Bab 4. Stack

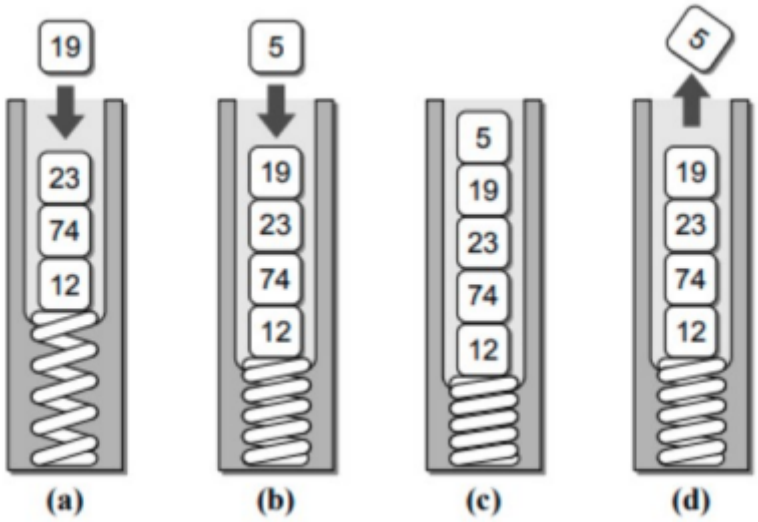
OBJEKTIF :

1. Mahasiswa mampu memahami Stack, Implementasi Stack Menggunakan List, Implementasikan Stack Dengan Linked List dan Aplikasi Stack
2. Mahasiswa mampu mengimplementasikan konsep yang ada pada topik Linked List menggunakan bahasa pemrograman Python.

4.1 Stack

Stack dalam Bahasa Indonesia diterjemahkan sebagai "tumpukan" adalah struktur data yang meniru prinsip kerja dari wadah tumpukan piring yang biasanya terdapat dalam kafetaria. Gambar berikut adalah contoh wadah piring ini:

Penambahan dan pengambilan piring dalam wadah piring menerapkan prinsip `last-in first-out` (LIFO). `Last-in first-out` berarti yang masuk terakhir adalah yang pertama kali keluar. Gambar berikut mengilustrasikan proses penambahan data dan pengambilan data pada stack:



Pada stack data yang ditambahkan akan diletakkan di atas data yang diletakkan sebelumnya, sehingga data yang terakhir ditambahkan akan berada di paling atas stack. Data yang berada di paling atas diistilahkan sebagai ***top*** dan operasi penambahan data ke stack ini diistilahkan sebagai ***push*** yang berarti dorong, karena ketika kita menambahkan data kita mendorong data ini ke dalam stack. Pada gambar di atas, Gambar (a) mengilustrasikan operasi *push* nilai 19 ke dalam stack. Gambar (b) mengilustrasikan proses *push* nilai 5 ke stack. Gambar (c) memperlihatkan stack setelah 19 dan 5 ditambahkan. Nilai 5 adalah *top* dari stack.

Kita hanya bisa menghapus data dalam stack satu per satu dimulai dari data yang berada di *top*. Operasi penghapusan data dalam stack ini disebut sebagai ***pop*** yang berarti mencabut, karena ketika kita menghapus data kita seperti mencabut data paling atas dari stack. Pada Gambar (d) di atas, operasi *pop* menghapus data *top* yaitu nilai 5, dari stack.

