STACKS,QUEUE, & DEQUES

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__.

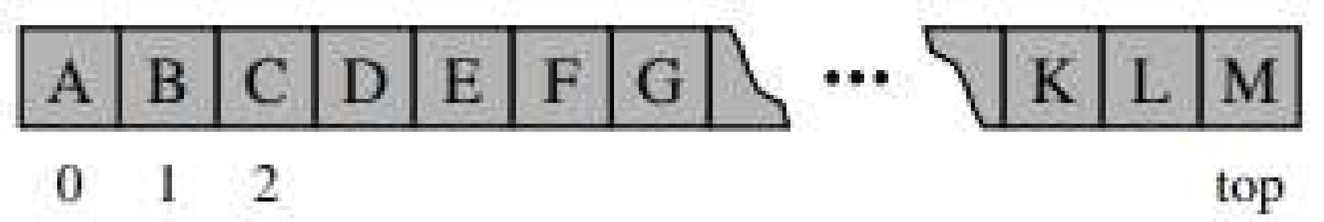
6.1.1 Stack untuk Tipe Data Abstrak

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

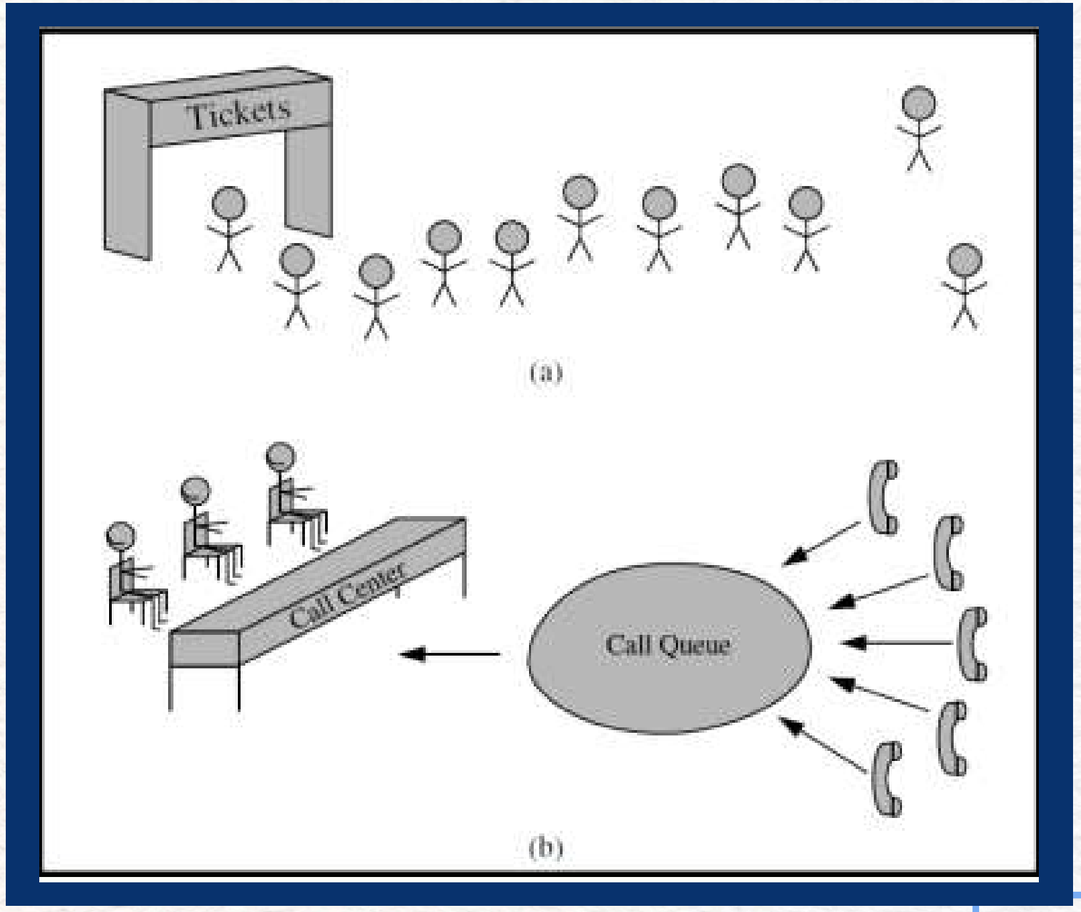
Algoritma di samping membaca dari kiri ke kanan melalui string mentah, menggunakan indeks j untuk melacak kemajuan kami dan metode find dari kelas str untuk menemukan karakter < dan > yang menentukan tag. Tag pembuka didorong ke dalam stack, dan dicocokkan dengan tag penutup saat dikeluarkan dari stack. 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?
```

QUEUES

Queue atau antrian merupakan struktur data linear yang mengikuti prinsip FIFO (First In First Out) , artinya adalah data yang pertama kali dimasukkan atau disimpan , maka data tersebut adalah yang pertama kali akan diakses atau dikeluarkan .

Metafora untuk terminologi ini adalah antrean orang yang menunggu untuk naik wahana di taman hiburan . Antrian FIFO juga digunakan oleh banyak perangkat komputasi , seperti networked printer, atau Web Server yang menanggapi permintaan .



6.2.1 Queue untuk Tipe Data Abstrak

Q.enqueue(e) : Menambah elemen e ke belakang antrian

Q. Q.dequeue() : Menghapus dan mengembalikan elemen pertama dari antrian Q: akan error jika antrian kosong.

Q.first() : Mengembalikan elemen yang ada di depan antrian Q tanpa menghapusnya; akan error jika antrian kosong.

Q.is_empty() : Mengembalikan True jika antrian Q tidak ada elemen apapun.

