauh ini, grafnya selalu cenderung "dari atas ke bawah". Daripada seperti itu, kita bisa mengubahnya menjadi kiri ke kanan untuk keseluruhan graf. Caranya, kita memasang `graph_attr` atau atribut graf, berbentuk `dict`, dan di dalamnya kita buat `"rankdir": "LR"` (*left-right*) seperti di bawah ini.

Setelah objek `Digraph` dibuat, barulah tiap *edge* yang kita tambahkan akan dari kiri ke kanan.

```
graf7 = gv.Digraph(graph_attr={"rankdir": "LR"})
graf7.edge("A", "B")
```

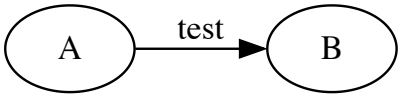
```
display(graf7)
```



Selain *node* diberi nama, *edge* juga bisa diberi keterangan, lho! Caranya, pasang nilai `label` ketika membuat *edge* baru:

```
graf8 = gv.Digraph(graph_attr={"rankdir": "LR"})
graf8.edge("A", "B", label="test")
```

```
display(graf8)
```



Sebenarnya, di dalam suatu *node*, ada yang namanya `name` (atau ID) dan ada juga yang disebut `label`.

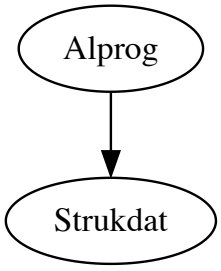
- `label` adalah tulisan yang tampil di gambar pada *node* tersebut
- `name` atau ID adalah sebutan yang dikenal oleh Graphviz ketika misalnya ingin membuat *edge*

Selama ini, yang kita tentukan adalah `name`. Kebetulan, khusus *node*, apabila `label` tidak ditentukan, maka otomatis akan diambil dari `name`.

Berikut ini, kita bisa coba menentukan `name` dan `label` sekaligus ketika membuat *node*:

```
graf9 = gv.Digraph()
graf9.node("matkul1", label="Alprog")
graf9.node("matkul2", label="Strukdat")
graf9.edge("matkul1", "matkul2")
```

```
display(graf9)
```



Perlu dicatat, apabila kita menambahkan *edge* sekaligus membuat *node* baru, kita tidak bisa memasang `label` untuk *node* baru tersebut.

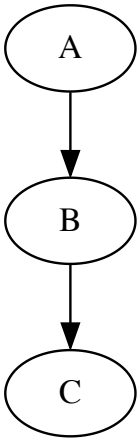
Sehingga, apabila kalian ingin membuat *node* dengan `label` tertentu, yang nantinya akan disambung ke *node* lain dengan *edge*, maka sebaiknya *node* baru tersebut dibuat dengan `.node()` terlebih dahulu, barulah `name` nya digunakan ketika membuat `.edge()`

Selain itu, bahkan graf itu sendiri juga bisa memiliki nama, yang ditentukan ketika membuat objek grafnya.

```
graf10 = gv.Digraph("Nama graf")
graf10.edge("A", "B")
```

```
        graf10.edge("B", "C")
```

```
        display(graf10)
```

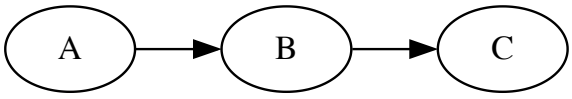


Coba letakkan *mouse* kalian pada gambarnya selama beberapa detik. Akan muncul tulisan “Nama graf”. (Kalau tidak muncul, coba klik kanan dulu, pencet “Open image in New Tab” atau semacamnya.)

Apabila kalian ingin menentukan misalnya `rankdir`, tuliskan setelah nama grafnya.

```
        graf11 = gv.Digraph("Graf ke kanan", graph_attr={"rankdir": "LR"})
        graf11.edge("A", "B")
        graf11.edge("B", "C")
```

```
        display(graf11)
```



Import/export, bahasa DOT, file `.gv`

Sebenarnya, Graphviz melibatkan yang namanya bahasa DOT (dibaca “dot”), yaitu semacam “bahasa komputer” untuk mendeskripsikan graf, yang kemudian diolah oleh Graphviz menjadi gambar.

(Sebenarnya, bahasa DOT mudah dipahami dan bisa kalian pelajari sendiri kalo iseng :D)

Tiap kali kita membuat graf baru dengan Graphviz melalui Python ini, Graphviz selalu menyusun bahasa DOT terlebih dahulu, baru mengolah bahasa DOT tersebut menjadi gambar.

Kita bisa melihat bahasa DOT untuk tiap graf melalui atribut `.source` seperti berikut:

```
        print(graf11.source)
```

```
digraph "Graf ke kanan" {
    graph [rankdir=LR]
    A -> B
    B -> C
}
```

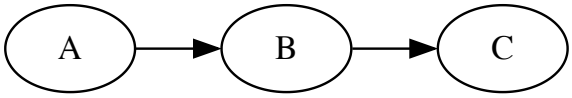
Kemudian, kita bisa memasukkan bahasa DOT tersebut ke dalam semacam *software* yang bisa mengolah bahasa DOT menjadi gambar. Contohnya adalah *link* berikut:

https://dreampuf.github.io/GraphvizOnline/

Sebaliknya, dari bahasa DOT, Graphviz juga bisa membuat objek `Digraph` misalnya, menggunakan `graphviz.Source()` seperti berikut:

```
        graf12 = gv.Source("""
        digraph "Graf ke kanan" {
            graph [rankdir=LR]
            A -> B
            B -> C
        }
        """)
```

```
        display(graf12)
```



Selain import seperti itu, baik bahasa DOT maupun gambar yang dibuat oleh Graphviz bisa di-*export* dengan menetapkan `.format` terlebih dahulu (misalnya "svg" atau "png"), lalu menggunakan `.render()` sebagai berikut:

```
        graf11.format = "svg"
        graf11.render()
```

```
'Graf ke kanan.gv.svg'
```

Seperti di Modul 3 kemarin ketika membahas I/O, ada *file* baru yang muncul.

- Apabila menggunakan Google Colaboratory, silakan tekan tombol folder di sebelah kiri.
- Apabila menggunakan Jupyter Notebook, silakan periksa folder yang di dalamnya ada *file* `.ipynb` yang sedang kalian gunakan.

Akan muncul dua *file* baru, yaitu:

1. `Graf ke kanan.gv`
2. `Graf ke kanan.gv.svg`

File pertama adalah *file* `.gv` (Graphviz) yang mengandung bahasa DOT yang disusun sebelum diolah menjadi gambar. *File* kedua adalah *file* gambar yang diolah, dalam format sesuai dengan yang kita tentukan.

Kita bisa membaca isi `Graf ke kanan.gv` sebagaimana kita membaca isi *text file*:

```
        with open("Graf ke kanan.gv", "r") as isi:
            print(isi.read())
```

```
digraph "Graf ke kanan" {
    graph [rankdir=LR]
    A -> B
```

```
    B -> C
}
```

Selain itu, perhatikan bahwa nama *file* nya sesuai dengan nama graf yang kita tentukan ketika membuat objek `graf11` tadi. Kalau lupa, kita bisa memeriksa nama graf melalui atribut `.nama`

```
print(graf11.name)
```

```
Graf ke kanan
```

Dengan atribut itu pula, kita bisa mengubah nama grafnya:

```
graf11.name = "Nama baru"
```

Sehingga, ketika misalnya Graphviz menyusun bahasa DOT, akan digunakan nama yang baru:

```
print(graf11.source)
```

```
digraph "Nama baru" {
    graph [rankdir=LR]
    A -> B
    B -> C
}
```

Variasi *node* dengan *HTML-like labels*

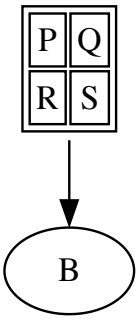
Ingat atribut `label` yang bisa dipasang ketika membuat suatu *node*? Sebenarnya, kita bisa memanfaatkan atribut tersebut untuk membuat bentuk *node* sesuka hati kita, lho! Terutama, kita bisa membuat *node* dengan bentuk seperti tabel.

Penulisan `label` seperti tabel ini mirip seperti struktur bahasa HTML, sehingga disebut ***HTML-like labels***.

Perhatikan *syntax* (penulisan) berikut.

```
graf13 = gv.Digraph()
graf13.node("A", shape="none", label="""<
<TABLE>
    <TR>
        <TD>P</TD>
        <TD>Q</TD>
    </TR>
    <TR>
        <TD>R</TD>
        <TD>S</TD>
    </TR>
</TABLE>
>""")

graf13.node("B") # node biasa
graf13.edge("A", "B")
```

```
display(graf13)
```



Perhatikan,

1. Ketika membuat *node* yang ingin berbentuk tabel, ditambahkan atribut `shape="none"` (bukan `None`) di samping menulis `label` nya.
2. `label` berupa *long string*, sehingga diawali dan diakhiri dengan tiga tanda kutip.
3. Karakter pertama dari *long string* tersebut haruslah `<` dan karakter terakhir haruslah `>`
4. Kemudian, penulisan tabel diawali dengan penulisan `<TABLE>`, kemudian `<TR>` (*table row*) untuk tiap baris, lalu `<TD>` (*table data*) untuk tiap sel. Masing-masing selalu ditutup dengan `</TD>`, `</TR>`, dan `</TABLE>`, bagaikan keberadaan `endif`, `endfor`, `endwhile` dan sebagainya di *pseudocode*.

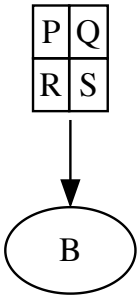
Agar lebih bagus, di bagian `<TABLE>` kita bisa menambahkan:

```
BORDER="0" CELLBORDER="1" CELLSPACING="0"
```

Seperti berikut:

```
graf14 = gv.Digraph()
graf14.node("A", shape="none", label="""<
<TABLE BORDER="0" CELLBORDER="1" CELLSPACING="0">
    <TR>
        <TD>P</TD>
        <TD>Q</TD>
    </TR>
    <TR>
        <TD>R</TD>
        <TD>S</TD>
    </TR>
</TABLE>
>""")

graf14.node("B")
graf14.edge("A", "B")
```

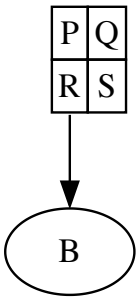
```
display(graf14)
```



Bagaimana kalau misalnya kita ingin panahnya seperti "berasal" dari sel tertentu? Caranya, kita bisa membuat yang namanya `port`, misalnya di sel R, kemudian *edge* yang dibuat akan kita sambung dari port tersebut, seperti berikut:

```
graf15 = gv.Digraph()
graf15.node("A", shape="none", label="""<
<TABLE BORDER="0" CELLBORDER="1" CELLSPACING="0">
    <TR>
        <TD>P</TD>
        <TD>Q</TD>
    </TR>
    <TR>
        <TD PORT="port1">R</TD>
        <TD>S</TD>
    </TR>
</TABLE>
>""")

graf15.node("B")
graf15.edge("A:port1", "B")
```

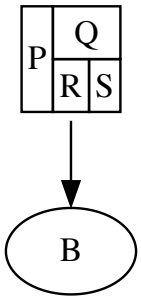
```
display(graf15)
```



Kalau di Microsoft Excel atau Google Sheets, kita bisa melakukan *merge* beberapa sel, entah secara horizontal atau vertikal atau bahkan dua-duanya. Ketika menyusun *HTML-like labels*, kita bisa menggunakan `COLSPAN` (merentang beberapa kolom) dan `ROWSPAN` (merentang beberapa baris) untuk membuat efek seperti di-*merge*.

```
graf16 = gv.Digraph()
graf16.node("A", shape="none", label="""<
<TABLE BORDER="0" CELLBORDER="1" CELLSPACING="0">
    <TR>
        <TD ROWSPAN="2">P</TD>
        <TD COLSPAN="2">Q</TD>
    </TR>
    <TR>
        <TD>R</TD>
        <TD>S</TD>
    </TR>
</TABLE>
>""")

graf16.node("B")
graf16.edge("A", "B")
```

```
        display(graf16)
```



(Singly) Linked List

Singly-linked list (seringkali disebut *linked list* saja) adalah semacam "rantai" dari *node*, di mana tiap *node* berisi 2 nilai, yaitu `data` dan `next` (yaitu *pointer* ke *node* lain). *Node* yang paling pertama itu ditunjuk oleh suatu *pointer* bernama `head`, yang menjadi awal dari *linked list*.

(Terkadang, *pointer* `next` ditulis `LINK`. Artinya dan kegunaannya sama.)

Pertama-tama, kita buat struktur *node* terlebih dahulu menggunakan `class`. (Apabila *pointer* `next` tidak menunjuk ke apapun, biasanya ditulis `NULL` atau di sini `None`.)

Biasanya, di kuliah, disebutnya `class Node` atau `Node` saja. Namun, berhubung modul ini akan membahas *doubly-linked list* dengan struktur yang agak berbeda, maka *node* untuk *singly-linked list* akan kita sebut `SLNode` (*singly-linked node*) agar berbeda.

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

Kita bisa bermain-main dengan *node* ini sebagaimana yang dibahas di kuliah. Misalnya, kita buat *node* baru yang menyimpan data 15:

```
        p = SLNode(15)
```

Saat ini, *node* tersebut ditunjuk oleh *pointer* yang di sini kita sebut `p`. Secara tidak langsung, kita telah membuat *linked list* dengan `head` nya adalah `p`.

Kita bisa mengakses data yang disimpan di `data` dan juga alamat yang tersimpan di `next`:

```
        print(p.data)
```

```
15
```

```
        print(p.next)
```

```
None
```

Saat ini, *node* yang ditunjuk oleh `p` itu belum menunjuk ke manapun, sehingga `p.next` masih bernilai `None`.

Kita bisa melihat alamat dari *node* itu sendiri menggunakan `id`:

```
        print(id(p))
```

```
4404463888
```

Alamat ini akan selalu berbeda tiap kali kita membuat *node* baru, dan di antara dua komputer kemungkinan besar juga berbeda. Memang wajar apabila alamat yang kalian dapatkan itu berbeda dengan yang tertera di modul.

Namun, alamat biasanya ditampilkan dalam bentuk heksadesimal (*base-16*), sedangkan yang kita dapatkan dengan `id` masih berupa bilangan bulat desimal (*base-10*). Kita bisa menggunakan `hex` untuk mengubah *base-10* menjadi *base-16*:

```
        print(hex(id(p)))
```

```
0x10686c910
```

Awalan `0x` itu hanya penanda bahwa bilangannya berupa heksadesimal.

Selanjutnya, kita bisa membuat *node* baru di `p.next`, yaitu yang ditunjuk oleh `p`, sebagai berikut:

```
        p.next = SLNode(28)
```

Sehingga, data 28 itu bisa diakses dari `p` seperti berikut:

```
        print(p.next.data)
```

```
28
```

Sedangkan, setelah *node* berisi 15 dan *node* berisi 28, belum ada *node* lagi, sehingga:

```
        print(p.next.next)
```

```
None
```

Mari kita buat *node* baru lagi setelah *node* berisi 28:

```
        p.next.next = SLNode(-3)
```

Sehingga, kita bisa mengakses data masing-masing *node* dari `p`:

```
        print(p.data)
        print(p.next.data)
        print(p.next.next.data)
```

```
15
28
-3
```

Kita bisa juga membuat *pointer* baru yang menunjuk ke *node* yang sudah ada. Misalnya, kita bisa membuat *pointer* bernama `q` yang menunjuk ke *node* yang berisi 28, seperti berikut:

```
q = p.next
```

Sehingga, `p.next.next` bisa diakses dengan `q.next`:

```
print(p.next.next.data)
print(q.next.data)
```

```
-3
-3
```

Bahkan, kita bisa mengubah data -3 menjadi yang lain melalui `q`, dan itu akan berubah juga jika diakses melalui `p`:

```
q.next.data = -63
print(q.next.data)
print(p.next.next.data)
```

```
-63
-63
```

Kok bisa? Karena, sesuai yang sudah kita tetapkan, `q` menunjuk ke *node* yang sama dengan `p.next`. Kita bisa periksa alamatnya:

```
print(hex(id(q)))
print(hex(id(p.next)))
```

```
0x10686d780
0x10686d780
```

Sehingga alamat dari *node* yang ditunjuk oleh `q.next` akan sama dengan yang ditunjuk oleh `p.next.next`:

```
print(hex(id(q.next)))
print(hex(id(p.next.next)))
```

```
0x10686d000
0x10686d000
```

Sejauh ini, kita sudah bermain dengan *node* dan membuat *linked list* secara manual. Sebenarnya, kita juga bisa membuat suatu `class` untuk suatu *linked list* secara keseluruhan. Di dalam `class` itu, kita bisa membuat atribut (variabel) yang menyimpan `head`, serta berbagai *method* (fungsi) untuk algoritma-algoritma operasi dasar yang kita pelajari di kuliah, seperti insert *node* di awal/akhir dan delete *node* di awal/akhir. Dengan begitu, kita bisa menggunakan *linked list* dengan lebih nyaman.

Kita akan menyebutnya `class SLList` (*singly-linked list*).