ADT Stack

Definisi ADT Stack
Stack adalah struktur data yang menyimpan sebuah koleksi *linear* dari data. Penambahan dan penghapusan data pada stack terbatas pada urutan *last-in first-out*. Operasi-operasi yang dimiliki oleh stack adalah sebagai berikut:

- `Stack()` : Membuat stack kosong.
- `isEmpty()` : Mengembalikan nilai Boolean yang menandakan apakah stack kosong.
- `len()` : Mengembalikan banyaknya data pada stack.
- `pop()` : Menghapus dan mengembalikan data yang berada di *top* (paling atas) dari stack. Data yang berada kedua paling atas kemudian menjadi data *top* (paling atas).
- `peek()` : Mengembalikan referensi ke data paling atas pada stack tanpa menghapusnya. Jika stack kosong, maka operasi ini menghasilkan *error*. Operasi *peek* tidak memodifikasi isi stack.
- `push(data)` : Menambahkan data ke *top* (bagian paling atas) dari stack.

Tabel berikut mencontohkan penggunaan *method-method* pada stack beserta nilai kembali dan perubahan dari isi stack:

Operasi	Nilai Kembali	Isi Stack (kanan adalah *top*)
`s = Stack()`	-	[]
`s.push(5)`	-	[5]
`s.push(3)`	-	[5, 3]
`len(s)`	2	[5, 3]
`s.pop()`	3	[5]
`s.isEmpty()`	`False`	[5]
`s.pop()`	5	[]
`s.isEmpty()`	`True`	[]
`s.pop()`	Error	[]
`s.push(7)`	-	[7]
`s.push(9)`	-	[7, 9]
`s.peek()`	9	[7, 9]
`s.push(4)`	-	[7, 9, 4]
`len(s)`	3	[7, 9, 4]
`s.pop()`	4	[7, 9]
`s.push(6)`	-	[7, 9, 6]
`s.push(8)`	-	[7, 9, 6, 8]
`s.pop()`	8	[7, 9, 6]

ADT Stack dapat diimplementasikan dalam berbagai cara. Dua cara yang paling umum adalah dengan menggunakan `list` dan *linked* list. Kita akan membahas kedua implementasi ini.

4.2 Implementasi Stack Menggunakan `list`

Implementasi ADT Stack dengan `list` adalah implementasi yang paling mudah. Hal yang harus kita tentukan adalah ujung dari list mana yang kita gunakan untuk merepresentasikan *top* dari stack. Untuk mendapatkan pengurutan yang paling efisien, kita menentukan elemen terakhir dari list merepresentasikan *top* dari stack dan elemen pertama merepresentasikan data paling bawah dari stack.

Implementasi Stack kita mulai dengan mendefinisikan *class* `Stack`:

```
class Stack:

    # ... Implementasi method-method
```

A. Constructor `Stack()`

Class Stack hanya perlu mempunyai satu *field* yaitu *field* yang mereferensikan sebuah `list` tempat data stack disimpan.

```
def __init__(self):
    self._data = list()
```

B. Method `isEmpty()`

Method `isEmpty()` mengembalikan `True` jika stack kosong dan mengembalikan `False` jika stack tidak kosong. Kita hanya perlu mengembalikan sebuah ekspresi yang menguji apakah `list` yang digunakan untuk menyimpan data stack mempunyai panjang 0.

```
def isEmpty(self):
    return len(self._data) == 0
```

C. Method `len()`

Method `len()` hanya perlu mengembalikan panjang dari `list` tempat menyimpan data stack.

```
def __len__(self):
    return len(self._data)
```

D. Method `peek()`

Method `peek()` mengembalikan nilai dari data di `top` (bagian paling atas) dari stack. Karena kita telah menentukan elemen terakhir dari `list` tempat menyimpan data stack adalah bagian *top* dari stack, maka kita hanya perlu mengembalikan nilai elemen terakhir dari `list` ini. Kita juga perlu menuliskan kondisional ketika *method* ini dipanggil pada stack kosong. Jika ini terjadi, kita meng-*raise* sebuah eksepsi generik dengan pesan stack kosong. Untuk menguji apakah stack kosong atau tidak kita hanya perlu memanggil method `isEmpty()` yang sebelumnya kita tulis.