len(Q) : Mengembalikan banyaknya elemen di antrian Q; pada Python, implementasinya menggunakan metode special __len__.

6.2.2 Implementasi Queue Berbasis Array

Using an Array Circularly

Untuk membuat implementasi queue yang lebih kokoh, kita biarkan elemen depan queue bergeser ke kanan, dan elemen-elemen dalam queue "berputar" di sekitar ujung array yang mendasarinya. Kita berasumsi bahwa array yang mendasari memiliki panjang tetap N yang lebih besar dari jumlah elemen sebenarnya dalam queue. Elemen baru dimasukkan ke arah "ujung" dari queue saat ini, bergerak dari depan ke indeks N - 1 dan berlanjut di indeks 0, lalu 1. Gambar di bawah mengilustrasikan queue seperti itu dengan elemen pertama E dan elemen terakhir M.

Figure 6.6: Modeling a queue with a circular array that wraps around the end.

6.2.2 IMPLEMENTASI QUEUE BERBASIS ARRAY

Implementasi Queue pada Python

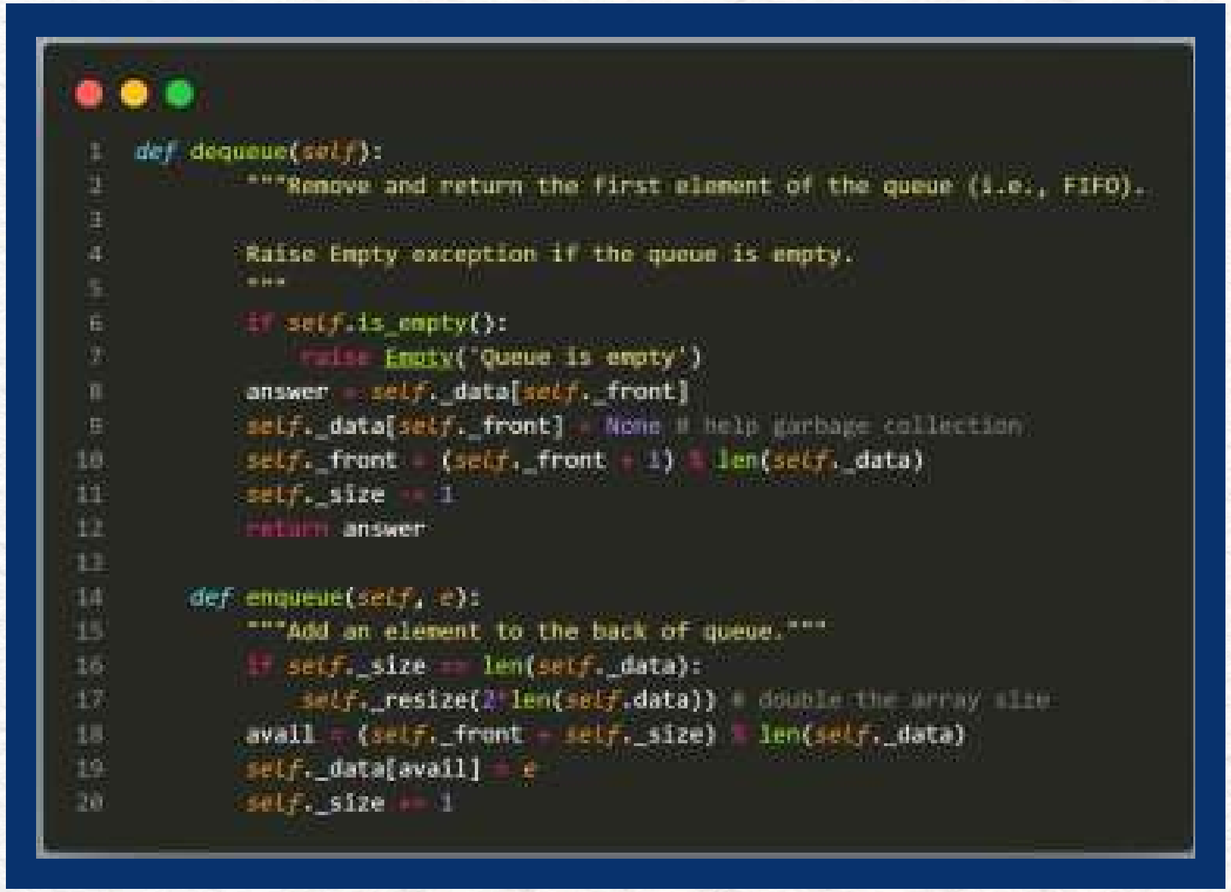
Implementasi queue menggunakan list Python secara melingkar dijelaskan dalam kode di samping. Queue ini menggunakan 3 variabel:

- _data: list dengan kapasitas tetap untuk menyimpan elemen queue.
- _size: jumlah elemen yang saat ini ada di queue.
- _front: indeks elemen pertama dalam list data (jika queue tidak kosong).

Awalnya kita siapkan list kosong dan set front ke 0. Jika kita panggil front atau dequeue pada queue kosong, exception Empty akan muncul.

6.2.2 IMPLEMENTASI QUEUE BERBASIS ARRAY

Menambahkan dan Menghapus Elemen