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

    def is_empty(self):
        if self.head == None:
            return True
        else:
```

```
        return False

    # Traversal, hanya untuk menghitung banyaknya node di linked list
    def get_size(self):
        count = 0
        current = self.head
        while current != None:
            count += 1
            current = current.next
        return count

    # Traversal, print masing-masing data node dari awal sampai akhir
    def print_all(self):
        print("head -> ", end="")
        temp = self.head
        while temp != None:
            print(temp.data, end = " -> ")
            temp = temp.next
        print("None")

    # Traversal, semacam linear search, cari letak node dengan data tertentu
    def get_pos(self, x):
        pos = -1
        current = self.head
        while current != None:
            pos += 1
            if current.data == x:
                return pos
            current = current.next
        return -1

    def ins_front(self, newdata):
        newnode = SLNode(newdata)
        newnode.next = self.head
        self.head = newnode

    def ins_end(self, newdata):
        newnode = SLNode(newdata)
        if self.is_empty():
            self.head = newnode
        else:
            temp = self.head
            while temp.next != None:
                temp = temp.next

            # sekarang temp sudah di node terakhir
            temp.next = newnode

    def ins_pos(self, newdata, pos):
        if pos == 0:
            self.ins_front(newdata)
        else:
            current_pos = 0
            current = self.head
            while (current != None) and (current_pos != pos-1):
                current = current.next
                current_pos += 1
```

```
            # Keluar loop, bisa karena current == None atau current_pos == pos
            # Kalau karena current_pos == pos-1, bisa insert
            if (current_pos == pos-1):
                newnode = SLNode(newdata)
                temp = current.next
                current.next = newnode
                newnode.next = temp
            # Tapi kalau karena current == None,
            # berarti posisi yang diminta melampaui panjang linked list
            else:
                print("Error: posisi melebihi panjang linked list")

    def del_front(self):
        if self.is_empty():
            print("Error: linked list sudah kosong")
        else:
            temp = self.head.next
            del self.head
            self.head = temp

    def del_end(self):
        if self.is_empty():
            print("Error: linked list sudah kosong")
        else:
            temp = self.head
            while temp.next.next != None:
                temp = temp.next

            # sekarang temp ada di node sebelum terakhir
            del temp.next
            temp.next = None

    # Mirip ins_pos, hanya berbeda di bagian current_pos == pos-1
    def del_pos(self, pos):
        if pos == 0:
            self.del_front()
        else:
            current_pos = 0
            current = self.head
            while (current != None) and (current_pos != pos-1):
                current = current.next
                current_pos += 1
            # Keluar loop, bisa karena current == None atau current_pos == pos
            # Kalau karena current_pos == pos-1, maka bisa dihapus selama
            # current.next yang mau dihapus itu memang ada
            if (current_pos == pos-1) and (current.next != None):
                temp = current.next.next
                del current.next
                current.next = temp
            # Tapi kalau karena current == None, atau current.next tidak ada,
            # berarti posisi yang diminta melampaui panjang linked list
            else:
                print("Error: posisi melebihi panjang linked list")

    # Menghapus semua node di linked list
    def del_all(self):
        while (not self.is_empty()):
```

```
            self.del_front()

    # Method untuk memperoleh digraph yang menggambarkan linked list nya :D
    def get_digraph(self):
        # Buat digraph baru yang sifatnya dari kiri ke kanan
        new_digraph = gv.Digraph(graph_attr={"rankdir": "LR"})

        # Pointer untuk menunjuk ke tiap node, mulai dari node pertama
        # (akan dilakukan traversal)
        current = self.head

        # Untuk menghitung node ke-sekian untuk nama node di Graphviz,
        # sehingga head menunjuk ke node0, lalu node0 menunjuk ke node1, dst
        counter = 0

        # Memperoleh alamat yang sedang disimpan di head
        # - asumsi awal: tidak ada alamat (None)
        next_id = None
        next_name = "node0" # ini nanti untuk nama node berikutnya di Graphviz
        # - kalau ternyata ada alamat...
        if current != None:
            # maka simpan alamat tersebut
            next_id = hex(id(current))
            # kita buat lebih spesifik untuk node berikutnya, tunjuk ke port i
            next_name = "node0:id"

        # Label (tabel) untuk pointer head
        # - pembuka tabel
        str_label = "<"
        str_label += "<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLSPACING=\"0\">"
        # - baris head
        str_label += "<TR><TD>head</TD></TR>"
        # - baris alamat (sekalian membuat port namanya "contents")
        str_label += "<TR><TD PORT=\"contents\">" + str(next_id) + "</TD></TR>
        # - penutup tabel
        str_label += "</TABLE>"
        str_label += ">"

        # Membuat node head, membuat edge dari head ke node berikutnya
        new_digraph.node("head", shape="none", label=str_label)
        new_digraph.edge("head:contents", next_name)
        # dari port "contents" ke node berikutnya, yang namanya next_name

        # Selama node yang ditunjuk bukan None, buatlah node nya di Graphviz,
        # lalu lanjut ke node selanjutnya (ini traversal)
        while current != None:
            # Alamat yang tersimpan pada current.next
            # - asumsi awal: tidak ada alamat; current adalah node terakhir
            next_id = None
            # - kalau ternyata ada alamat...
            if current.next != None:
                # maka simpan alamat tersebut
                next_id = hex(id(current.next))

            # Persiapan label (tabel) untuk node
            # - pembuka tabel
            str_label = "<"
```

```
                str_label += "<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLSPACING=\"0
                # - baris tulisan "data", "next"
                str_label += "<TR><TD>data</TD><TD>next</TD></TR>"
                # - baris untuk isi data dan isi next
                str_label += "<TR>"
                str_label += "<TD>" + str(current.data) + "</TD>"
                str_label += "<TD PORT=\"next\">" + str(next_id) + "</TD>"
                str_label += "</TR>"
                # - baris tulisan "alamat node", merentang dua kolom
                str_label += "<TR><TD COLSPAN=\"2\">alamat node</TD></TR>"
                # - baris untuk isi alamat node, merentang dua kolom
                str_label += "<TR>"
                str_label += "<TD PORT=\"id\" COLSPAN=\"2\">"
                str_label += str(hex(id(current)))
                str_label += "</TD>"
                str_label += "</TR>"
                # - penutup tabel
                str_label += "</TABLE>"
                str_label += ">"

                # Membuat node baru di Graphviz dengan label (tabel) tersebut
                new_digraph.node("node" + str(counter), shape="none", label = str_

                # Menentukan nama dua port yang bakal disambung dengan edge,
                # yaitu (node saat ini):next disambung ke node(berikutnya):id
                # yaitu bagian "next" disambung ke bagian alamat di node berikutny
                nama_node_next = "node" + str(counter) + ":next"
                if current.next != None:
                    nama_alamat_node_berikutnya = "node" + str(counter+1) + ":id"
                # atau ke node(berikutnya) saja tanpa id kalau itu ternyata None,
                # karena None tidak akan memiliki port id
                else:
                    nama_alamat_node_berikutnya = "node" + str(counter+1)

                # Menyambung keduanya
                new_digraph.edge(nama_node_next, nama_alamat_node_berikutnya)

                # Lanjut ke node selanjutnya
                current = current.next
                counter += 1
            # Kalau sudah keluar loop, artinya current menunjuk ke None
            # Berarti tinggal membuat "node" terakhir berisi tulisan None
            # (karena sambungannya sudah dibuat di dalam loop, tinggal node nya)
            new_digraph.node("node" + str(counter), shape="none", label="None")

            # Digraph sudah jadi
            return new_digraph
```

```
test = SLList()
test.ins_front(5)
test.ins_front(15)
test.ins_front(25)
test.ins_front(35)
```

```
test.print_all()
```

```
head -> 35 -> 25 -> 15 -> 5 -> None
```

```
        print(test.get_pos(15))
```

```
2
```

```
        print(test.get_pos(39))
```

```
-1
```

```
        test.ins_end(100)
```

```
        test.print_all()
```

```
head -> 35 -> 25 -> 15 -> 5 -> 100 -> None
```

```
        test.del_front()
        test.del_front()
```

```
        test.print_all()
```

```
head -> 15 -> 5 -> 100 -> None
```

```
        test.del_pos(3)
```

```
Error: posisi melebihi panjang linked list
```

```
        test.del_pos(2)
```

```
        test.print_all()
```

```
head -> 15 -> 5 -> None
```

```
        test.ins_pos(-42, 7)
```

```
Error: posisi melebihi panjang linked list
```

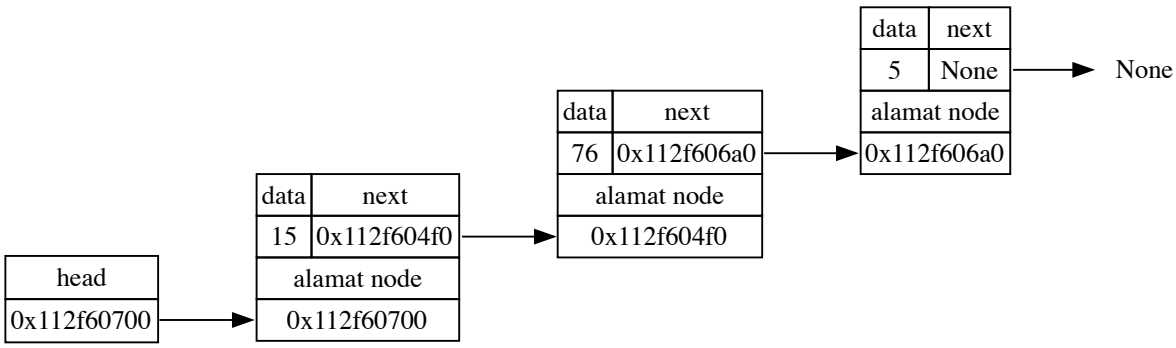
```
        test.ins_pos(76, 1)
```

```
        test.print_all()
```

```
head -> 15 -> 76 -> 5 -> None
```

```
        gambar = test.get_digraph()
```

```
        display(gambar)
```



Doubly Linked List

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

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

    # Masih sama persis dengan singly linked list
    def is_empty(self):
        if self.head == None:
            return True
        else:
            return False

    # Traversal, hanya untuk menghitung banyaknya node di linked list
    # Masih sama persis dengan singly linked list
    def get_size(self):
        count = 0
        current = self.head
        while current != None:
            count += 1
            current = current.next
        return count

    # Traversal, print masing-masing data node dari awal sampai akhir
    def print_all(self):
        print("head -> ", end="")
        temp = self.head
        while (temp != None) and (temp.next != None):
            print(temp.data, end = " <-> ")
            temp = temp.next
        # Khusus node terakhir:
        if (temp != None) and (temp.next == None):
            print(temp.data, end = " <- ")
        print("tail")

    def ins_front(self, newdata):
```

```
        newnode = DLNode(newdata)
        newnode.next = self.head
        if self.head != None:
            self.head.prev = newnode
        self.head = newnode
        if self.tail == None: # jika tadinya doubly linked list kosong,
            # maka newnode menjadi node pertama, ditunjuk oleh head dan tail
            self.tail = newnode

    # Berbeda dengan singly linked list, tinggal insert di tail;
    # tidak perlu traversal
    def ins_end(self, newdata):
        newnode = DLNode(newdata)
        newnode.prev = self.tail
        if self.tail != None:
            self.tail.next = newnode
        self.tail = newnode
        if self.head == None: # jika tadinya doubly linked list kosong,
            # maka newnode menjadi node pertama, ditunjuk oleh head dan tail
            self.head = newnode

    def ins_pos(self, newdata, pos):
        if pos == 0:
            self.ins_front(newdata)
            return
        n = self.get_size()
        if pos == n:
            self.ins_end(newdata)
        elif pos > n:
            print("Error: posisi melebihi panjang linked list")
        else:
            current_pos = 0
            current = self.head
            while (current_pos != pos-1):
                current = current.next
                current_pos += 1
            # Keluar loop berarti current_pos == pos-1
            newnode = DLNode(newdata)
            newnode.prev = current
            newnode.next = current.next
            current.next = newnode
            # Sudah pasti newnode.next != None,
            # karena kasus pos == n sudah ditangani
            newnode.next.prev = newnode

    def del_front(self):
        if self.is_empty():
            print("Error: linked list sudah kosong")
        else:
            temp = self.head.next
            del self.head
            self.head = temp
            if temp != None:
                temp.prev = None
            else: # jika temp == None, maka self.head == None,
                # berarti sekarang doubly linkd list kosong,
                # sehingga tail juga menunjuk ke None
```

```
            self.tail = None

    def del_end(self):
        if self.is_empty():
            print("Error: linked list sudah kosong")
        else:
            temp = self.tail.prev
            del self.tail
            self.tail = temp
            if temp != None:
                temp.next = None
            else: # jika temp == None, maka self.tail == None,
                # berarti sekarang doubly linkd list kosong,
                # sehingga head juga menunjuk ke None
                self.head = None

    def del_pos(self, pos):
        if pos == 0:
            self.del_front()
            return
        n = self.get_size()
        if pos == n-1:
            self.del_end()
        elif pos > n-1:
            print("Error: posisi melebihi panjang linked list")
        else:
            current_pos = 0
            current = self.head
            while (current_pos != pos-1):
                current = current.next
                current_pos += 1
            temp = current.next.next
            del current.next
            current.next = temp
            # Sudah pasti temp != None,
            # karena kasus pos == (n-1) sudah ditangani
            temp.prev = current

    # Method untuk memperoleh digraph yang menggambarkan linked list nya :D
    def get_digraph(self):
        # Buat digraph baru yang sifatnya dari kiri ke kanan
        new_digraph = gv.Digraph(graph_attr={"rankdir": "LR"})

        # Pointer untuk menunjuk ke tiap node, mulai dari node pertama
        # (akan dilakukan traversal)
        current = self.head

        # Untuk menghitung node ke-sekian untuk nama node di Graphviz,
        # sehingga head menunjuk ke node0, lalu node0 menunjuk ke node1, dst
        counter = 0

        # Memperoleh alamat yang sedang disimpan di head
        # - asumsi awal: tidak ada alamat (None)
        next_id = None
        next_name = "node0" # ini nanti untuk nama node berikutnya di Graphviz
        # - kalau ternyata ada alamat...
        if current != None:
```

```
            # maka simpan alamat tersebut
            next_id = hex(id(current))
            # kita buat lebih spesifik untuk node berikutnya, tunjuk ke port i
            next_name = "node0:id"

        # Label (tabel) untuk pointer head
        # - pembuka tabel
        str_label = "<"
        str_label += "<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLSPACING=\"0\">"
        # - baris head
        str_label += "<TR><TD>head</TD></TR>"
        # - baris alamat (sekalian membuat port namanya "contents")
        str_label += "<TR><TD PORT=\"contents\">" + str(next_id) + "</TD></TR>
        # - penutup tabel
        str_label += "</TABLE>"
        str_label += ">"

        # Membuat node head, membuat edge dari head ke node berikutnya
        new_digraph.node("head", shape="none", label=str_label)
        new_digraph.edge("head:contents", next_name)
        # dari port "contents" ke node berikutnya, yang namanya next_name

        # Selama node yang ditunjuk bukan None, buatlah node nya di Graphviz,
        # lalu lanjut ke node selanjutnya (ini traversal)
        while current != None:
            # Alamat yang tersimpan pada current.next
            # - asumsi awal: tidak ada alamat; current adalah node terakhir
            next_id = None
            # - kalau ternyata ada alamat...
            if current.next != None:
                # maka simpan alamat tersebut
                next_id = hex(id(current.next))

            # serupa untuk prev
            prev_id = None
            if current.prev != None:
                prev_id = hex(id(current.prev))

            # Persiapan label (tabel) untuk node
            # - pembuka tabel
            str_label = "<"
            str_label += "<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLSPACING=\"0
            # - baris tulisan "prev", "data", "next"
            str_label += "<TR><TD>prev</TD><TD>data</TD><TD>next</TD></TR>"
            # - baris untuk isi prev, isi data, dan isi next
            str_label += "<TR>"
            str_label += "<TD PORT=\"prev\">" + str(prev_id) + "</TD>"
            str_label += "<TD>" + str(current.data) + "</TD>"
            str_label += "<TD PORT=\"next\">" + str(next_id) + "</TD>"
            str_label += "</TR>"
            # - baris tulisan "alamat node", merentang dua kolom
            str_label += "<TR><TD COLSPAN=\"3\">alamat node</TD></TR>"
            # - baris untuk isi alamat node, merentang dua kolom
            str_label += "<TR>"
            str_label += "<TD PORT=\"id\" COLSPAN=\"3\">"
            str_label += str(hex(id(current)))
            str_label += "</TD>"
```

```
            str_label += "</TR>"
            # - penutup tabel
            str_label += "</TABLE>"
            str_label += ">"

            # Membuat node baru di Graphviz dengan label (tabel) tersebut
            new_digraph.node("node" + str(counter), shape="none", label = str_

            # Menentukan nama dua port yang bakal disambung dengan edge,
            # yaitu (node saat ini):next disambung ke node(berikutnya):id
            # yaitu bagian "next" disambung ke bagian alamat di node berikutny
            nama_node_next = "node" + str(counter) + ":next"

            # tambahan untuk doubly linked list
            nama_node_prev = "node" + str(counter) + ":prev"

            if current.next != None:
                nama_alamat_node_berikutnya = "node" + str(counter+1) + ":id"
            # atau ke node(berikutnya) saja tanpa id kalau itu ternyata None,
            # karena None tidak akan memiliki port id
            else:
                nama_alamat_node_berikutnya = "node" + str(counter+1)

            # Menyambung keduanya
            new_digraph.edge(nama_node_next, nama_alamat_node_berikutnya)

            # tambahan untuk doubly linked list
            if current.prev != None:
                nama_alamat_node_sebelumnya = "node" + str(counter-1) + ":id"
            else:
                nama_alamat_node_sebelumnya = "node" + str(counter-1)
            if current == self.head:
                new_digraph.node("node-1", shape="none", label="None")
            new_digraph.edge(nama_node_prev, nama_alamat_node_sebelumnya)

            # Lanjut ke node selanjutnya
            current = current.next
            counter += 1
        # Kalau sudah keluar loop, artinya current menunjuk ke None
        # Berarti tinggal membuat "node" terakhir berisi tulisan None
        # (karena sambungannya sudah dibuat di dalam loop, tinggal node nya)
        new_digraph.node("node" + str(counter), shape="none", label="None")

        # Tambah pointer tail
        # - asumsi awal: tidak ada alamat (None)
        tail_id = None
        tail_name = "node" + str(counter-1) # ini nanti untuk nama node tail
        # - kalau ternyata ada alamat...
        if self.tail != None:
            # maka simpan alamat tersebut
            tail_id = hex(id(self.tail))
            # kita buat lebih spesifik untuk node berikutnya, tunjuk ke port i
            tail_name += ":id"

        # Label (tabel) untuk pointer tail
        # - pembuka tabel
        str_label = "<"
```

```
            str_label += "<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLSPACING=\"0\">"
            # - baris head
            str_label += "<TR><TD>tail</TD></TR>"
            # - baris alamat (sekalian membuat port namanya "contents")
            str_label += "<TR><TD PORT=\"contents\">" + str(tail_id) + "</TD></TR>
            # - penutup tabel
            str_label += "</TABLE>"
            str_label += ">"

            # Membuat node tail, membuat edge dari tail ke node nya
            new_digraph.node("tail", shape="none", label=str_label)
            new_digraph.edge("tail:contents", tail_name)
            # dari port "contents" ke node yang ditunjuk tail, namanya tail_name

            # Digraph sudah jadi
            return new_digraph
```

