STACKS

Algoritma dan struktur data

Syahrul Akbar Ramdhani (1123094000027)

Daftar Isi

STACKS

Stack adalah kumpulan objek yang mengikuti prinsip last-in, first-out (LIFO). Pengguna dapat memasukkan objek ke dalam stack kapan saja, tetapi hanya dapat mengakses atau menghapus objek yang terakhir dimasukkan yang masih tersisa (di bagian yang disebut “top" dari stack).
Contoh dari stack adalah tempat penyimpanan permen yang mekanismenya mengikuti prinsip LIFO.

Stack adalah struktur data fundamental yang banyak digunakan dalam berbagai aplikasi,seperti:
Contoh 6.1: Browser web menyimpan alamat situs yang baru dikunjungi dalam stack. Setiap kali pengguna mengunjungi situs baru, alamat situs tersebut akan “pushed" ke dalam tumpukan alamat. Browser kemudian memungkinkan pengguna untuk “pop" ke situs yang dikunjungi sebelumnya menggunakan tombol "back".
Contoh 6.2: Editor teks biasanya menyediakan mekanisme "undo" yang membatalkan operasi pengeditan terbaru dan mengembalikan ke keadaan dokumen sebelumnya. Operasi undo ini dapat dilakukan dengan menyimpan perubahan teks dalam stack.

6.1.1 Stack untuk Tipe Data Abstrak

Stack adalah struktur data yang paling sederhana namun termasuk yang paling penting. Stack digunakan dalam berbagai aplikasi yang berbeda, dan sebagai alat untuk banyak struktur data dan algoritma yang lebih canggih. Secara formal, stack adalah tipe data abstrak (ADT) di mana sebuah instance S mendukung dua metode berikut:

- S.push(e) : Menambahkan elemen e ke paling atas dari stack S.
- S.pop() : Menghapus dan mengembalikan elemen paling atas dari stack S; akan menyebabkan error jika stack kosong.
- S.top() : Memanggil elemen paling atas dari stack S tanpa menghapusnya; akan menyebabkan error jika stack kosong.
- S.is_empty(): Mengembalikan True jika stack S tidak ada isinya.
- len(S) : Mengembalikan banyaknya elemen yang ada di dalam stack S; pada Python kita mengimplementasikannya dengan metode special __len__.

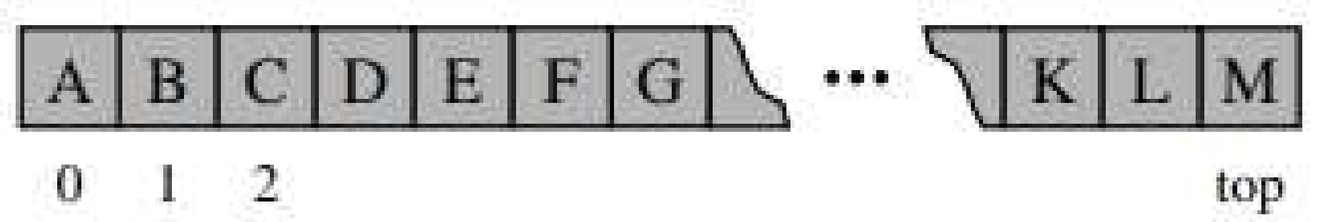
6.1.1 Stack untuk Tipe Data Abstrak

Menurut konvensi , kita berasumsi bahwa stack yang baru dibuat adalah kosong , dan tidak ada batasan awal pada kapasitas stack . Elemen yang ditambahkan ke stack dapat memiliki tipe data apa pun . Tabel disamping menunjukkan serangkaian operasi stack dan pengaruhnya pada stack S bilangan bulat yang awalnya kosong :

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop( )	3	[5]
S.is_empty( )	False	[5]
S.pop( )	5	[ ]
S.is_empty( )	True	[ ]
S.pop( )	“error”	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top( )	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]

6.1.2 Implementasi Sederhana Stack Berbasis Array

Kita dapat mengimplementasikan stack dengan menyimpan elemen-elemennya dalam list Python. Kelas list sudah mendukung penambahan elemen ke akhir dengan metode append, dan menghapus elemen terakhir dengan metode pop, sehingga wajar untuk menempatkan elemen teratas stack di akhir list, seperti yang ditunjukkan pada Gambar.