Sehingga, definisi *method* `peek()` dapat dituliskan seperti berikut:

```
def peek(self):
    if self.isEmpty():
        raise Exception('Stack kosong. Tidak ada data top.')
    else:
        return self._data[-1]
```

Pada *method* `peek()` kita mengembalikan element terakhir dari *field* `_data` yang berupa `list` menggunakan operasi pengindeksan terhadap `list` tersebut dengan indeks -1.

E. Method `pop()`

Method `pop()` menghapus data *top* dalam stack dan mengembalikan nilai data yang di-*pop*. Untuk melakukan ini, kita hanya perlu memanggil method `pop()` pada `list` yang menyimpan data stack. Seperti juga pada method `peek()` kita harus menuliskan kondisional ketika stack kosong. Jika *method* `pop()` ini dipanggil pada stack kosong, maka kita meng-*raise* sebuah eksepsi generik dengan pesan stack kosong.

Berikut adalah definisi *method* `pop()`:

```
def pop(self):
    if self.isEmpty():
        raise Exception('Stack kosong. Tidak ada data yang dapat di-pop.')
    else:
        return self._data.pop()
```

F. Method `push(data)`

Method `push(data)` menambahkan data pada *top* dari stack. Dan karena elemen terakhir dari `list` yang menyimpan data stack adalah *top* dari stack, maka kita hanya perlu memanggil *method* `append(data)` pada `list` tersebut. Sehingga, *method* `push(data)` dapat dituliskan seperti berikut:

```
def push(self, data):
    self._data.append(data)
```

Kode Lengkap Class `Stack`

Berikut adalah kode lengkap dari class `Stack` yang kita simpan dalam *module* bernama `liststack`:

Module *`liststack.py`*

```
# Implementasi ADT Stack menggunakan list
class Stack:
    # Constructor dari class Stack.
    # Field 1 (_data): sebuah list untuk menyimpan data stack.
    def __init__(self):
        self._data = list()

    # Method isEmpty() mengembalikan True jika stack kosong atau False
    # jika stack tidak kosong.
    def isEmpty(self):
        return len(self) == 0

    # Method len() mengembalikan banyaknya data dalam stack.
    def __len__(self):
        return len(self._data)

    # Method peek(), mengembalikan data top dari stack tanpa menghapusnya.
    # Jika stack kosong, meng-raise eksepsi generik.
    def peek(self):
        if self.isEmpty():
            raise Exception('Stack kosong. Tidak ada data top.')
        else:
            return self._data[-1]

    # Method pop() menghapus dan mengembalikan data top pada stack.
    # Jika stack kosong, meng-raise Exception generik.
    def pop(self):
        if self.isEmpty():
            raise Exception('Stack kosong. Tidak ada data yang dapat di-pop.')
        else:
            return self._data.pop()

    # Method push(data) memasukkan data ke top dari stack
```

```
    def push(self, data):
        self._data.append(data)
```

Untuk menguji impelementasi ADT Stack yang telah kita tulis, kita dapat menuliskan program berikut:

```
from liststack import Stack

def main():
    # Prompt untuk meminta input dari pengguna
    PROMPT = 'Masukkan nilai integer (nilai negatif untuk mengakhiri): '

    # Buat stack
    myStack = Stack()

    # Uji apakah stack kosong
    if myStack.isEmpty:
        print('Stack kosong.')
    else:
        print('Stack tidak kosong.')

    # Prompt pengguna untuk memasukkan data ke stack
    nilai = int(input(PROMPT))
    while nilai >= 0:
        myStack.push(nilai)
        nilai = int(input(PROMPT))

    # Tampilkan panjang stack
    print('Panjang stack: ', end='')
    print(len(myStack))

    # Tampilkan top dari stack
    print('Top dari stack: ', end='')
    print(myStack.peek())

    # Cetak isi stack
    print('Isi stack:')
    while not myStack.isEmpty():
        nilai = myStack.pop()
        print(nilai)

main()
```

Contoh *output* program yang menguji stack yang kita tulis adalah sebagai berikut:

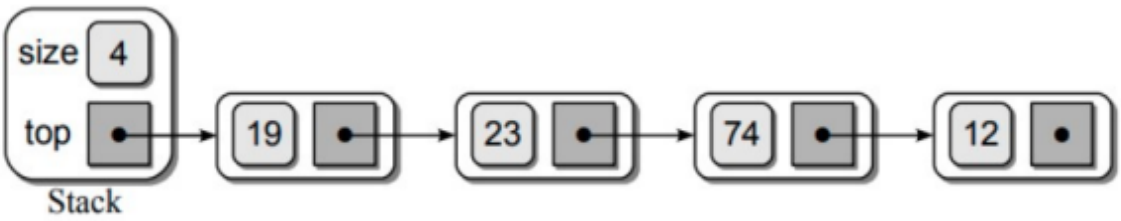
```
Stack kosong.
Masukkan nilai integer (nilai negatif untuk mengakhiri): 78
Masukkan nilai integer (nilai negatif untuk mengakhiri): 89
Masukkan nilai integer (nilai negatif untuk mengakhiri): 43
Masukkan nilai integer (nilai negatif untuk mengakhiri): 56
Masukkan nilai integer (nilai negatif untuk mengakhiri): 78
Masukkan nilai integer (nilai negatif untuk mengakhiri): 43
Masukkan nilai integer (nilai negatif untuk mengakhiri): 90
Masukkan nilai integer (nilai negatif untuk mengakhiri): -1
```

```
Panjang stack: 7
Top dari stack: 90
Isi stack:
90
43
78
56
43
89
78
```

4.3 Mengimplemetasikan Stack dengan Linked List

Implementasi ADT Stack menggunakan `list` tidak efisien untuk stack yang melakukan banyak operasi *push* dan *pop*. Ini karena operasi `append()` dan operasi `pop()` dari `list` membutuhkan realokasi ruang memori untuk mengakomodasi penambahan atau pengurangan elemen pada `list`. Implementasi ADT Stack menggunakan linked list memperbaiki kekurangan ini.

Untuk menggunakan linked list, kita harus menentukan bagaimana kita merepresentasikan struktur stack. Dengan `list`, merepresentasikan *top* dari stack dengan elemen terakhir dari `list` adalah cara yang paling efisien. Namun, dengan linked list, merepresentasikan *top* dengan ujung awal dari linked list adalah cara yang paling efisien. Ini karena, pada linked list, penambahan dan penghapusan node di awal linked list dapat dilakukan dengan mudah. Ilustrasi implementasi stack dengan linked list dapat dilihat pada gambar berikut:



Implementasi Stack kita mulai dengan mendefinisikan class `Stack`:

```
class Stack:
    # ... Implementasi stack menggunakan linked list.
```

A. Class `StackNode`

Kita membutuhkan sebuah class terpisah untuk merepresentasikan node-node dari linked list. Kita menamakan class ini dengan nama `StackNode` dan menuliskan definisi class tersebut seperti berikut:

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

B. Constructor `Stack()`

Class `Stack` dengan linked list membutuhkan dua *field*: *field* pertama untuk variabel referensi ke *head* dari linked list yang kita gunakan sebagai *top* dari stack, dan *field* kedua adalah variabel untuk menyimpan ukuran dari stack.

Constructor `Stack()` dapat dituliskan seperti berikut:

```
def __init__(self):
    self._top = None
    self._size = 0
```

C. Method `isEmpty()`

Method `isEmpty()` mengembalikan `True` jika stack kosong dan mengembalikan `False` jika stack tidak kosong. Ketika stack kosong, *field* `_top` dari *object* stack akan mereferensikan nol dan *field* `_size` dari *object* stack akan bernilai nol. Kita dapat menguji salah satu dari dua kondisi tersebut untuk mengimplementasikan method `isEmpty()`. Pada definisi method `isEmpty()` di bawah kita mengembalikan hasil pengujian apakah *field* `_top` dari stack mereferensikan `None`:

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

D. Method `len()`

Method `len()` dapat kita impelementasikan untuk mengembalikan nilai *field* `_size`.

```
def __len__(self):
    return self._size
```

E. Method `peek()`

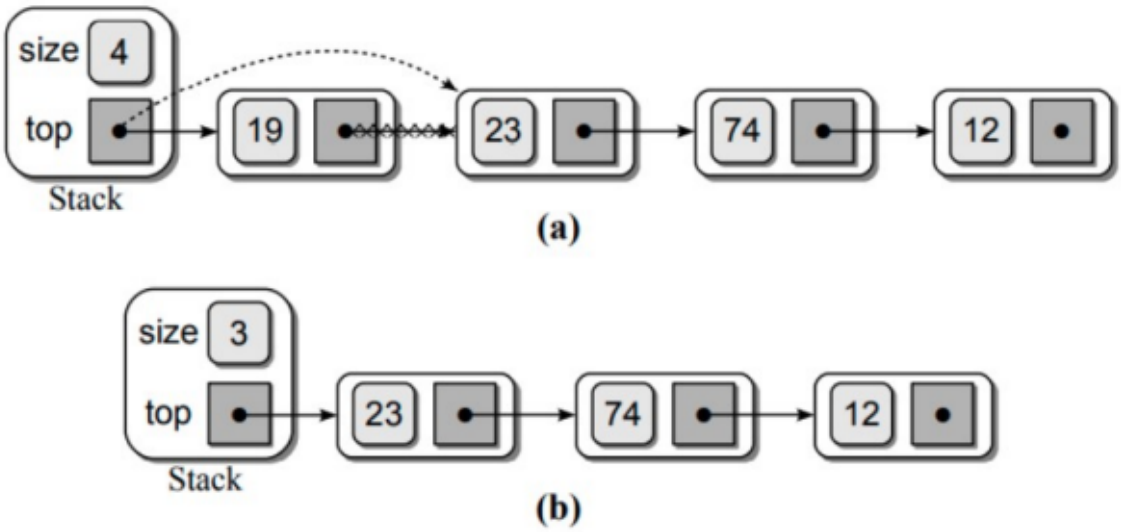
Method `peek()` dapat dituliskan dengan mengembalikan *field* data dari node pada *top*. Sebuah kondisional juga harus dituliskan untuk kondisi stack kosong.

Berikut adalah implementasi method `peek()`:

```
def peek(self):
    if self.isEmpty():
        raise Exception('Stack kosong. Tidak ada data top.')
    else:
        return self._top.data
```

F. Method `pop()`

Method `pop()` menghapus node pertama dalam linked list dan mengembalikan nilai data yang dihapus. Operasi pop ini diilustrasikan pada gambar berikut:



Gambar (a) menghapus node pertama (*top* dari stack) dengan menghapus *link* dari node pertama dan menugaskan variabel referensi *top* ke node kedua. Gambar (b) adalah hasil dari linked list setelah operasi *pop*.

Sama seperti pada method `peek()`, kita juga perlu menuliskan kondisional ketika stack kosong. Sehingga method `pop()` dapat dituliskan seperti berikut:

```
def pop(self):
    if self.isEmpty():
        raise Exception('Stack kosong. Tidak ada data yang dapat di-pop')
    else:
        dataDihapus = self._top.data
        self._top = self._top.next
        self._size = -1
        return dataDihapus
```

G. Method `push(data)`

Method `push(data)` menambahkan data ke *top* dari stack. Kita dapat melakukannya dengan membuat node baru dan menyisipkan node tersebut ke awal dari linked list.