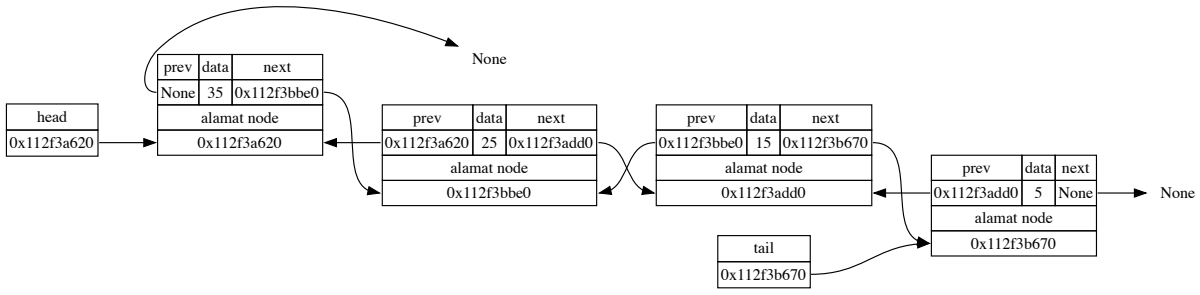
```
testDL = DLList()
testDL.ins_front(5)
testDL.ins_front(15)
testDL.ins_front(25)
testDL.ins_front(35)
```

```
testDL.print_all()
```

```
head -> 35 <-> 25 <-> 15 <-> 5 <- tail
```

```
gambarDL = testDL.get_digraph()
```

```
display(gambarDL)
```



Modul 4 Struktur Data: *Stack* dan notasi *prefix*, *infix*, *postfix*

Di praktikum kali ini tentang *stack*, kita akan membahas implementasi *stack* (baik dengan *array* maupun dengan *linked list*) serta contoh penggunaannya. Selain itu, kita akan membahas tentang penggunaan *stack* ketika berurusan dengan notasi *prefix*, *infix*, dan *postfix*.

```
import numpy as np
import graphviz as gv
```

Implementasi dan contoh penggunaan *stack*

Implementasi *stack* dengan *array*

```
class ArrayStack:
    def __init__(self, dtype, max):
        self.dtype = dtype
        self.max = max
        self.array = np.empty(max, dtype=dtype)
        self.top = -1

    def get_size(self):
        return self.top + 1

    def get_capacity(self):
        return self.max

    def get_dtype(self):
        return self.dtype

    def is_empty(self):
        if self.get_size() > 0:
            return False
        else:
            return True

    def is_full(self):
        if self.get_size() >= self.get_capacity():
            # if top+1 >= max
            # atau sama saja, if top >= max-1
            return True
        else:
            return False

    def push(self, newdata):
```

```
        if self.is_full():
            print("Error push: stack sudah penuh.")
        else:
            self.top += 1
            self.array[self.top] = newdata

    def peek(self):
        if self.is_empty():
            print("Error peek: stack sedang kosong.")
            return None
        else:
            return self.array[self.top]

    def pop(self):
        if self.is_empty():
            print("Error pop: stack sudah kosong sebelumnya.")
            return None
        else:
            output = self.array[self.top]
            self.top -= 1
            return output

    def print_stack(self):
        i = self.top
        while i >= 0:
            print(self.array[i])
            i -= 1

    # print array
    def print_storage(self):
        print(self.array)

    def get_digraph_stack(self):
        new_digraph = gv.Digraph()
        # gambar akan terdiri dari satu tabel saja, satu kolom,
        # dan tiap baris adalah tiap elemen di stack

        tabel_besar = "<"
        # pembuka tabel
        tabel_besar += "<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLSPACING=\"0\"
        # menambahkan tiap elemen sebagai baris tersendiri
        i = self.top
        if i < 0:
            tabel_besar += "<TR><TD>"
            tabel_besar += "(Stack sedang kosong; tidak ada data sama sekali.)
            tabel_besar += "</TD></TR>"
        while i >= 0:
            tabel_besar += "<TR><TD>"
            tabel_besar += str(self.array[i])
            tabel_besar += "</TD></TR>"
            i -= 1
        # penutup tabel
        tabel_besar += "</TABLE>"
        tabel_besar += ">"
        new_digraph.node("ArrayStack", shape="none", label=tabel_besar)
        return new_digraph
```

```
            def get_digraph_storage(self):
                # menggambar array
                new_digraph = gv.Digraph()

                # pembuka tabel
                tabel_besar = "<"
                tabel_besar += "<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLSPACING=\"0\"
                # tabel hanya terdiri dari satu baris
                tabel_besar += "<TR>"
                # satu elemen per kolom
                for i in range(self.get_capacity()):
                    tabel_besar += "<TD>"
                    tabel_besar += str(self.array[i])
                    tabel_besar += "</TD>"
                # penutup baris
                tabel_besar += "</TR>"
                # penutup tabel
                tabel_besar += "</TABLE>"
                tabel_besar += ">"
                new_digraph.node("array", shape="none", label=tabel_besar)
                return new_digraph
```

```
arraystack = ArrayStack(int, 5)
arraystack.push(5)
arraystack.push(80)
arraystack.push(100)
```

```
arraystack.print_stack()
```

```
100
80
5
```

```
print(arraystack.get_capacity())
```

```
5
```

```
arraystack.print_storage()
```

```
[                 5                  80                 100
 4622241330054037504 4625478292286210048]
```

```
print(arraystack.peek())
```

```
100
```

```
arraystack.print_stack()
```

```
100
80
5
```

```
        nilai = arraystack.pop()
        print(nilai)
```

```
100
```

```
        arraystack.print_stack()
```

```
80
5
```

```
        arraystack.print_storage()
```

```
[                  5                  80                 100
 4622241330054037504 4625478292286210048]
```

```
        arraystack.push(-10)
        arraystack.push(57)
```

```
        arraystack.print_stack()
```

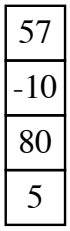
```
57
-10
80
5
```

```
        arraystack.print_storage()
```

```
[                  5                  80                 -10
                  57 4625478292286210048]
```

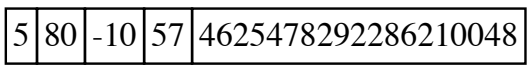
```
        graf1 = arraystack.get_digraph_stack()
```

```
        display(graf1)
```




```
        graf2 = arraystack.get_digraph_storage()
```

```
        display(graf2)
```



```
        arraystack.push(90)
```

```
        arraystack.push(46)
```

```
Error push: stack sudah penuh.
```

```
        arraystack.print_storage()
```

```
[  5  80 -10  57  90]
```

```
        print(arraystack.pop())
        print(arraystack.pop())
        print(arraystack.pop())
        print(arraystack.pop())
        print(arraystack.pop())
```

```
90
57
-10
80
5
```

```
        print(arraystack.pop())
```

```
Error pop: stack sudah kosong sebelumnya.
None
```

```
        print(arraystack.get_size())
```

```
0
```

```
        arraystack.print_stack()
```

```
        arraystack.print_storage()
```

```
[  5  80 -10  57  90]
```

```
        display(arraystack.get_digraph_stack())
```

(Stack sedang kosong; tidak ada data sama sekali.)

Implementasi *stack* dengan *singly-inked list*

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

```
class SLStack:
    def __init__(self):
        # "head" ganti nama jadi top
        self.top = None

    def is_empty(self):
        if self.top == None:
            return True
        else:
            return False

    def push(self, newdata):
        newnode = SLNode(newdata)
        newnode.next = self.top
        self.top = newnode

    def peek(self):
        if self.is_empty():
            print("Error peek: stack sedang kosong.")
        else:
            return self.top.data

    def pop(self):
        if self.is_empty():
            print("Error pop: stack sudah kosong sebelumnya.")
        else:
            output = self.top.data
            temp = self.top
            self.top = self.top.next
            del temp
            return output

    def get_size(self):
        temp = self.top
        size = 0
        while temp != None:
            size += 1
            temp = temp.next
        return size

    def print_stack(self):
        temp = self.top
        while temp != None:
            print(temp.data)
            temp = temp.next

    # print linked list
    def print_storage(self):
        print("top -> ", end="")
        temp = self.top
        while temp != None:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")

    def get_digraph_stack(self):
```

```
        new_digraph = gv.Digraph()
        # gambar akan terdiri dari satu tabel saja, satu kolom,
        # dan tiap baris adalah tiap elemen di stack
        tabel_besar = ""
        tabel_besar += "<"
        tabel_besar += "<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLSPACING=\"0\"
        temp = self.top
        if temp == None:
            tabel_besar += "<TR><TD>"
            tabel_besar += "(Stack sedang kosong; tidak ada data sama sekali.)
            tabel_besar += "</TD></TR>"
        while temp != None:
            tabel_besar += "<TR><TD>"
            tabel_besar += str(temp.data)
            tabel_besar += "</TD></TR>"
            temp = temp.next
        # penutup tabel
        tabel_besar += "</TABLE>"
        tabel_besar += ">"
        new_digraph.node("SLStack", shape="none", label=tabel_besar)
        return new_digraph

    # copas dari modul linked list, tapi head ganti jadi top
    def get_digraph_storage(self):
        # Buat digraph baru yang sifatnya dari kiri ke kanan
        new_digraph = gv.Digraph(graph_attr={"rankdir": "LR"})

        # Pointer untuk menunjuk ke tiap node, mulai dari node pertama
        # (akan dilakukan traversal)
        current = self.top

        # Untuk menghitung node ke-sekian untuk nama node di Graphviz,
        # sehingga top menunjuk ke node0, lalu node0 menunjuk ke node1, dst
        counter = 0

        # Memperoleh alamat yang sedang disimpan di top
        # - asumsi awal: tidak ada alamat (None)
        next_id = None
        next_name = "node0" # ini nanti untuk nama node berikutnya di Graphviz
        # - kalau ternyata ada alamat...
        if current != None:
            # maka simpan alamat tersebut
            next_id = hex(id(current))
            # kita buat lebih spesifik untuk node berikutnya, tunjuk ke port i
            next_name = "node0:id"

        # Label (tabel) untuk pointer top
        # - pembuka tabel
        str_label = "<"
        str_label += "<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLSPACING=\"0\">"
        # - baris top
        str_label += "<TR><TD>top</TD></TR>"
        # - baris alamat (sekalian membuat port namanya "contents")
        str_label += "<TR><TD PORT=\"contents\">" + str(next_id) + "</TD></TR>
        # - penutup tabel
        str_label += "</TABLE>"
        str_label += ">"
```

```
        # Membuat node top, membuat edge dari top ke node berikutnya
        new_digraph.node("top", shape="none", label=str_label)
        new_digraph.edge("top:contents", next_name)
        # dari port "contents" ke node berikutnya, yang namanya next_name

        # Selama node yang ditunjuk bukan None, buatlah node nya di Graphviz,
        # lalu lanjut ke node selanjutnya (ini traversal)
        while current != None:
            # Alamat yang tersimpan pada current.next
            # - asumsi awal: tidak ada alamat; current adalah node terakhir
            next_id = None
            # - kalau ternyata ada alamat...
            if current.next != None:
                # maka simpan alamat tersebut
                next_id = hex(id(current.next))

            # Persiapan label (tabel) untuk node
            # - pembuka tabel
            str_label = "<"
            str_label += "<TABLE BORDER=\"0\" CELLBORDER=\"1\" CELLSPACING=\"0
            # - baris tulisan "data", "next"
            str_label += "<TR><TD>data</TD><TD>next</TD></TR>"
            # - baris untuk isi data dan isi next
            str_label += "<TR>"
            str_label += "<TD>" + str(current.data) + "</TD>"
            str_label += "<TD PORT=\"next\">" + str(next_id) + "</TD>"
            str_label += "</TR>"
            # - baris tulisan "alamat node", merentang dua kolom
            str_label += "<TR><TD COLSPAN=\"2\">alamat node</TD></TR>"
            # - baris untuk isi alamat node, merentang dua kolom
            str_label += "<TR>"
            str_label += "<TD PORT=\"id\" COLSPAN=\"2\">"
            str_label += str(hex(id(current)))
            str_label += "</TD>"
            str_label += "</TR>"
            # - penutup tabel
            str_label += "</TABLE>"
            str_label += ">"

            # Membuat node baru di Graphviz dengan label (tabel) tersebut
            new_digraph.node("node" + str(counter), shape="none", label = str_

            # Menentukan nama dua port yang bakal disambung dengan edge,
            # yaitu (node saat ini):next disambung ke node(berikutnya):id
            # yaitu bagian "next" disambung ke bagian alamat di node berikutny
            nama_node_next = "node" + str(counter) + ":next"
            if current.next != None:
                nama_alamat_node_berikutnya = "node" + str(counter+1) + ":id"
            # atau ke node(berikutnya) saja tanpa id kalau itu ternyata None,
            # karena None tidak akan memiliki port id
            else:
                nama_alamat_node_berikutnya = "node" + str(counter+1)

            # Menyambung keduanya
            new_digraph.edge(nama_node_next, nama_alamat_node_berikutnya)
```

```
                    # Lanjut ke node selanjutnya
                    current = current.next
                    counter += 1
                # Kalau sudah keluar loop, artinya current menunjuk ke None
                # Berarti tinggal membuat "node" terakhir berisi tulisan None
                # (karena sambungannya sudah dibuat di dalam loop, tinggal node nya)
                new_digraph.node("node" + str(counter), shape="none", label="None")

                # Digraph sudah jadi
                return new_digraph
```

```
slstack = SLStack()
slstack.print_storage()
```

```
top -> None
```

```
slstack.push("abc")
slstack.push("fg")
slstack.push("ijk")
slstack.push("pqrs")
slstack.push("xyz")
```

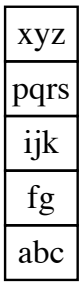
```
slstack.print_stack()
```

```
xyz
pqrs
ijk
fg
abc
```

```
slstack.print_storage()
```

```
top -> xyz -> pqrs -> ijk -> fg -> abc -> None
```

```
display(slstack.get_digraph_stack())
```




```
print(slstack.pop())
print(slstack.pop())
print(slstack.pop())
```

```
xyz
pqrs
```

```
ijk
```

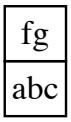
```
        slstack.print_stack()
```

```
fg
abc
```

```
        slstack.print_storage()
```

```
top -> fg -> abc -> None
```

```
        display(slstack.get_digraph_stack())
```



Contoh sederhana: *reverse* suatu *list/array*

```
        def reverse_array_arraystack(array_old):
            array = array_old.copy()

            # memeriksa tipe data dari elemen pertama
            tipe_data = type(array[0])
            # khusus array, bisa juga menggunakan array.dtype

            arraystack = ArrayStack(tipe_data, len(array))
            for i in range(len(array)):
                arraystack.push(array[i])
            for i in range(len(array)):
                array[i] = arraystack.pop()
            return array
```

```
        list1 = ["m", "a", "t", "e", "k"]
        list2 = reverse_array_arraystack(list1)
        print(list2)
```

```
['k', 'e', 't', 'a', 'm']
```

```
        def reverse_array_slstack(array_old):
            array = array_old.copy()
            slstack = SLStack()
            for i in range(len(array)):
                slstack.push(array[i])
            for i in range(len(array)):
                array[i] = slstack.pop()
            return array
```

```
        array1 = np.array(["m", "a", "t", "e", "k"])
        array2 = reverse_array_slstack(array1)
```

```
        print(array2)
```

```
['k' 'e' 't' 'a' 'm']
```

(TODO) Notasi *prefix*, *infix*, dan *postfix*

Notasi *prefix*, *infix*, dan *postfix* adalah tiga jenis notasi (cara penulisan) untuk menuliskan operasi aritmetika seperti penjumlahan, perkalian, dan sebagainya.

Misalnya, kita bisa menuliskan penjumlahan `3 + 5`, di mana dua angka, 3 dan 5, dioperasikan oleh suatu "operator" yaitu + (plus). Perhatikan bahwa operator berada di tengah, di antara kedua angka. Penulisan seperti ini disebut notasi *infix*, dan inilah penulisan yang biasa kita kenal.

Ada juga cara penulisan di mana operator ditempatkan sebelum kedua angka, disebut notasi *prefix*, seperti berikut: `+ 3 5`

Walaupun terlihat agak aneh, kita bisa saja mendefinisikan fungsi seperti *pseuducode* berikut:

```
        function add(x, y)
            return x+y
        endfunction
```

Kemudian penggunaannya adalah `add(3, 5)`, secara tidak langsung menggunakan notasi *prefix* :)

Selain *prefix* untuk di awal dan *infix* untuk di tengah, kita juga bisa menempatkan operator setelah kedua angka, disebut notasi *postfix*. Contohnya: `3 5 +`

Notasi *postfix* sebenarnya tidak terlalu asing, karena misalnya untuk menuliskan faktorial itu biasanya menggunakan tanda seru **setelah** angkanya, lagi-lagi secara tidak langsung menggunakan notasi *postfix*, seperti: `4!`

Salah satu keuntungan menggunakan notasi *prefix* maupun *postfix* adalah bisa menghilangkan kurung tanpa menyebabkan ambigu. Contohnya, dalam notasi *infix* kita bisa menuliskan `5 * (6 + 7)` agar penjumlahan dilakukan terlebih dahulu. Sedangkan, notasi *prefix* maupun *postfix* dijamin tidak membutuhkan kurung:

- *Prefix*: `* 5 + 6 7`
- *Postfix*: `6 7 + 5 *`

Stack bisa sangat membantu untuk mengubah antara notasi *prefix*, *infix*, dan *postfix*.

Tokenisasi

Sebelum membahas konversi antara notasi *prefix*, *infix*, dan *postfix*, kita perlu membahas sebentar mengenai "tokenisasi" (*tokenization*), yaitu proses "memecah" suatu string yang utuh menjadi "bagian-bagiannya".

Misalnya, kalau kita punya notasi *infix* dalam *string* `"3 + 5"`, kita bisa melakukan *tokenization* untuk memecahnya menjadi `["3", "+", "5"]`.

Cara mudah untuk melakukan tokenisasi, bisa dengan sekedar menganggap tiap "bagian" atau tiap "token" terpisahkan oleh spasi, sehingga bisa di-*split* begitu saja:

```
        def tokenize(string_utuh):
            hasil = string_utuh.split(" ") # string berisi satu spasi
            return hasil
```

```
        print(tokenize("3 + 5"))
```

```
['3', '+', '5']
```

Agar cara mudah ini berhasil (terutama untuk notasi *infix*), bahkan antara kurung buka/tutup juga harus diberi spasi, ya!

```
        print(tokenize("5 * ( 6 + 7 )"))
```

```
['5', '*', '(', '6', '+', '7', ')']
```

Precedence dan *associativity*

Sebelumnya, telah disebutkan bahwa salah satu keuntungan notasi *prefix* maupun *postfix* dibandingkan notasi *infix* adalah penulisan yang tidak ambigu tanpa diperlukannya kurung. Agar bisa mengubah notasi *infix* menjadi notasi *prefix* ataupun notasi *postfix*, tentunya kita harus bisa membaca notasi *infix* secara tidak ambigu. Artinya, kita harus kenal dengan aturan **urutan pengoperasian**.