Meskipun kita dapat langsung menggunakan kelas list sebagai pengganti kelas stack formal, list juga dapat menambah atau menghapus elemen dari sembarang indeks yang akan merusak abstraksi yang diwakili oleh ADT stack. Selain itu, terminologi yang digunakan oleh kelas list tidak secara tepat sesuai dengan nomenklatur tradisional untuk ADT stack, khususnya perbedaan antara append dan push.

6.1.2 Implementasi Sederhana Stack Berbasis Array

The Adapter Pattern

Pola desain adapter berlaku dalam konteks apa pun di mana kita secara efektif ingin memodifikasi kelas yang ada sehingga metodenya cocok dengan kelas atau antarmuka lain yang terkait. Salah satu cara umum untuk menerapkan pola desain adapter adalah dengan mendefinisikan kelas baru sedemikian rupa sehingga berisi instance dari kelas yang ada sebagai bidang tersembunyi, dan kemudian menerapkan setiap metode kelas baru menggunakan metode variabel instance tersembunyi ini. Dalam konteks ADT stack, kita dapat mengadaptasi kelas list Python menggunakan korespondensi yang ditunjukkan pada tabel berikut.

Stack Method	*Realization with Python list*
S.push(e)	L.append(e)
S.pop( )	L.pop( )
S.top( )	L[−1]
S.is_empty( )	len(L) == 0
len(S)	len(L)

6.1.2 Implementasi Sederhana Stack Berbasis Array

Implementasi Stack Menggunakan List Python

Ketika pop dipanggil pada list Python kosong, secara formal akan memunculkan IndexError, karena list adalah sequence berbasis indeks. Pilihan itu tampaknya tidak tepat untuk stack, karena tidak ada asumsi indeks. Sebagai gantinya, kita dapat mendefinisikan kelas exception baru yang lebih sesuai. Kode di bawah mendefinisikan kelas Empty sebagai subkelas trivial dari kelas Exception Python.

```
class Empty(Exception):
  """Error attempting to access an element from an empty container."""
  pass
```

6.1.2 Implementasi Sederhana Stack Berbasis Array

Contoh Penggunaan

```
S = ArrayStack( )                 # contents: [ ]
S.push(5)                         # contents: [5]
S.push(3)                         # contents: [5, 3]
print(len(S))                     # contents: [5, 3];        outputs 2
print(S.pop( ))                   # contents: [5];           outputs 3
print(S.is_empty( ))              # contents: [5];           outputs False
print(S.pop( ))                   # contents: [ ];           outputs 5
print(S.is_empty( ))              # contents: [ ];           outputs True
S.push(7)                         # contents: [7]
S.push(9)                         # contents: [7, 9]
print(S.top( ))                   # contents: [7, 9];        outputs 9
S.push(4)                         # contents: [7, 9, 4]
print(len(S))                     # contents: [7, 9, 4];     outputs 3
print(S.pop( ))                   # contents: [7, 9];        outputs 4
S.push(6)                         # contents: [7, 9, 6]
```

6.1.2 Implementasi Sederhana Stack Berbasis Array

```
class ArrayStack:
  """LIFO Stack implementation using a Python list as underlying storage."""
  def init (self):
      """Create an empty stack."""
      self. data = [ ] # nonpublic list instance
  def len (self):
      """Return the number of elements in the stack."""
      return len(self. data)
  def is_empty(self):
      """Return True if the stack is empty."""
      return len(self. data) == 0
  def push(self, e):
      """Add element e to the top of the stack."""
      self. data.append(e) # new item stored at end of list
  def top(self):
      """"Return (but do not remove) the element at the top of the stack. Raise Empty exception if the stack is empty."""
      if self.is_empty( ):
        raise Empty( Stack is empty )
      return self. data[-1] # the last item in the list
  def pop(self):
      """"Remove and return the element from the top of the stack (i.e., LIFO).Raise Empty exception if the stack is empty."""
      if self.is_empty( ):
        raise Empty( Stack is empty )
      return self. data.pop( ) # remove last item from list
```