Berikut adalah implementasi method `push(data)`:

```
def push(self, data):
    newNode = _StackNode(data)
    newNode.next = self._top
    self._top = newNode
    self._size += 1
```

Kode Lengkap Implementasi ADT Stack Menggunakan Linked List

Kode lengkap implementasi ADT Stack menggunakan linked list adalah sebagai berikut:

Module *`linkedliststack.py`*

```
# Implementasi ADT Stack menggunakan linked list
class Stack:
    # Constructor dari class Stack.
```

```
    # Field 1 (_top): variabel referensi ke top node.
    # Field 2 (_size): variabel yang menyimpan banyaknya node dalam stack.
    def __init__(self):
        self._top = None
        self._size = 0

    # Method isEmpty() mengembalikan True jika stack kosong atau False
    # jika stack tidak kosong.
    def isEmpty(self):
        return self._size == 0

    # Method len() mengembalikan banyaknya data dalam stack.
    def __len__(self):
        return self._size

    # Method peek(), mengembalikan data top dari stack tanpa menghapusnya.
    # Jika stack kosong, meng-raise eksepsi generik.
    def peek(self):
        if self.isEmpty():
            raise Exception('Stack kosong. Tidak ada data top.')
        else:
            return self._top.data

    # Method pop() menghapus dan mengembalikan data top pada stack.
    # Jika stack kosong, meng-raise Exception generik.
    def pop(self):
        if self.isEmpty():
            raise Exception('Stack kosong. Tidak ada data yang dapat di-pop.')
        else:
            dataDihapus = self._top.data
            self._top = self._top.next
            self._size -= 1
            return dataDihapus

    # Method push(data) memasukkan data ke top dari stack
    def push(self, data):
        newNode = _StackNode(data)
        newNode.next = self._top
        self._top = newNode
        self._size += 1

# Class Node sebagai class untuk penyimpanan data
class _StackNode():
    def __init__(self, data):
        self.data = data
        self.next = None
```

Kita dapat menguji implementasi stack dengan linked list di atas menggunakan program berikut:

```
from linkedliststack import Stack

def main():
    # Prompt untuk meminta input dari pengguna
    PROMPT = 'Masukkan nilai integer (nilai negatif untuk mengakhiri): '

    # Buat stack
    myStack = Stack()
```

```
    # Uji apakah stack kosong
    if myStack.isEmpty:
        print('Stack kosong.')
    else:
        print('Stack tidak kosong.')

    # Prompt pengguna untuk memasukkan data ke stack
    nilai = int(input(PROMPT))
    while nilai >= 0:
        myStack.push(nilai)
        nilai = int(input(PROMPT))

    # Tampilkan panjang stack
    print('Panjang stack: ', end='')
    print(len(myStack))

    # Tampilkan top dari stack
    print('Top dari stack: ', end='')
    print(myStack.peek())

    # Cetak isi stack
    print('Isi stack:')
    while not myStack.isEmpty():
        nilai = myStack.pop()
        print(nilai)

main()
```

Output dari program di atas:

```
Masukkan nilai integer (nilai negatif untuk mengakhiri): 94
Masukkan nilai integer (nilai negatif untuk mengakhiri): 56
Masukkan nilai integer (nilai negatif untuk mengakhiri): 98
Masukkan nilai integer (nilai negatif untuk mengakhiri): 56
Masukkan nilai integer (nilai negatif untuk mengakhiri): 88
Masukkan nilai integer (nilai negatif untuk mengakhiri): 34
Masukkan nilai integer (nilai negatif untuk mengakhiri): 65
Masukkan nilai integer (nilai negatif untuk mengakhiri): -1
Panjang stack: 7
Top dari stack: 65
Isi stack:
65
34
88
56
98
56
94
```

4.4 Aplikasi Stack

Salah satu penerapan struktur data stack adalah untuk menuliskan program yang mencocokkan pasangan simbol pengelompok pada ekspresi aritmatika. Misalkan ekspresi aritmatika dapat terdiri dari simbol-simbol pengelompokkan seperti:

- Tanda kurung: "(" dan ")"
- Tanda kurung kurawal: "{" dan "}"
- Tanda kurung kotak: "[" dan "]"