Urusan urutan pengoperasian terbagi menjadi dua:

- *Precedence*, semacam tingkatan prioritas antara operasi yang berbeda, yang mana yang dilakukan duluan (apalagi kalau tidak ada tanda kurung)
- *Associativity*, urutan pengoperasian antara dua operasi yang *precedence* nya sama, apakah dari kiri ke kanan atau kanan ke kiri

Misalkan ada penulisan notasi *infix*: `9 + 8 * 7`

Tentunya perkalian dilakukan terlebih dahulu, barulah penjumlahan. Artinya, perkalian memiliki ***higher precedence*** (atau *precedence* yang lebih tinggi) daripada penjumlahan; bisa juga dikatakan, penjumlahan memiliki ***lower precedence*** (atau *precedence* yang lebih rendah) daripada perkalian.

Sedangkan, misal ada penulisan notasi *infix*: `8 / 4 * 2` dan `8 * 4 / 2`

Keduanya dilakukan dari kiri ke kanan. Artinya:

- Tidak ada prioritas yang lebih utama antara pembagian maupun perkalian, sehingga keduanya memiliki ***equal precedence*** (atau *precedence* yang sama).
- *Associativity* dari pembagian maupun perkalian bersifat *left-to-right*.

Precedence dan *associativity* dari beberapa operator bisa didata:

Precedence	Operator	*Associativity*
3	`^`	*right-to-left*
2	`* /`	*left-to-right*
1	`+ -`	*left-to-right*

Perhatikan:

- Perpangkatan bersifat *right-to-left* karena $a^{b^c} = a^{(b^c)}$.
- Pembagian maupun pengurangan bersifat *left-to-right* karena

$a/b/c = (a/b)/c$ dan

$a - b - c = (a - b) - c$.

- Kebetulan, perkalian maupun penjumlahan memiliki sifat asosiatif, yaitu

  $(a * b) * c = a * (b * c)$

  $(a + b) + c = a + (b + c)$

  sehingga perkalian maupun penjumlahan sebenarnya bersifat *left-to-right* maupun *right-to-left* sekaligus, yaitu

  $a * b * c = (a * b) * c = a * (b * c)$

  $a + b + c = (a + b) + c = a + (b + c)$

  Namun, untuk mempermudah klasifikasi, kita bisa mengkategorikan perkalian dan penjumlahan bersifat *left-to-right*.

(TODO) Urusan notasi *prefix*, *infix*, dan *postfix* dengan *stack*

Notasi *infix* menjadi *postfix*

Setelah tokenisasi, berikut langkah mengubah notasi *infix* menjadi *postfix*.

Siapkan suatu *stack* kosong, serta tempat (misal *string* kosong) untuk menyimpan hasil infix. Lalu, *scanning* (melihat satu-per-satu) tiap token dari kiri ke kanan, dan ikuti ketentuan berikut:

1. Apabila token adalah operand/angka, langsung tambahkan ke hasil infix
2. Apabila *stack* kosong, atau apabila elemen teratas pada *stack* adalah kurung kiri, maka push token tersebut ke dalam *stack*
3. Apabila token adalah kurung kiri yaitu "(", push ke dalam *stack*
4. Apabila token adalah kurung kanan yaitu ")", lakukan while loop: lakukan pop pada stack, masukkan hasil pop tersebut ke hasil infix, hentikan while loop apabila hasil pop tersebut adalah kurung kiri.
5. Apabila token memiliki *precedence* yang lebih tinggi daripada elemen teratas pada *stack*, maka push token tersebut ke dalam *stack*.
6. Apabila token memiliki *precedence* yang lebih rendah daripada elemen teratas pada *stack*, lakukan langkah berikut: lakukan pop pada stack, lalu masukkan hasil pop tersebut ke hasil infix.
7. Apabila token memiliki *precedence* yang setara dengan elemen teratas pada *stack*, perhatikan *associativity* dari operator tersebut, lalu:
   a. Apabila untuk operator tersebut bersifat left-to-right: lakukan pop pada stack, masukkan hasil pop ke hasil infix, lalu push token
   b. Sedangkan apabila bersifat right-to-left: push token tersebut ke dalam stack

Setelah suatu token teratasi, tentunya langsung lanjut melihat token berikutnya. Apabila semua token sudah teratasi sedangkan stack belum kosong, maka ulangi sampai stack kosong: lakukan pop, masukkan hasil pop ke hasil infix.

Notasi *infix* menjadi *prefix*

Evaluasi notasi *prefix*

Evaluasi notasi *postfix*

Notasi *postfix* menjadi *infix*

Notasi *prefix* menjadi *infix*

Modul 5 Struktur Data: *Queue* dan berbagai implementasinya

Di praktikum kali ini, kita akan membahas tentang struktur data *queue* serta berbagai "implementasi"nya dalam Python (yaitu berbagai cara membuat struktur data *queue* di Python), baik menggunakan *array* maupun *linked list*.

Queue itu sendiri adalah suatu struktur data dengan dua ujung, di mana data bisa dimasukkan dari salah satu ujung tertentu (yang disebut *rear*) dan data bisa dikeluarkan dari ujung yang satunya lagi (yang disebut *front*). *Queue* dikatakan menganut prinsip FIFO (*First In First Out*), karena data yang pertama masuk akan menjadi data yang pertama keluar.

Kita akan menggunakan *array* dari numpy, sehingga perlu melakukan import:

```
import numpy as np
```

Implementasi (*linear*) *queue* dengan *array*

```
class ArrayLinQueue:
    def __init__(self, dtype, array_max):
        self.dtype = dtype
        self.array_max = array_max
        self.array = np.empty(array_max, dtype=dtype)
        self.front = -1
        self.rear = -1

    def get_size(self):
        size = (self.rear - self.front) + 1
        return size

    def get_capacity_array(self):
        return self.array_max

    def get_capacity_queue(self):
        if self.front == -1:
            capacity_queue = self.array_max
        else:
            capacity_queue = self.array_max - self.front
        return capacity_queue

    def is_empty(self):
        if self.front == -1:
            return True
        else:
```

```
            return False

    def is_full(self):
        if self.rear == self.array_max - 1:
            return True
        else:
            return False

    def enqueue(self, newdata):
        if self.is_full():
            print("Error enqueue: queue sudah penuh sebelumnya")
        elif self.front == -1:
            self.front += 1
            self.rear += 1
            self.array[self.rear] = newdata
        else:
            self.rear += 1
            self.array[self.rear] = newdata

    def peek(self):
        if self.is_empty():
            print("Error peek: queue sedang kosong")
        else:
            return self.array[self.front]

    def dequeue(self):
        if self.is_empty():
            print("Error dequeue: queue sudah kosong sebelumnya")
            return None
        elif (self.get_size() == 1):
            # Jika di queue hanya ada satu elemen, dan ingin di-dequeue,
            # maka queue akan kosong setelah itu
            output = self.array[self.front]
            self.front = -1
            self.rear = -1
            return output
        else:
            output = self.array[self.front]
            self.front += 1
            return output

    def print_storage(self):
        print(self.array)

    def print_queue(self):
        print("front : ", end="")
        if self.is_empty():
            print("(tidak ada data) : rear")
        else:
            for i in range(self.front, self.rear): # i = front, ..., rear-1
                print(self.array[i], end=" | ")
            print(self.array[self.rear], end="") # untuk i = rear
            print(" : rear")
```

```
arraylinqueue = ArrayLinQueue(int, 5)
```

```
        arraylinqueue.print_queue()
```

```
front : (tidak ada data) : rear
```

```
        arraylinqueue.print_storage()
```

```
[                  0 4602678819172646912 4607182418800017408
 4609434218613702656 4611686018427387904]
```

```
        arraylinqueue.enqueue(-18)
        arraylinqueue.enqueue(67)
        arraylinqueue.enqueue(32)
```

```
        arraylinqueue.print_queue()
```

```
front : -18 | 67 | 32 : rear
```

```
        arraylinqueue.print_storage()
```

```
[                -18                  67                  32
 4609434218613702656 4611686018427387904]
```

```
        print(arraylinqueue.front)
        print(arraylinqueue.rear)
```

```
0
2
```

```
        arraylinqueue.enqueue(-29)
```

```
        arraylinqueue.print_queue()
```

```
front : -18 | 67 | 32 | -29 : rear
```

```
        arraylinqueue.print_storage()
```

```
[                -18                  67                  32
                 -29 4611686018427387904]
```

```
        print(arraylinqueue.front)
        print(arraylinqueue.rear)
```

```
0
3
```

```
        print(arraylinqueue.peek())
```

```
-18
```

```
        arraylinqueue.print_queue()
```

```
front : -18 | 67 | 32 | -29 : rear
```

```
        nilai = arraylinqueue.dequeue()
        print(nilai)
```

```
-18
```

```
        arraylinqueue.print_queue()
```

```
front : 67 | 32 | -29 : rear
```

```
        arraylinqueue.print_storage()
```

```
[               -18                 67                  32
                -29 4611686018427387904]
```

```
        print(arraylinqueue.front)
        print(arraylinqueue.rear)
```

```
1
3
```

```
        print(arraylinqueue.dequeue())
```

```
67
```

```
        arraylinqueue.print_queue()
```

```
front : 32 | -29 : rear
```

```
        print(arraylinqueue.dequeue())
        print(arraylinqueue.dequeue())
```

```
32
-29
```

```
        arraylinqueue.print_queue()
```

```
front : (tidak ada data) : rear
```

```
        print(arraylinqueue.front)
        print(arraylinqueue.rear)
```

```
-1
-1
```

```
        print(arraylinqueue.dequeue())
```

```
Error dequeue: queue sudah kosong sebelumnya
None
```

```
        arraylinqueue.enqueue(-25)
        arraylinqueue.enqueue(13)
        arraylinqueue.enqueue(48)
        arraylinqueue.enqueue(-87)
        arraylinqueue.enqueue(38)
```

```
        arraylinqueue.print_queue()
```

```
front : -25 | 13 | 48 | -87 | 38 : rear
```

```
        arraylinqueue.print_storage()
```

```
[-25  13  48 -87  38]
```

```
        print(arraylinqueue.is_full())
```

```
True
```

```
        print(arraylinqueue.front)
        print(arraylinqueue.rear)
```

```
0
4
```

```
        arraylinqueue.enqueue(-53)
```

```
Error enqueue: queue sudah penuh sebelumnya
```

```
        print(arraylinqueue.dequeue())
        print(arraylinqueue.dequeue())
```

```
-25
13
```

```
        arraylinqueue.print_queue()
```

```
front : 48 | -87 | 38 : rear
```

```
        arraylinqueue.print_storage()
```

```
[-25  13  48 -87  38]
```

```
        print(arraylinqueue.front)
        print(arraylinqueue.rear)
```

```
2
4
```

```
        arraylinqueue.enqueue(-53)
```

```
Error enqueue: queue sudah penuh sebelumnya
```

Implementasi *circular queue* dengan *array*

```
class ArrayCircQueue:
    def __init__(self, dtype, max):
        self.dtype = dtype
        self.max = max
        self.array = np.empty(max, dtype=dtype)
        self.front = -1
        self.rear = -1

    def is_empty(self):
        if self.front == -1:
            return True
        else:
            return False

    def is_full(self):
        if self.front == (self.rear + 1) % self.max:
            return True
        else:
            return False

    def get_size(self):
        if self.is_empty():
            size = 0
        elif self.front <= self.rear:
            size = (self.rear - self.front) + 1
        else:
            size = self.max - (self.front - self.rear - 1)
        return size

    def get_capacity(self):
        return self.max

    def enqueue(self, newdata):
        if self.is_full():
            print("Error enqueue: queue sudah penuh sebelumnya")
        elif self.front == -1:
            self.front += 1
            self.rear += 1
            self.array[self.rear] = newdata
        else:
            self.rear = (self.rear + 1) % self.max # hanya berbeda di sini
            self.array[self.rear] = newdata

    # Masih sama persis
    def peek(self):
        if self.is_empty():
            print("Error peek: queue sedang kosong")
        else:
            return self.array[self.front]

    def dequeue(self):
        if self.is_empty():
            print("Error dequeue: queue sudah kosong sebelumnya")
            return None
```

```
                elif (self.get_size() == 1):
                    # Jika di queue hanya ada satu elemen, dan ingin di-dequeue,
                    # maka queue akan kosong setelah itu
                    output = self.array[self.front]
                    self.front = -1
                    self.rear = -1
                    return output
                else:
                    output = self.array[self.front]
                    self.front = (self.front + 1) % self.max # hanya berbeda di sini
                    return output

            def print_storage(self):
                print(self.array)

            def print_queue(self):
                print("front : ", end="")
                if self.is_empty():
                    print("(tidak ada data) : rear")
                else:
                    # i = front, ..., rear-1 (kurang lebih begitu)
                    i = self.front
                    while i != self.rear:
                        print(self.array[i], end=" | ")
                        i = (i + 1) % self.max
                    # untuk i = rear
                    print(self.array[self.rear], end="")
                    print(" : rear")
```

```
        arraycircqueue = ArrayCircQueue(int, 5)
        arraycircqueue.print_queue()
```

```
front : (tidak ada data) : rear
```

```
        arraycircqueue.print_storage()
```

```
[4607182418800017408 4613374868287651840 4618441417868443648
 4622241330054037504 4625478292286210048]
```

```
        arraycircqueue.enqueue(65)
        arraycircqueue.enqueue(-11)
        arraycircqueue.enqueue(43)
```

```
        arraycircqueue.print_queue()
```

```
front : 65 | -11 | 43 : rear
```

```
        arraycircqueue.print_storage()
```

```
[                65                 -11                 43
 4622241330054037504 4625478292286210048]
```

```
        print(arraycircqueue.front)
        print(arraycircqueue.rear)
```

```
0
2
```

```
        arraycircqueue.enqueue(97)
        arraycircqueue.enqueue(-12)
```

```
        arraycircqueue.print_queue()
```

```
front : 65 | -11 | 43 | 97 | -12 : rear
```

```
        arraycircqueue.enqueue(41)
```

```
Error enqueue: queue sudah penuh sebelumnya
```

```
        arraycircqueue.print_storage()
```

```
[ 65 -11  43  97 -12]
```

```
        print(arraycircqueue.front)
        print(arraycircqueue.rear)
```

```
0
4
```

```
        print(arraycircqueue.peek())
```

```
65
```

```
        arraycircqueue.print_queue()
```

```
front : 65 | -11 | 43 | 97 | -12 : rear
```

```
        print(arraycircqueue.dequeue())
```

```
65
```

```
        arraycircqueue.print_queue()
```

```
front : -11 | 43 | 97 | -12 : rear
```

```
        arraycircqueue.print_storage()
```

```
[ 65 -11  43  97 -12]
```

```
        print(arraycircqueue.front)
        print(arraycircqueue.rear)
```

```
1
4
```

```
        print(arraycircqueue.dequeue())
        print(arraycircqueue.dequeue())
```

```
-11
43
```

```
        arraycircqueue.print_queue()
```

```
front : 97 | -12 : rear
```

```
        print(arraycircqueue.front)
        print(arraycircqueue.rear)
```

```
3
4
```

```
        arraycircqueue.print_storage()
```

```
[ 65 -11  43  97 -12]
```

```
        arraycircqueue.enqueue(-74)
```

```
        arraycircqueue.print_queue()
```

```
front : 97 | -12 | -74 : rear
```

```
        arraycircqueue.print_storage()
```

```
[-74 -11  43  97 -12]
```

```
        print(arraycircqueue.front)
        print(arraycircqueue.rear)
```

```
3
0
```

```
        arraycircqueue.enqueue(19)
```

```
        arraycircqueue.print_queue()
```

```
front : 97 | -12 | -74 | 19 : rear
```

```
        arraycircqueue.print_storage()
```

```
[-74  19  43  97 -12]
```

```
        print(arraycircqueue.front)
        print(arraycircqueue.rear)
```

```
3
1
```

```
        arraycircqueue.enqueue(85)
```

```
        arraycircqueue.print_queue()
```

```
front : 97 | -12 | -74 | 19 | 85 : rear
```

```
        arraycircqueue.print_storage()
```

```
[-74  19  85  97 -12]
```

```
        print(arraycircqueue.front)
        print(arraycircqueue.rear)
```

```
3
2
```

```
        arraycircqueue.enqueue(-31)
```

```
Error enqueue: queue sudah penuh sebelumnya
```

```
        print(arraycircqueue.dequeue())
```

```
97
```

```
        arraycircqueue.print_queue()
```

```
front : -12 | -74 | 19 | 85 : rear
```

```
        arraycircqueue.print_storage()
```

```
[-74  19  85  97 -12]
```

```
        print(arraycircqueue.front)
        print(arraycircqueue.rear)
```

```
4
2
```

```
        print(arraycircqueue.dequeue())
```

```
-12
```

```
        arraycircqueue.print_queue()
```

```
front : -74 | 19 | 85 : rear
```

```
        arraycircqueue.print_storage()
```

```
[-74  19  85  97 -12]
```

```
        print(arraycircqueue.front)
        print(arraycircqueue.rear)
```

```
0
2
```

```
        arraycircqueue.enqueue(27)
```

```
        arraycircqueue.print_queue()
```

```
front : -74 | 19 | 85 | 27 : rear
```

```
        arraycircqueue.print_storage()
```

```
[-74  19  85  27 -12]
```

```
        print(arraycircqueue.front)
        print(arraycircqueue.rear)
```

```
0
3
```

```
        print(arraycircqueue.dequeue())
```

```
-74
```

```
        arraycircqueue.print_queue()
```

```
front : 19 | 85 | 27 : rear
```

```
        arraycircqueue.print_storage()
```

```
[-74  19  85  27 -12]
```

```
        print(arraycircqueue.front)
        print(arraycircqueue.rear)
```

```
1
3
```