6.1.2 Implementasi Sederhana Stack Berbasis Array

Analisis Implementasi Stack Berbasis Array

Implementasi untuk top, is_empty, dan len menggunakan waktu konstan dalam kasus terburuk. Waktu O(1) untuk push dan pop adalah batas teramortized (lihat Bagian 5.3.2); pemanggilan khas ke salah satu metode ini menggunakan waktu konstan, tetapi terkadang ada kasus terburuk O(n), di mana n adalah jumlah elemen saat ini di stack, ketika suatu operasi menyebabkan list untuk mengubah ukuran array internalnya. Penggunaan ruang untuk stack adalah O(n). Untuk space usage adalah O(n) dengan n adalah jumlah elemen di stack

Operation	Running Time
S.push(e)	$O(1)^*$
S.pop()	$O(1)^*$
S.top()	$O(1)$
S.is_empty()	$O(1)$
len(S)	$O(1)$

*amortized

6.1.3 MEMBALIK DATA MENGGUNAKAN STACK

Sebagai konsekuensi dari protokol LIFO, stack dapat digunakan sebagai alat umum untuk membalik urutan data. Misalnya, kita mungkin ingin mencetak baris-baris file dalam urutan terbalik untuk menampilkan kumpulan data dalam urutan menurun. Ini dapat dicapai dengan membaca setiap baris dan memasukkannya ke dalam stack, lalu menulis baris-baris tersebut dalam urutan mereka dikeluarkan.

```
def reverse file(filename):
  """Overwrite given file with its contents line-by-line reversed."""
  S = ArrayStack( )
  original = open(filename)
  for line in original:
    S.push(line.rstrip( \n )) # we will re-insert newlines when writing
  original.close( )

  # now we overwrite with contents in LIFO order
  output = open(filename, w ) # reopening file overwrites original
  while not S.is empty( ):
    output.write(S.pop( ) + \n ) # re-insert newline characters
  output.close( )
```

6.1.4 MENCOCOKKAN TANDA KURUNG DAN TAG HTML

```
def is_matched(expr):
 """Return True if all delimiters are properly match; False otherwise."""
 lefty = "({[" # opening delimiters
 righty = ")}]" # respective closing delims
 S = ArrayStack( )
 for c in expr:
   if c in lefty:
     S.push(c) # push left delimiter on stack
   elif c in righty:
     if S.is_empty( ):
       return False # nothing to match with
     if righty.index(c) != lefty.index(S.pop( )):
       return False # mismatched
 return S.is_empty( )
```

Contoh aplikasi dari stack adalah mempertimbangkan ekspresi aritmatika yang mungkin berisi berbagai pasangan simbol pengelompokan, seperti:

- Kurung: “(“ dan “)”
- Kurung Kurawal: “{“ dan “}”
- Kurung Siku : “[“ dan “]”

Setiap simbol pembuka dan penutup harus sama, contohnya seperti [(5+x)-(y+z)]. Berikut beberapa ilustrasi contoh dari konsep ini:

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

6.1.4 MENCOCOKKAN TANDA KURUNG DAN TAG HTML

Mencocokkan Tag dalam Bahasa Markup

Algoritma di samping membaca dari kiri ke kanan melalui string mentah, menggunakan indeks j untuk melacak kemajuan kami dan metode find dari kelas str untuk menemukan karakter < dan > yang menentukan tag. Tag pembuka didorong ke dalam stack, dan dicocokkan dengan tag penutup saat dikeluarkan dari stack. Dengan analisis serupa, algoritma ini berjalan dalam waktu O(n), di mana n adalah jumlah karakter dalam sumber HTML mentah.

```
def is_matched_html(raw):
  """Return True if all HTML tags are properly match; False otherwise."""
  S = ArrayStack( )
  j = raw.find( "<" ) # find first '<' character (if any)
  while j != -1:
    k = raw.find( ">" , j+1) # find next '>' character
    if k == -1:
        return False # invalid tag
    tag = raw[j+1:k] # strip away < >
    if not tag.startswith("/"): # this is opening tag
       S.push(tag)
    else: # this is closing tag
      if S.is_empty( ):
        return False # nothing to match with
      if tag[1:] != S.pop( ):
        return False # mismatched delimiter
    j = raw.find( "<" , k+1) # find next '<' character (if any)
  return S.is_empty( ) # were all opening tags matched?
```

MARI BERDISKUSI

Silakan bertanya.

TERIMA KASIH

Mohon maaf atas kesalahan dan kekurangan selama presentasi berlangsung.