Setiap simbol pembuka harus dipasangkan dengan simbol penutupnya. Sebagai contoh, kurung kotak pembuka, "[", harus dipasangkan dengan kurung kotak penutup, "]", seperti dalam ekspresi: `[(5 + x) - (y + z)]`. Contoh-contoh berikut memberikan memperlihatkan penulisan simbol-simbol pengelompokkan yang benar dan tidak benar:

- `()(()){([()])}`: Benar
- `(((()(())){([()])}))`: Benar
- `)(()){([()])}`: Salah
- `({[])}`: Salah
- `(`: Salah

Sebuah ekspresi dianggap seimbang jika simbol-simbol pengelompokkan pembuka mempunyai pasangan simbol penutupnya dan dituliskan dalam urutan yang benar.

Langkah-langkah untuk menentukan apakah sebuah ekspresi dituliskan dengan simbol pengelompokkan yang seimbang:

- *Scan* karakter per karakter dari ekspresi yang diberikan.
- Jika karakter adalah simbol pembuka "{", "[", atau "(", *push* karakter tersebut ke stack.
- Jika karakter adalah simbol penutup "}", "]", atau ")", *pop* karakter dalam stack. Jika karakter yang di-*pop* dari stack bukanlah simbol pembuka yang cocok dari simbol penutup yang di-*scan*, maka hentikan proses *scan* dan simpulkan ekspresi tidak seimbang.
- Setelah men-*scan* semua karakter dalam ekspresi, jika masih terdapat simbol-simbol pengelompok dalam stack (jika stack tidak kosong), maka ekspresi tidak seimbang. Dan jika stack kosong, maka ekspresi seimbang.

Misalkan, ekspresi:

```
{[(5 + 6) * 7] / 2}
```

Tabel berikut memperlihatkan proses *scan* dari ekspresi di atas:

Karakter Di-scan	Operasi	Isi Stack
`{`	`push`	`{`
`[`	`push`	`{ [`
`(`	`push`	`{ [ (`
`5`	-	`{ [ (`
`+`	-	`{ [ (`
`6`	-	`{ [ (`
`)`	`pop` dan cocokkan	`{ [`
`*`	-	`{ [`
`7`	-	`{ [`
`]`	`pop` dan cocokkan	`{`
`/`	-	`{`
`2`	-	`{`
`}`	`pop` dan cocokkan	kosong

Fungsi berikut menerima sebuah ekspresi dan mengembalikan `True` jika ekspresi yang diberikan mempunyai simbol pengelompok seimbang dan mengembalikan `False` jika ekspresi yang diberikan tidak mempunyai simbol pengelompok yang seimbang.

```
from linkedliststack import Stack

def isMatched(expr):
    s = Stack()
    for char in expr:
        if char in '{[(':
            s.push(char)
        elif char in '}])':
            if s.isEmpty():
                return False
            else:
                left = s.pop()
                if (char == '}' and left != '{') or \
                   (char == ']' and left != '[') or \
                   (char == ')' and left != '('):
                   return False
    return s.isEmpty()
```

Program berikut menggunakan fungsi di atas untuk menentukan apakah ekspresi yang dimasukkan pengguna seimbang:

```
def main():
    expr = input("Ketikkan sebuah ekspresi: ")
    if isMatched(expr) == True:
        print("Ekspresi yang Anda masukkan BENAR.")
    else:
        print("Ekspresi yang Anda masukkan SALAH.")

main()
```

Output program:

```
Ketikkan sebuah ekspresi: (a + b
Ekspresi yang Anda masukkan SALAH.
```

Berdasarkan *output* diatas, ekspresi yang diberikan tidak mempunyai simbol pengelompok yang seimbang. Maka hasilnya akan bernilai salah

REFERENSI

[1] Necaise, Rance D. 2011. Data structures and algorithms using Python .