Implementasi (*linear*) *queue* dengan *linked list*

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

```
        self.data = data
        self.next = next
```

```
class SLLinQueue:
    def __init__(self):
        # head=front, tail=rear
        self.front = None
        self.rear = None

    def is_empty(self):
        if self.front == None:
            return True
        else:
            return False

    def get_size(self):
        size = 0
        temp = self.front
        while (temp != None):
            size += 1
            temp = temp.next
        return size

    # insert di akhir linked list
    def enqueue(self, newdata):
        newnode = SLNode(newdata)
        if self.is_empty():
            self.front = newnode
            self.rear = newnode
        else:
            self.rear.next = newnode
            self.rear = newnode

    def peek(self):
        if self.is_empty():
            print("Error peek: queue sedang kosong")
            return None
        else:
            return self.front.data

    # hapus di awal linked list
    def dequeue(self):
        if self.is_empty():
            print("Error dequeue: queue sudah kosong sebelumnya")
            return None
        else:
            output = self.front.data
            temp = self.front
            self.front = self.front.next
            del temp
            return output

    def print_queue(self):
        print("front : ", end="")
        if self.is_empty():
            print("(tidak ada data) : rear")
```

```
                else:
                    temp = self.front
                    while temp != None:
                        if temp.next != None:
                            print(temp.data, end = " | ")
                        else:
                            print(temp.data, end="")
                        temp = temp.next
                    print(" : rear")

            def print_storage(self):
                print("front -> ", end="")
                if self.is_empty():
                    print("None <- rear")
                else:
                    temp = self.front
                    while temp != None:
                        if temp.next != None:
                            print(temp.data, end = " -> ")
                        else:
                            print(temp.data, end = " <- ")
                        temp = temp.next
                    print("rear")
```

```
        sllinqueue = SLLinQueue()
        sllinqueue.print_queue()
        sllinqueue.print_storage()
```

```
front : (tidak ada data) : rear
front -> None <- rear
```

```
        sllinqueue.enqueue(10)
        sllinqueue.print_queue()
        sllinqueue.print_storage()
```

```
front : 10 : rear
front -> 10 <- rear
```

```
        sllinqueue.enqueue(98)
        sllinqueue.print_queue()
        sllinqueue.print_storage()
```

```
front : 10 | 98 : rear
front -> 10 -> 98 <- rear
```

```
        sllinqueue.enqueue(-43)
        sllinqueue.print_queue()
        sllinqueue.print_storage()
```

```
front : 10 | 98 | -43 : rear
front -> 10 -> 98 -> -43 <- rear
```

```
        print(sllinqueue.peek())
```

```
10
```

```
        sllinqueue.print_queue()
        sllinqueue.print_storage()
```

```
front : 10 | 98 | -43 : rear
front -> 10 -> 98 -> -43 <- rear
```

```
        print(sllinqueue.dequeue())
```

```
10
```

```
        sllinqueue.print_queue()
        sllinqueue.print_storage()
```

```
front : 98 | -43 : rear
front -> 98 -> -43 <- rear
```

Implementasi *circular queue* dengan (*circular*) *linked list*

```
        class SLCircQueue:
            def __init__(self):
                # head=front, tail=rear
                self.front = None
                self.rear = None

            def is_empty(self):
                if self.front == None:
                    return True
                else:
                    return False

            def get_size(self):
                size = 0
                temp = self.front
                if temp == None:
                    return size
                else:
                    size += 1
                    temp = temp.next
                while (temp != self.front):
                    size += 1
                    temp = temp.next
                return size

            def enqueue(self, newdata):
                newnode = SLNode(newdata)
                if self.is_empty():
                    self.front = newnode
                    self.rear = newnode
                    newnode.next = newnode
                else:
                    self.rear.next = newnode
                    self.rear = newnode
```

```
            newnode.next = self.front

        # masih sama persis
        def peek(self):
            if self.is_empty():
                print("Error peek: queue sedang kosong")
                return None
            else:
                return self.front.data

        def dequeue(self):
            if self.is_empty():
                print("Error dequeue: queue sudah kosong sebelumnya")
                return None
            elif (self.front == self.rear): # sama saja self.get_size() == 1
                output = self.front.data
                del self.front
                self.front = None
                self.rear = None
                return output
            else:
                output = self.front.data
                temp = self.front
                self.front = self.front.next
                del temp
                self.rear.next = self.front
                return output

        def print_queue(self):
            print("front : ", end="")
            if self.is_empty():
                print("(tidak ada data) : rear")
            else:
                temp = self.front
                while temp.next != self.front:
                    print(temp.data, end = " | ")
                    temp = temp.next
                print(temp.data, end="")
                print(" : rear")

        def print_storage(self):
            print("front -> ", end="")
            if self.is_empty():
                print("None (<- rear)")
            else:
                temp = self.front
                while temp.next != self.front:
                    print(temp.data, end = " -> ")
                    temp = temp.next
                print(temp.data, end = "")
                print(" (<- rear) -> front")
```

```
slcircqueue = SLCircQueue()
slcircqueue.print_queue()
slcircqueue.print_storage()
```

```
front : (tidak ada data) : rear
front -> None (<- rear)
```

```
        slcircqueue.enqueue(-91)
        slcircqueue.print_queue()
        slcircqueue.print_storage()
```

```
front : -91 : rear
front -> -91 (<- rear) -> front
```

```
        slcircqueue.enqueue(14)
        slcircqueue.print_queue()
        slcircqueue.print_storage()
```

```
front : -91 | 14 : rear
front -> -91 -> 14 (<- rear) -> front
```

```
        slcircqueue.enqueue(30)
        slcircqueue.print_queue()
        slcircqueue.print_storage()
```

```
front : -91 | 14 | 30 : rear
front -> -91 -> 14 -> 30 (<- rear) -> front
```

```
        slcircqueue.peek()
```

```
-91
```

```
        slcircqueue.print_queue()
        slcircqueue.print_storage()
```

```
front : -91 | 14 | 30 : rear
front -> -91 -> 14 -> 30 (<- rear) -> front
```

```
        print(slcircqueue.dequeue())
```

```
-91
```

```
        slcircqueue.print_queue()
        slcircqueue.print_storage()
```

```
front : 14 | 30 : rear
front -> 14 -> 30 (<- rear) -> front
```

(TODO) Pengayaan: *Deque* atau *double-ended queue* (DEQ)

Modul 6 Struktur Data: *Binary Tree*, *Binary Search Tree* (BST)

```
import numpy as np
import graphviz as gv
```

Implementasi *binary tree*

Binary Tree dengan *array*

```
class ArrayBintree:
    def __init__(self, dtype, height, emptydata=-9999):
        self.dtype = dtype
        self.height = height
        self.emptydata = emptydata
        self.array_size = 2**(height+1) - 1
        self.array = np.empty(self.array_size, dtype=dtype)
        for i in range(self.array_size):
            self.array[i] = emptydata

    def get_root(self):
        root_data = self.array[0]
        if root_data == self.emptydata:
            return None
        else:
            return root_data

    def set_root(self, newdata):
        self.array[0] = newdata

    def get_data(self, node_idx):
        if node_idx < self.array_size:
            return self.array[node_idx]
        else:
            print("Error get_data: indeks di luar ukuran tree")
            return None

    def set_data(self, node_idx, newdata):
        if node_idx < self.array_size:
            self.array[node_idx] = newdata
        else:
            print("Error set_data: indeks di luar ukuran tree")

    def get_left_child_idx(self, node_idx):
        left_idx = 2*node_idx + 1
        if left_idx < self.array_size:
```

```
            return left_idx
        else:
            return -1

    def get_left_child(self, node_idx):
        left_idx = self.get_left_child_idx(node_idx)
        if left_idx != -1:
            data = self.array[left_idx]
            if data != self.emptydata:
                return data
            else:
                return None
        else:
            return None

    def get_right_child_idx(self, node_idx):
        right_idx = 2*node_idx + 2
        if right_idx < self.array_size:
            return right_idx
        else:
            return -1

    def get_right_child(self, node_idx):
        right_idx = self.get_right_child_idx(node_idx)
        if right_idx != -1:
            data = self.array[right_idx]
            if data != self.emptydata:
                return data
            else:
                return None
        else:
            return None

    def get_parent_idx(self, node_idx):
        if node_idx == 0:
            return -1
        idx = int(np.floor( (node_idx - 1)/2 ))
        return idx

    # preorder: tengah, kiri, kanan
    def get_preorder(self, current=0, result=None):
        is_starting_node = False
        if result == None:
            is_starting_node = True
            result = []

        # tengah
        current_data = self.array[current]
        if current_data != self.emptydata:
            result.append(current_data)

        # kiri
        left_idx = self.get_left_child_idx(current)
        if left_idx != -1:
            self.get_preorder(current=left_idx, result=result)

        # kanan
```

```
        right_idx = self.get_right_child_idx(current)
        if right_idx != -1:
            self.get_preorder(current=right_idx, result=result)

        if is_starting_node:
            return result

    # inorder: kiri, tengah, kanan
    def get_inorder(self, current=0, result=None):
        is_starting_node = False
        if result == None:
            is_starting_node = True
            result = []

        # kiri
        left_idx = self.get_left_child_idx(current)
        if left_idx != -1:
            self.get_inorder(current=left_idx, result=result)

        # tengah
        current_data = self.array[current]
        if current_data != self.emptydata:
            result.append(current_data)

        # kanan
        right_idx = self.get_right_child_idx(current)
        if right_idx != -1:
            self.get_inorder(current=right_idx, result=result)

        if is_starting_node:
            return result

    # postorder: kiri, kanan, tengah
    def get_postorder(self, current=0, result=None):
        is_starting_node = False
        if result == None:
            is_starting_node = True
            result = []

        # kiri
        left_idx = self.get_left_child_idx(current)
        if left_idx != -1:
            self.get_postorder(current=left_idx, result=result)

        # kanan
        right_idx = self.get_right_child_idx(current)
        if right_idx != -1:
            self.get_postorder(current=right_idx, result=result)

        # tengah
        current_data = self.array[current]
        if current_data != self.emptydata:
            result.append(current_data)

        if is_starting_node:
            return result
```

```
            def get_digraph_simple(self):
                digraph = gv.Digraph()
                for idx in range(self.array_size):
                    data = self.array[idx]
                    if data != self.emptydata:
                        digraph.node("node" + str(idx), label=str(data))
                        left_idx = self.get_left_child_idx(idx)
                        right_idx = self.get_right_child_idx(idx)
                        if left_idx != -1:
                            digraph.edge("node" + str(idx), "node" + str(left_idx))
                            if self.array[left_idx] == self.emptydata:
                                digraph.node("node" + str(left_idx), label="NULL", sha
                        if right_idx != -1:
                            digraph.edge("node" + str(idx), "node" + str(right_idx))
                            if self.array[right_idx] == self.emptydata:
                                digraph.node("node" + str(right_idx), label="NULL", sh
                return digraph
```

```
        arraybintree = ArrayBintree(int, 2)
```

```
        print(arraybintree.array)
```

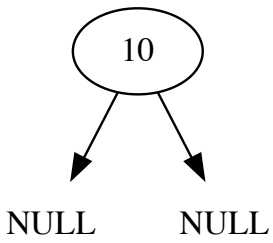
```
[-9999 -9999 -9999 -9999 -9999 -9999 -9999]
```

```
        arraybintree.set_root(10)
```

```
        print(arraybintree.array)
```

```
[   10 -9999 -9999 -9999 -9999 -9999 -9999]
```

```
        display(arraybintree.get_digraph_simple())
```

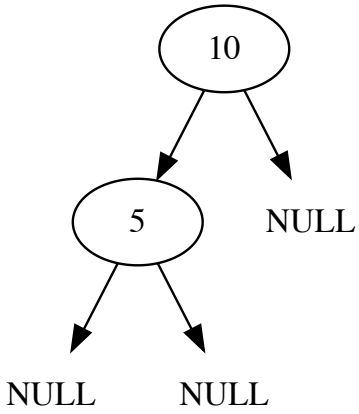



```
        arraybintree.set_data(
            arraybintree.get_left_child_idx(0),
            5
        )
```

```
        print(arraybintree.array)
```

```
[   10     5 -9999 -9999 -9999 -9999 -9999]
```

```
        display(arraybintree.get_digraph_simple())
```

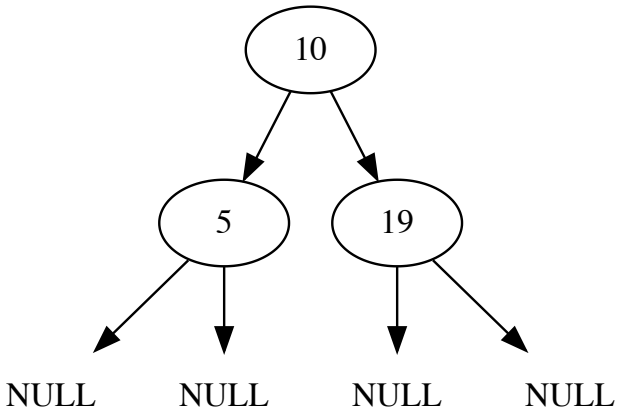



```
        arraybintree.set_data(
            arraybintree.get_right_child_idx(0),
            19
        )
```

```
        print(arraybintree.array)
```

```
[   10     5    19 -9999 -9999 -9999 -9999]
```

```
        display(arraybintree.get_digraph_simple())
```

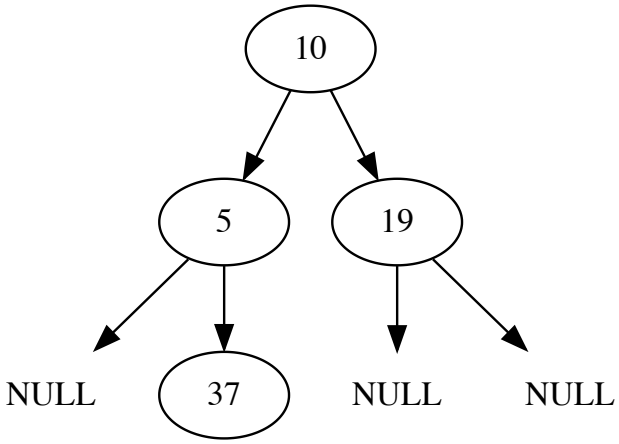



```
        arraybintree.set_data(
            arraybintree.get_right_child_idx(arraybintree.get_left_child_idx(0)),
            37
        )
```

```
        print(arraybintree.array)
```

```
[   10     5    19 -9999    37 -9999 -9999]
```

```
        display(arraybintree.get_digraph_simple())
```




```
        arraybintree.get_data(
            arraybintree.get_right_child_idx(arraybintree.get_left_child_idx(0))
        )
```

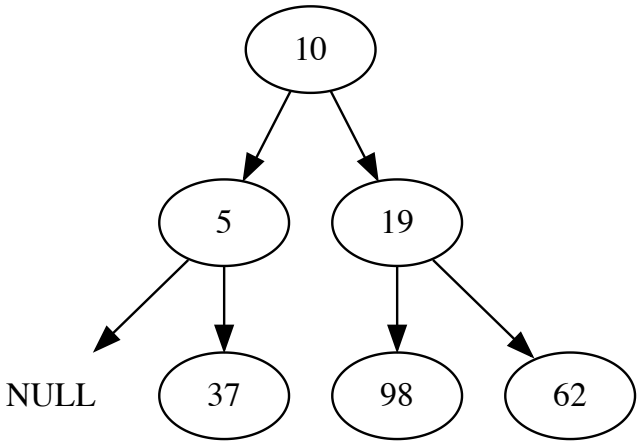
```
37
```

```
        arraybintree.array[5] = 98
        arraybintree.array[6] = 62
```

```
        print(arraybintree.array)
```

```
[   10     5    19 -9999    37    98    62]
```

```
        display(arraybintree.get_digraph_simple())
```

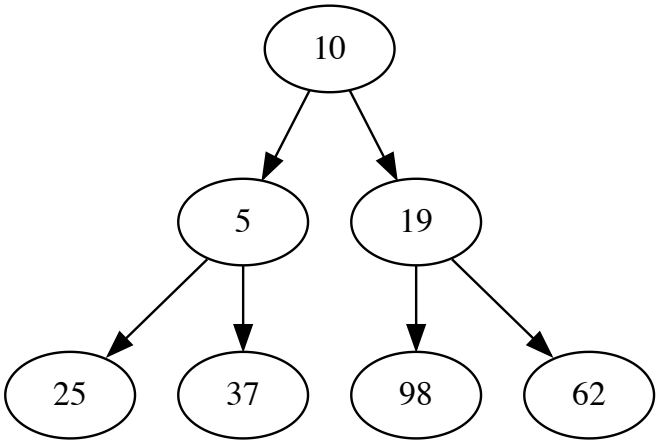



```
        arraybintree.array[3] = 25
```

```
        print(arraybintree.array)
```

```
[10  5 19 25 37 98 62]
```

```
        display(arraybintree.get_digraph_simple())
```




```
arraybintree.get_preorder()
```

```
[10, 5, 25, 37, 19, 98, 62]
```

```
arraybintree.get_inorder()
```

```
[25, 5, 37, 10, 98, 19, 62]
```

```
arraybintree.get_postorder()
```

```
[25, 37, 5, 98, 62, 19, 10]
```

Binary Tree dengan *pointer* (*linked binary tree*)

```
class BintreeNode:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right
```

```
class LinkedBintree:
    def __init__(self):
        self.root = None

    def is_empty(self):
        if self.root == None:
            return True
        else:
            return False

    def get_root_data(self):
        if self.is_empty():
            print("Error get_root_data: tree sedang kosong")
            return None
        else:
            return self.root.data
```

```
    def set_root_data(self, newdata):
        if self.is_empty():
            self.root = BintreeNode(newdata)
        else:
            self.root.data = newdata

    # preorder: tengah, kiri, kanan
    def get_preorder(self, current=None, result=None, get_addresses=False):
        is_starting_node = False
        if result == None:
            is_starting_node = True
            result = []
            current = self.root

        if current != None:
            # tengah
            if (not get_addresses):
                result.append(current.data)
            else:
                result.append(current)

            # kiri
            if current.left != None:
                self.get_preorder(current.left, result=result)

            # kanan
            if current.right != None:
                self.get_preorder(current.right, result=result)

        if is_starting_node:
            return result

    # inorder: kiri, tengah, kanan
    def get_inorder(self, current=None, result=None, get_addresses=False):
        is_starting_node = False
        if result == None:
            is_starting_node = True
            result = []
            current = self.root

        if current != None:
            # kiri
            if current.left != None:
                self.get_inorder(current.left, result=result)

            # tengah
            if (not get_addresses):
                result.append(current.data)
            else:
                result.append(current)

            # kanan
            if current.right != None:
                self.get_inorder(current.right, result=result)

        if is_starting_node:
```

```
            return result

        # postorder: kiri, kanan, tengah
        def get_postorder(self, current=None, result=None, get_addresses=False):
            is_starting_node = False
            if result == None:
                is_starting_node = True
                result = []
                current = self.root

            if current != None:
                # kiri
                if current.left != None:
                    self.get_postorder(current.left, result=result)

                # kanan
                if current.right != None:
                    self.get_postorder(current.right, result=result)

                # tengah
                if (not get_addresses):
                    result.append(current.data)
                else:
                    result.append(current)

            if is_starting_node:
                return result

        # berdasarkan algoritma preorder traversal :D
        def get_digraph_simple(self, current=None, node_name=None, result=None):
            is_starting_node = False
            if result == None:
                is_starting_node = True
                result = gv.Digraph()
                current = self.root
                node_name = "root"

            if current != None:
                # tengah
                result.node(node_name, label=str(current.data))

                # kiri
                left_name = node_name + "->left"
                result.edge(node_name, left_name)
                self.get_digraph_simple(
                    current=current.left, node_name=left_name, result=result
                )

                # kanan
                right_name = node_name + "->right"
                self.get_digraph_simple(
                    current=current.right, node_name=right_name, result=result
                )
                result.edge(node_name, right_name)
            else:
                result.node(node_name, label="NULL", shape="none")
```

```
                if is_starting_node:
                    return result
```

```
        linkedbintree = LinkedBintree()
```

```
        print(linkedbintree.root)
```

```
None
```

```
        linkedbintree.root = BintreeNode(26)
```

```
        print(linkedbintree.root)
```

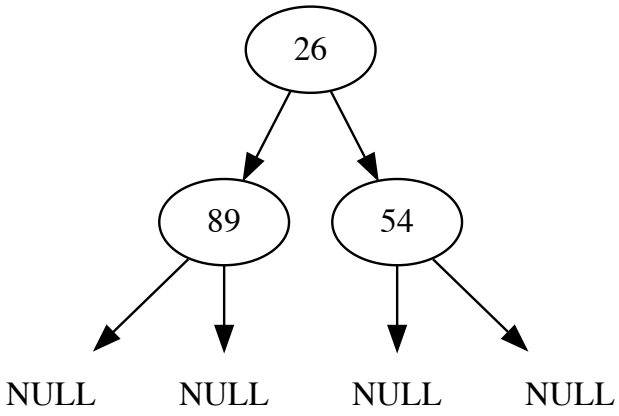
```
<__main__.BintreeNode object at 0x10ccbd060>
```

```
        print(linkedbintree.root.data)
```

```
26
```

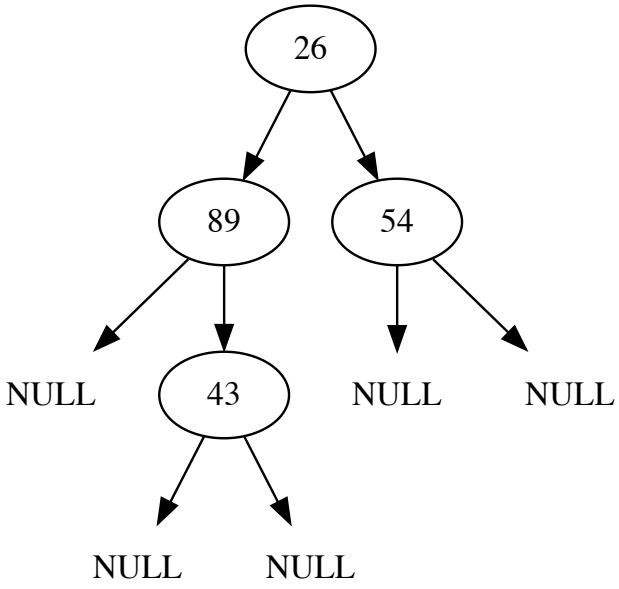
```
        linkedbintree.root.left = BintreeNode(89)
        linkedbintree.root.right = BintreeNode(54)
```

```
        display(linkedbintree.get_digraph_simple())
```




```
        linkedbintree.root.left.right = BintreeNode(43)
```

```
        display(linkedbintree.get_digraph_simple())
```

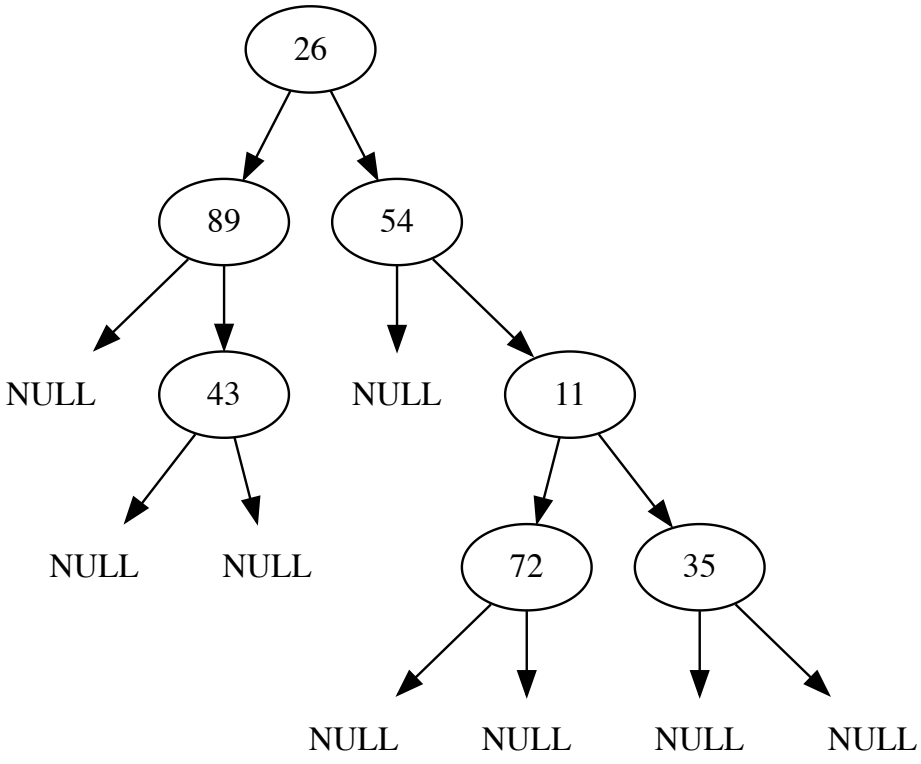



```
print(linkedbintree.root.left.right.data)
```

```
43
```

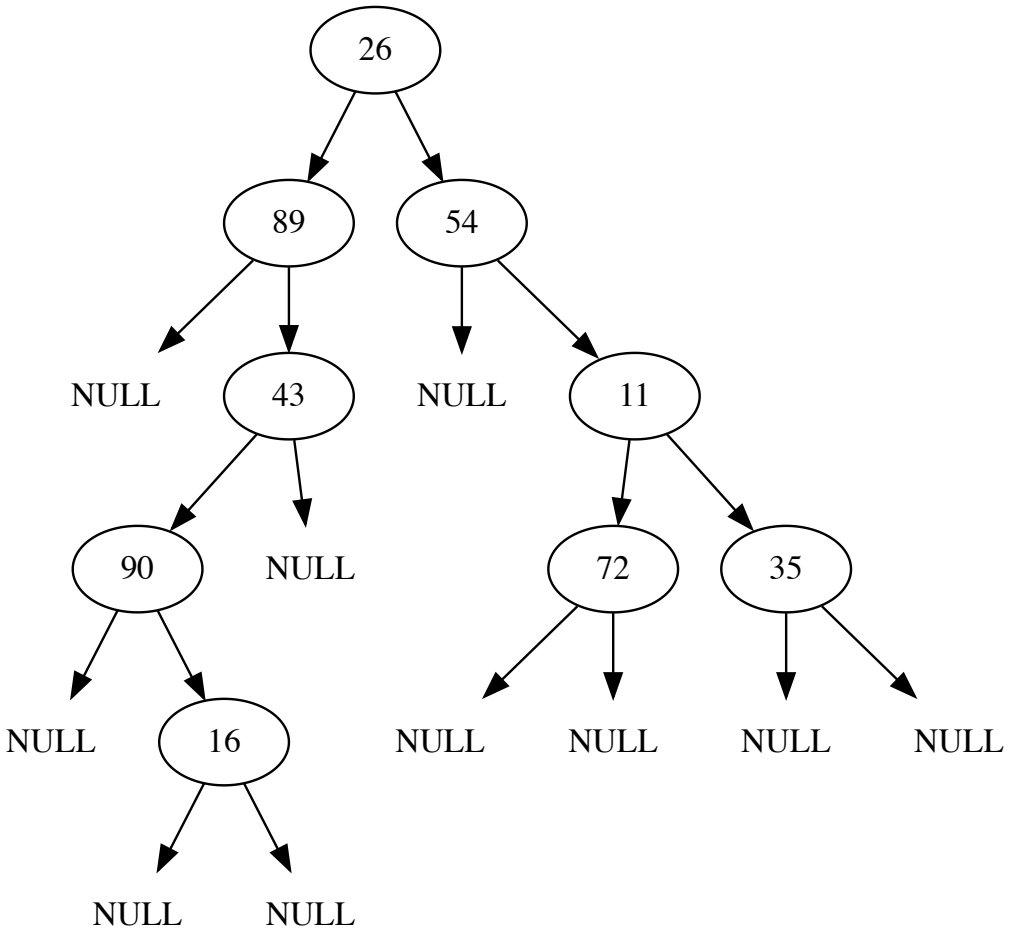
```
linkedbintree.root.right.right = BintreeNode(11)
linkedbintree.root.right.right.left = BintreeNode(72)
linkedbintree.root.right.right.right = BintreeNode(35)
```

```
display(linkedbintree.get_digraph_simple())
```



```
        linkedbintree.root.left.right.left = BintreeNode(90)
        linkedbintree.root.left.right.left.right = BintreeNode(16)
```

```
        display(linkedbintree.get_digraph_simple())
```




```
        linkedbintree.get_preorder()
```

```
[26, 89, 43, 90, 16, 54, 11, 72, 35]
```

```
        linkedbintree.get_inorder()
```

```
[89, 90, 16, 43, 26, 54, 72, 11, 35]
```

```
        linkedbintree.get_postorder()
```

```
[16, 90, 43, 89, 72, 35, 11, 54, 26]
```

Binary Search Tree (BST) dengan *pointer* (*linked* BST)

Binary Search Tree (BST) adalah *binary tree* dengan beberapa sifat dan fitur tambahan. Sehingga, untuk implementasi BST, kita cukup menambahkan beberapa *method* ke `class` *binary tree* yang sudah dibuat.

Daripada mengetik ulang semua *method* yang sudah dibuat di `class` *binary tree*, kita bisa menerapkan salah satu prinsip OOP yaitu *inheritance*, agar langsung mewariskan semua fitur yang sudah dibuat di implementasi *binary tree*.

Karena lebih fleksibel (tidak ada keterbatasan ukuran), kita akan membuat BST dengan *pointer* (juga disebut *linked* BST) saja, berarti meng-*inherit* dari `class LinkedBintree`.

(Membuat BST dengan *array* juga memungkinkan, meng-*inherit* dari `class ArrayBintree`, tetapi akan ada beberapa pertimbangan tambahan, misalnya untuk memastikan posisi *node* yang di-*insert* tidak melebihi kapastias *array*.)

```
class LinkedBST(LinkedBintree):
    def __init__(self):
        # menggunakan __init__ dari parent class,
        # melalui super() yaitu parent class
        super().__init__()

    # semua method dari LinkedBintree otomatis sudah terdefinisi

    # cari elemen di BST
    def search(self, x):
        temp = self.root
        while (temp != None):
            if x == temp.data:
                return x
            elif x < temp.data:
                temp = temp.left
            else:
                temp = temp.right
        return None

    # insertion
    def insert(self, newdata):
        if self.root == None:
            self.root = BintreeNode(newdata)
            return
        temp = self.root
        while (temp != None):
            if newdata == temp.data:
                print("Error insert: data sudah ada di BST, yaitu", newdata)
                return
            elif newdata < temp.data:
                if temp.left == None:
                    temp.left = BintreeNode(newdata)
                    return
                else:
                    temp = temp.left
            else: # newdata > temp.data
                if temp.right == None:
                    temp.right = BintreeNode(newdata)
                    return
                else:
                    temp = temp.right

    # deletion
    def delete(self, x, inorder_pred=False):
```

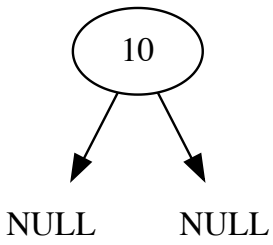
```
        if self.is_empty():
            print("Error: BST kosong")
            return
        prev = self.root
        turn = ""
        if x < prev.data:
            if prev.left == None:
                print("Error delete: tidak ditemukan data yang bernilai", x)
                return
            else:
                temp = prev.left
                turn = "left"
        elif x > prev.data:
            if prev.right == None:
                print("Error delete: tidak ditemukan data yang bernilai", x)
                return
            else:
                temp = prev.right
                turn = "right"
        else:
            temp = prev

        while (temp != None):
            if temp.data == x:
                break
            elif x < temp.data:
                if temp.left == None:
                    print("Error delete: tidak ditemukan data yang bernilai",
                    return
                else:
                    prev = temp
                    temp = temp.left
                    turn = "left"
            else: # x > temp.data
                if temp.right == None:
                    print("Error delete: tidak ditemukan data yang bernilai",
                    return
                else:
                    prev = temp
                    temp = temp.right
                    turn = "right"

        # kasus 0 children
        if (temp.left == None) and (temp.right == None):
            if turn == "left":
                prev.left = None
            elif turn == "right":
                prev.right = None
            del temp
            return

        # kasus 1 child, di kiri
        elif (temp.left != None) and (temp.right == None):
            if turn == "left":
                prev.left = temp.left
            elif turn == "right":
                prev.right = temp.left
```

```
                    del temp
                    return

                # kasus 1 child, di kanan
                elif (temp.left == None) and (temp.right != None):
                    if turn == "left":
                        prev.left = temp.right
                    elif turn == "right":
                        prev.right = temp.right
                    del temp
                    return

                # kasus 2 children
                elif inorder_pred: # metode inorder predecessor (left subtree)
                    inorder_left = []
                    self.get_inorder(current=temp.left, result=inorder_left)
                    replacement = inorder_left[-1] # elemen terakhir
                    self.delete(replacement, inorder_pred=inorder_pred)
                    temp.data = replacement
                    return
                else: # metode inorder successor (right subtree)
                    inorder_right = []
                    self.get_inorder(current=temp.right, result=inorder_right)
                    replacement = inorder_right[0]
                    self.delete(replacement, inorder_pred=inorder_pred)
                    temp.data = replacement
                    return
```

```
linkedbst = LinkedBST()
```

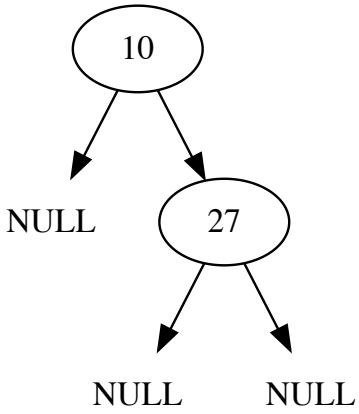
```
linkedbst.insert(10)
```

```
display(linkedbst.get_digraph_simple())
```



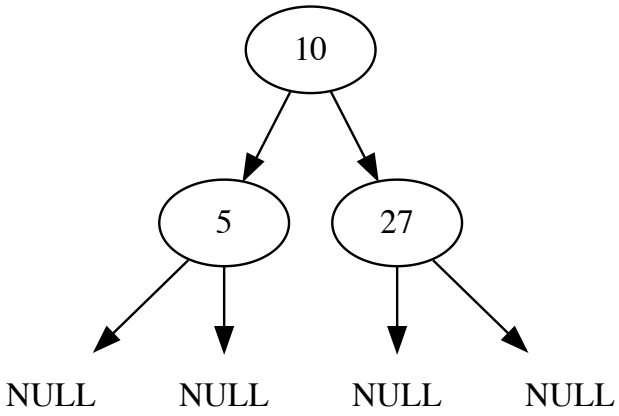

```
linkedbst.insert(27)
```

```
display(linkedbst.get_digraph_simple())
```



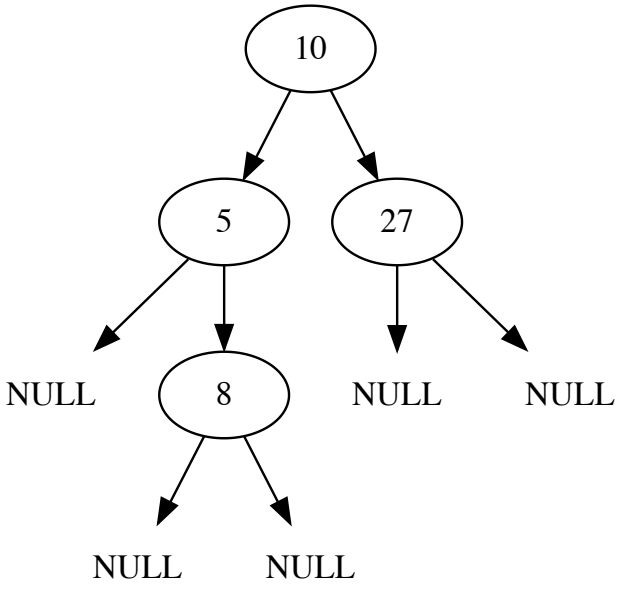

```
linkedbst.insert(5)
```

```
display(linkedbst.get_digraph_simple())
```




```
linkedbst.insert(8)
```

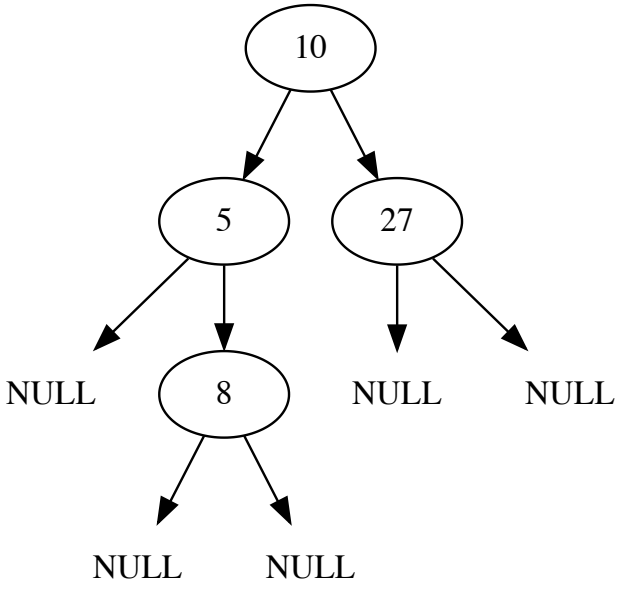
```
display(linkedbst.get_digraph_simple())
```




```
        linkedbst.insert(8)
```

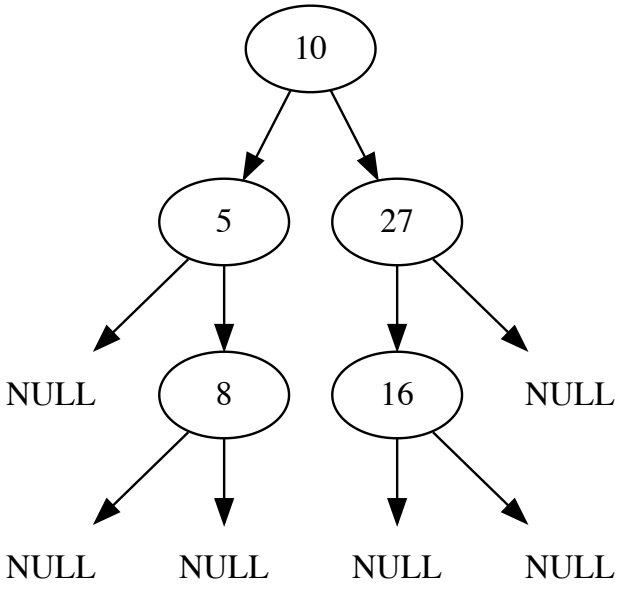
```
Error insert: data sudah ada di BST, yaitu 8
```

```
        display(linkedbst.get_digraph_simple())
```



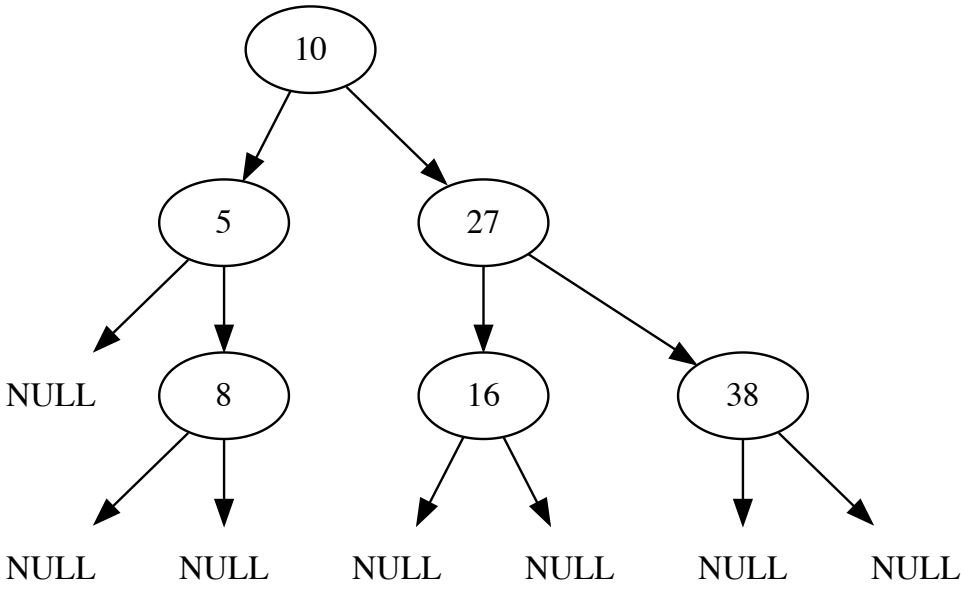

```
        linkedbst.insert(16)
```

```
        display(linkedbst.get_digraph_simple())
```



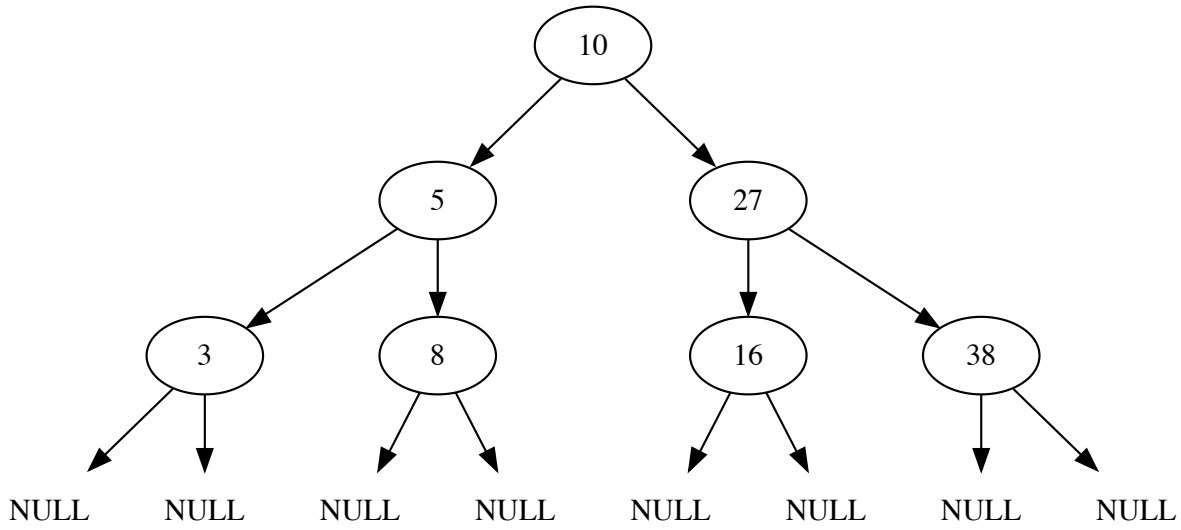

```
        linkedbst.insert(38)
```

```
        display(linkedbst.get_digraph_simple())
```



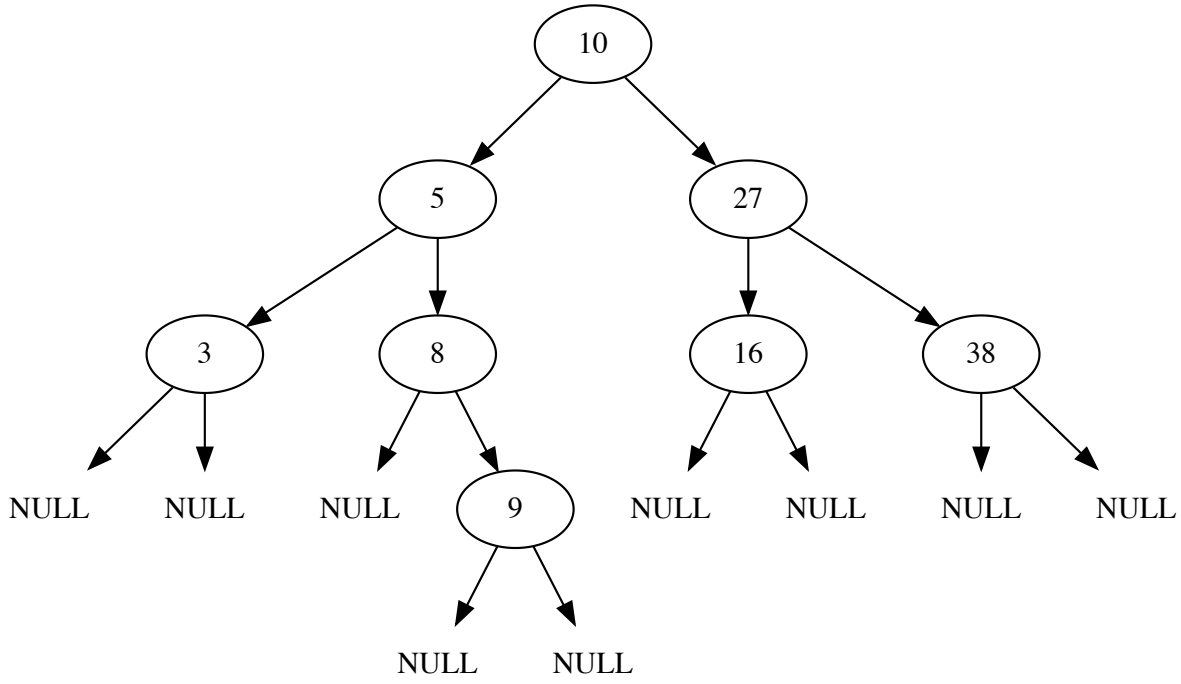

```
        linkedbst.insert(3)
```

```
        display(linkedbst.get_digraph_simple())
```




```
        linkedbst.insert(9)
```

```
        display(linkedbst.get_digraph_simple())
```




```
        linkedbst.get_preorder()
```

```
[10, 5, 3, 8, 9, 27, 16, 38]
```

```
        linkedbst.get_inorder()
```

```
[3, 5, 8, 9, 10, 16, 27, 38]
```

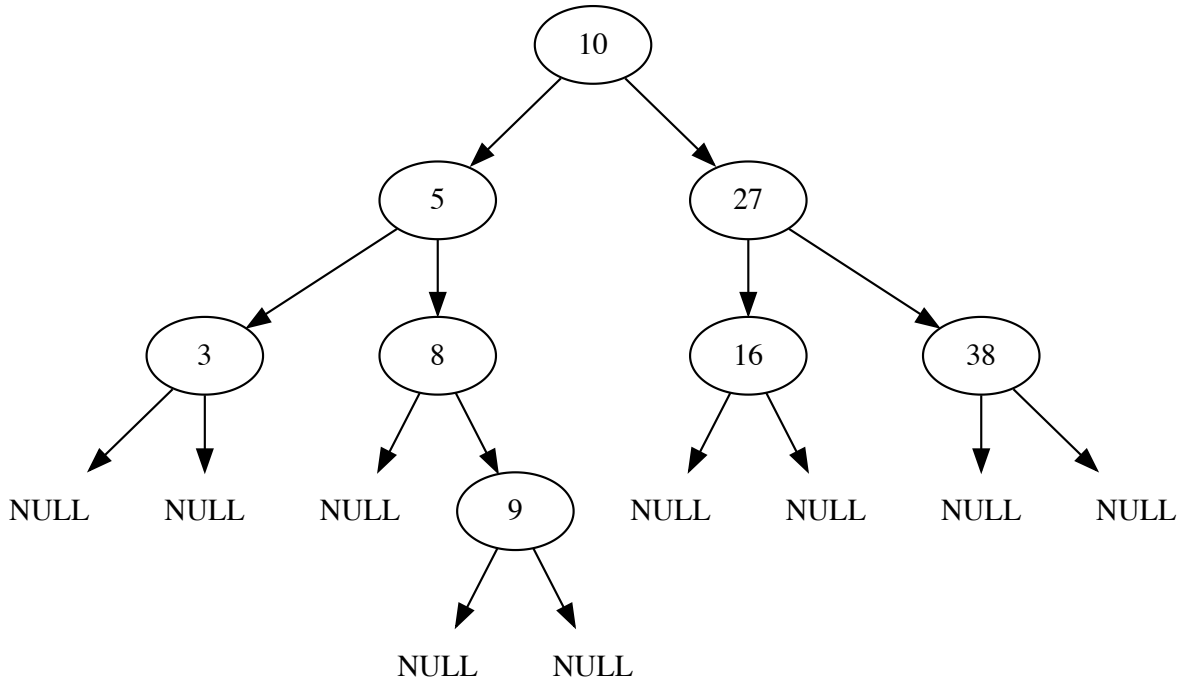
```
        linkedbst.get_postorder()
```

```
[3, 9, 8, 5, 16, 38, 27, 10]
```

```
        linkedbst.delete(50)
```

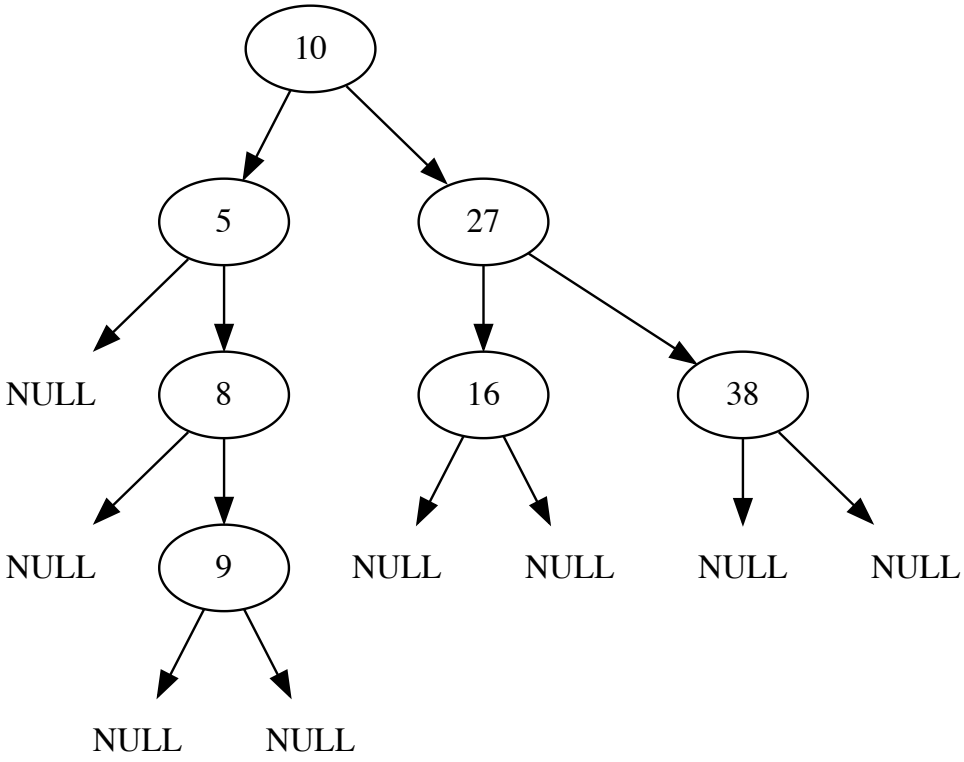
```
Error delete: tidak ditemukan data yang bernilai 50
```

```
        display(linkedbst.get_digraph_simple())
```



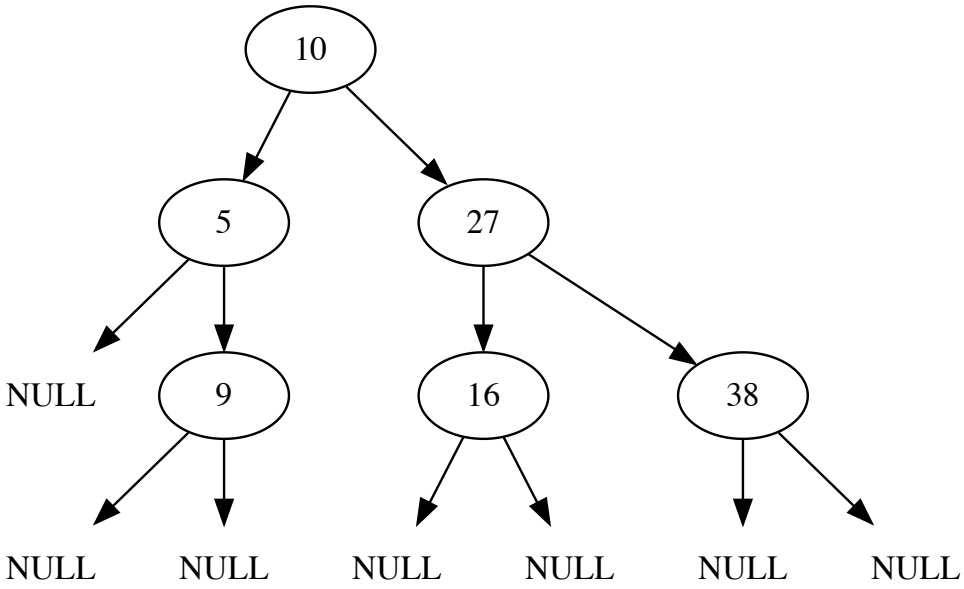

```
        linkedbst.delete(3)
```

```
        display(linkedbst.get_digraph_simple())
```



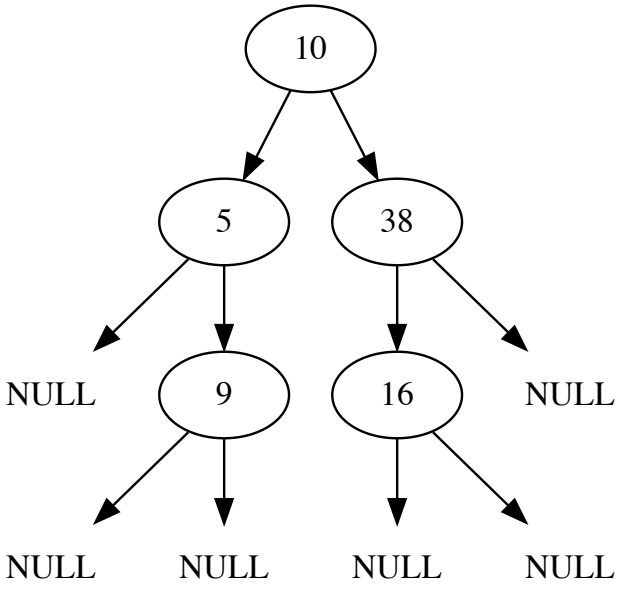

```
        linkedbst.delete(8)
```

```
        display(linkedbst.get_digraph_simple())
```



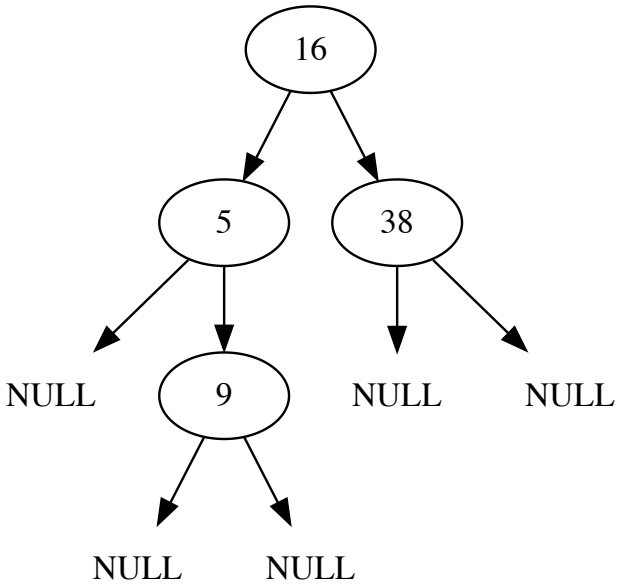

```
        linkedbst.delete(27)
```

```
        display(linkedbst.get_digraph_simple())
```



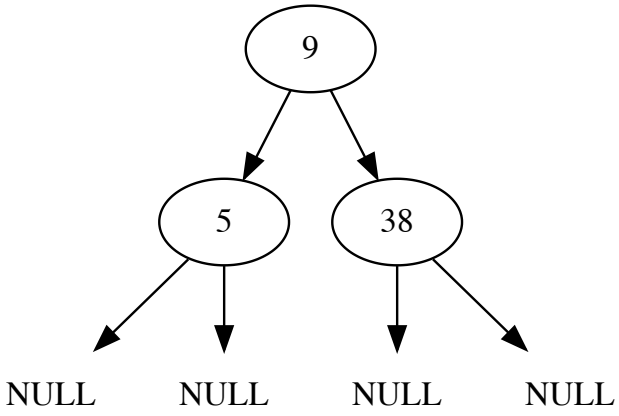

```
linkedbst.delete(10)
```

```
display(linkedbst.get_digraph_simple())
```




```
linkedbst.delete(16, inorder_pred=True)
```

```
display(linkedbst.get_digraph_simple())
```



(TODO) (Pengayaan) `LinkedBintree` dari *preorder*, *inorder*, dan/atau *postorder*

Kita akan membuat `LinkedBintree` saja, karena *height* dari *tree* yang akan dibentuk tidak bisa ditentukan sebelum *tree* selesai terbentuk, sedangkan pembuatan `ArrayBintree` melibatkan penentuan *height* di awal-awal sebelum *tree* dibentuk.

Jika diberikan *preorder* dengan *inorder*, atau *postorder* dengan *inorder*, maka hanya ada satu *binary tree* yang mungkin.

Namun, apabila diberikan *preorder* dengan *postorder*, maka *binary tree* yang dibentuk belum tentu unik. Meskipun demikian, apabila ditambahkan syarat bahwa *binary tree* yang dibentuk harus bersifat *complete*, maka *binary tree* yang dibentuk menjadi unik.

Oleh karena itu, untuk kasus diberikan *preorder* dengan *postorder*, ada algoritma biasa (tanpa syarat tersebut) dan algoritma dengan syarat tersebut.

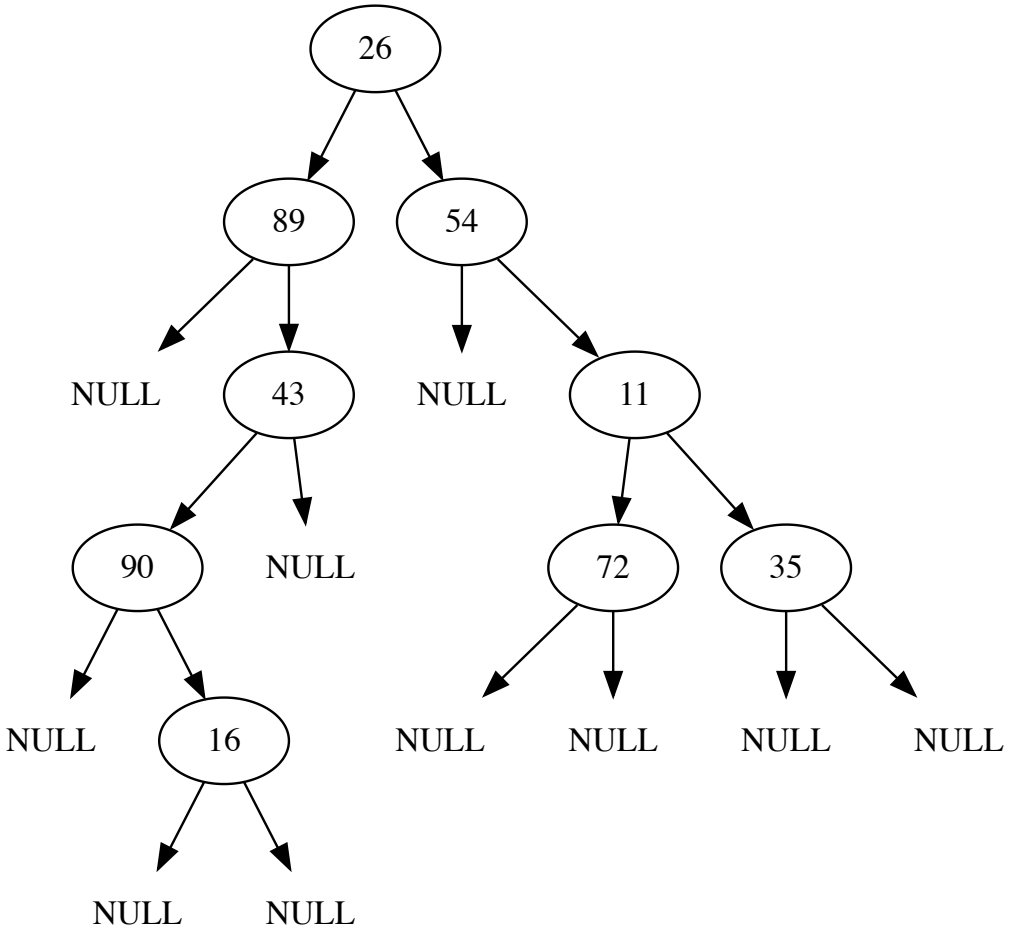
`LinkedBintree` dari *preorder* dan *inorder*

```
        def linkedbintree_from_preorder_inorder(
                preorder, inorder, is_starting_node=True
            ):

            # Nanti di paling bawah tree kalau inorder sudah kosong,
            # tidak perlu buat node lagi; langsung return None (NULL)
            if len(inorder) == 0:
                return None

            # 1. Di antara semua elemen inorder, mana yang paling kiri di preorder?
            # Simpan index inorder nya
            selesai = False
            preorder_idx = 0
            while (preorder_idx < len(preorder)) and (not selesai):
                # lihat tiap elemen preorder dari kiri ke kanan,
                elemen_preorder = preorder[preorder_idx]
                # dan untuk tiap elemen preorder, periksa satu-satu apakah sama dengan
                # salah satu elemen inorder
                inorder_idx = 0
                while (inorder_idx < len(inorder)) and (not selesai):
                    if inorder[inorder_idx] == elemen_preorder:
```

```
                selesai = True
            else:
                inorder_idx += 1
        preorder_idx += 1

    # 2. Buatlah node dengan data di index tersebut di inorder.
    # Kalau belum ada root (karena LinkedBintree belum dibentuk sama sekali),
    # buatlah objek LinkedBintree dengan rootnya adalah node tersebut
    current_root = BintreeNode(inorder[inorder_idx])
    if is_starting_node:
        result = LinkedBintree()
        result.root = current_root

    # 3. Pisah inorder menjadi dua bagian,
    # yaitu sebelah kiri dari elemen inorder_idx dan sebelah kanan darinya
    inorder_left = inorder[:inorder_idx]
    inorder_right = inorder[(inorder_idx+1):]

    current_root.left = linkedbintree_from_preorder_inorder(
        preorder, inorder_left, is_starting_node=False
    )
    current_root.right = linkedbintree_from_preorder_inorder(
        preorder, inorder_right, is_starting_node=False
    )

    if is_starting_node:
        return result
    else:
        return current_root
```

```
hasil_pre_in = linkedbintree_from_preorder_inorder(
    preorder=[26, 89, 43, 90, 16, 54, 11, 72, 35],
    inorder=[89, 90, 16, 43, 26, 54, 72, 11, 35]
)
```

```
display(hasil_pre_in.get_digraph_simple())
```



`LinkedBintree` dari *postorder* dan *inorder*

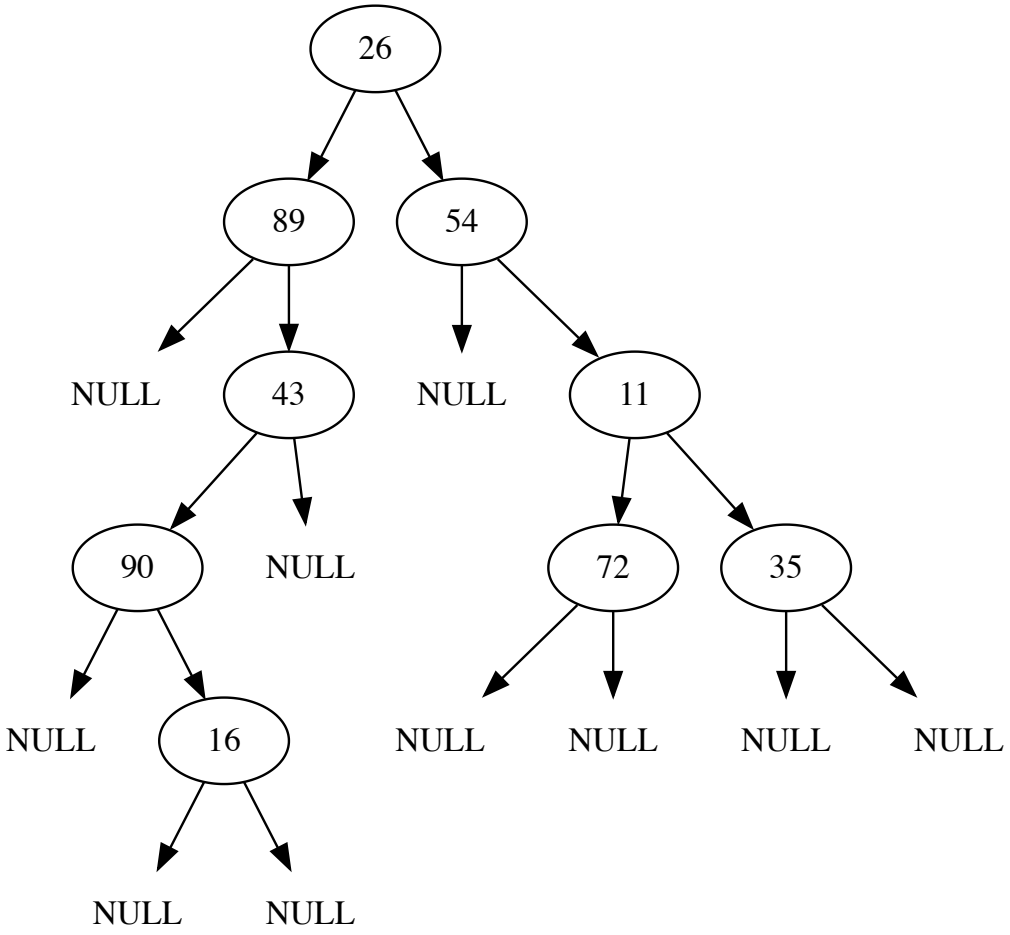
Algoritma ini hampir sama dengan algoritma membentuk *binary tree* dari *preorder* dan *inorder*. Bedanya, di algoritma ini, dicari elemen *inorder* yang paling ***kanan*** di *postorder*, daripada yang paling kiri di *preorder*.

```
        def linkedbintree_from_postorder_inorder(
                postorder, inorder, is_starting_node=True
            ):

            # Nanti di paling bawah tree kalau inorder sudah kosong,
            # tidak perlu buat node lagi; langsung return None (NULL)
            if len(inorder) == 0:
                return None

            # 1. Di antara semua elemen inorder, mana yang paling KANAN di postorder?
            # Simpan index inorder nya
            selesai = False
            postorder_idx = len(postorder)-1 # mulai dari paling kanan, daripada dari
            while (postorder_idx >= 0) and (not selesai):
                # lihat tiap elemen preorder DARI KANAN KE KIRI,
                elemen_postorder = postorder[postorder_idx]
                # dan untuk tiap elemen postorder, periksa satu-satu apakah sama denga
                # salah satu elemen inorder
                inorder_idx = 0
                while (inorder_idx < len(inorder)) and (not selesai):
                    if inorder[inorder_idx] == elemen_postorder:
```

```
                selesai = True
            else:
                inorder_idx += 1
        postorder_idx -= 1

    # 2. Buatlah node dengan data di index tersebut di inorder.
    # Kalau belum ada root (karena LinkedBintree belum dibentuk sama sekali),
    # buatlah objek LinkedBintree dengan rootnya adalah node tersebut
    current_root = BintreeNode(inorder[inorder_idx])
    if is_starting_node:
        result = LinkedBintree()
        result.root = current_root

    # 3. Pisah inorder menjadi dua bagian,
    # yaitu sebelah kiri dari elemen inorder_idx dan sebelah kanan darinya
    inorder_left = inorder[:inorder_idx]
    inorder_right = inorder[(inorder_idx+1):]

    current_root.left = linkedbintree_from_postorder_inorder(
        postorder, inorder_left, is_starting_node=False
    )
    current_root.right = linkedbintree_from_postorder_inorder(
        postorder, inorder_right, is_starting_node=False
    )

    if is_starting_node:
        return result
    else:
        return current_root
```

```
hasil_post_in = linkedbintree_from_postorder_inorder(
    postorder=[16, 90, 43, 89, 72, 35, 11, 54, 26],
    inorder=[89, 90, 16, 43, 26, 54, 72, 11, 35]
)
```

```
display(hasil_post_in.get_digraph_simple())
```



(TODO) `LinkedBintree` dari *preorder* dan *postorder* (cara biasa)

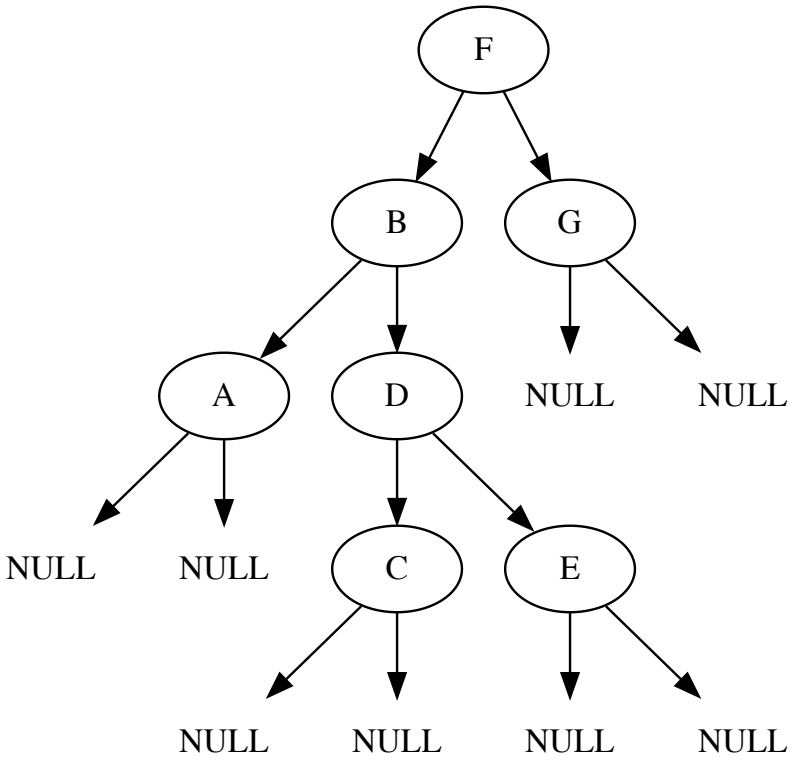
```
        def linkedbintree_from_preorder_postorder(
                preorder, postorder, is_starting_node=True
        ):

            if (not is_starting_node):
                if len(preorder) == 0 or len(postorder) == 0:
                    return None
                if len(preorder) == 1:
                    return BintreeNode(preorder[0])
                if len(postorder) == 1:
                    return BintreeNode(postorder[0])

            # 1. Buatlah node baru dengan datanya adalah preorder[0]
            # (atau sama saja elemen terakhir dari postorder).
            # Kalau belum ada root (karena LinkedBintree belum dibentuk sama sekali),
            # buatlah objek LinkedBintree dengan rootnya adalah node tersebut
            current_root = BintreeNode(preorder[0])
            if is_starting_node:
                result = LinkedBintree()
                result.root = current_root

            # 2. Tentukan list postorder untuk left subtree dan untuk right subtree:
            # 2a. Carilah letak preorder[1] di postorder, misal postorder_idx
            # 2b. Belah postorder menjadi dua, dengan postorder_idx masuk ke kiri,
```

```
        #     dan elemen terakhir postorder tidak masuk keduanya

        postorder_idx = 0
        while (postorder_idx < len(postorder) and
               postorder[postorder_idx] != preorder[1]):
            postorder_idx += 1

        # 0 <= indeks < (postorder_idx+1)
        postorder_left = postorder[ 0 : (postorder_idx+1) ]

        # (postorder_idx+1) <= indeks < elemen terakhir (indeks -1)
        postorder_right = postorder[ (postorder_idx+1) : -1 ]

        # 3. Tentukan list preorder untuk left subtree dan untuk right subtree:
        # 3a. Carilah letak postorder[-2] di preorder, misal preorder_idx
        # 3b. Belah preorder menjadi dua, dengan preorder_idx masuk ke kanan,
        #     dan elemen pertama preorder tidak masuk keduanya

        preorder_idx = 0
        while (preorder_idx < len(preorder) and
               preorder[preorder_idx] != postorder[-2]):
            preorder_idx += 1

        # 1 <= indeks < preorder_idx
        preorder_left = preorder[ 1 : preorder_idx ]

        # preorder_idx <= indeks
        preorder_right = preorder[ preorder_idx : ]

        print("preorder_left", len(preorder_left))
        print("preorder_right", len(preorder_right))
        print("postorder_left", len(postorder_left))
        print("postorder_right", len(postorder_right))

        # 4. Langkah rekursif: melakukan langkah yang sama di left subtree dan
        # right subtree, hasilnya disambung ke current_root

        current_root.left = linkedbintree_from_preorder_postorder(
            preorder=preorder_left, postorder=postorder_left,
            is_starting_node=False
        )
        current_root.right = linkedbintree_from_preorder_postorder(
            preorder=preorder_right, postorder=postorder_right,
            is_starting_node=False
        )

        if is_starting_node:
            return result
        else:
            return current_root
```

```
    test_pre_post = linkedbintree_from_preorder_postorder(
        preorder=["F", "B", "A", "D", "C", "E", "G", "I", "H"],
        postorder=["A", "C", "E", "D", "B", "H", "I", "G", "F"]
    )
```

```
preorder_left 5
preorder_right 3
postorder_left 5
postorder_right 3
preorder_left 1
preorder_right 3
postorder_left 1
postorder_right 3
preorder_left 1
preorder_right 1
postorder_left 1
postorder_right 1
preorder_left 0
preorder_right 2
postorder_left 2
postorder_right 0
```

```
        display(test_pre_post.get_digraph_simple())
```



(TODO) `LinkedBintree` dari *preorder* dan *postorder* (cara dijamin *complete*)

(TODO) (Pengayaan) `LinkedBST` dari *preorder* atau *postorder*

(TODO) `LinkedBST` dari *preorder*

(TODO) `LinkedBST` dari *postorder*

(TODO) (Pengayaan) m-*ary tree*