```
def dequeue(self):
        """Remove and return the first element of the queue (i.e., FIFO).

        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Empty('Queue is empty')
        answer = self._data[self._front]
        self._data[self._front] = None # help garbage collection
        self._front = (self._front + 1) % len(self._data)
        self._size -= 1
        return answer

    def enqueue(self, e):
        """Add an element to the back of queue."""
        if self._size == len(self._data):
            self._resize(2 * len(self.data)) # double the array size
        avail = (self._front + self._size) % len(self._data)
        self._data[avail] = e
        self._size += 1
```

- Enqueue menambahkan elemen baru ke belakang queue. Indeks elemen baru ditentukan dengan rumus:

  avail = (self.front + self.size) % len(self.data)

- Dequeue menghapus elemen terdepan dari queue. Elemen yang dihapus akan disimpan sementara sebelum dihapus permanen dari list data menggunakan

  self.data[self.front] = None.

Hal ini untuk membantu mekanisme Python dalam mengelola memori. Setelah dihapus, _front di-update untuk menunjukkan elemen berikutnya yang menjadi yang terdepan.

6.2.2 IMPLEMENTASI QUEUE BERBASIS ARRAY

Jika ukuran queue sama dengan kapasitas list data, maka kapasitas list data akan digandakan . Ini mirip dengan cara kerja DynamicArray pada Section 5 . 3 . 1 . Namun , resize pada queue membutuhkan perhatian lebih .

Kita buat list sementara untuk menyimpan data lama, lalu alokasikan list baru dengan ukuran dua kali lipat . Saat memindahkan data, kita sengaja mengatur ulang elemen terdepan queue menjadi indeks 0 di array baru (lihat pada gambar) . Penataan ulang ini penting karena perhitungan modular aritmetik bergantung pada ukuran array.

Mengubah Ukuran Queue

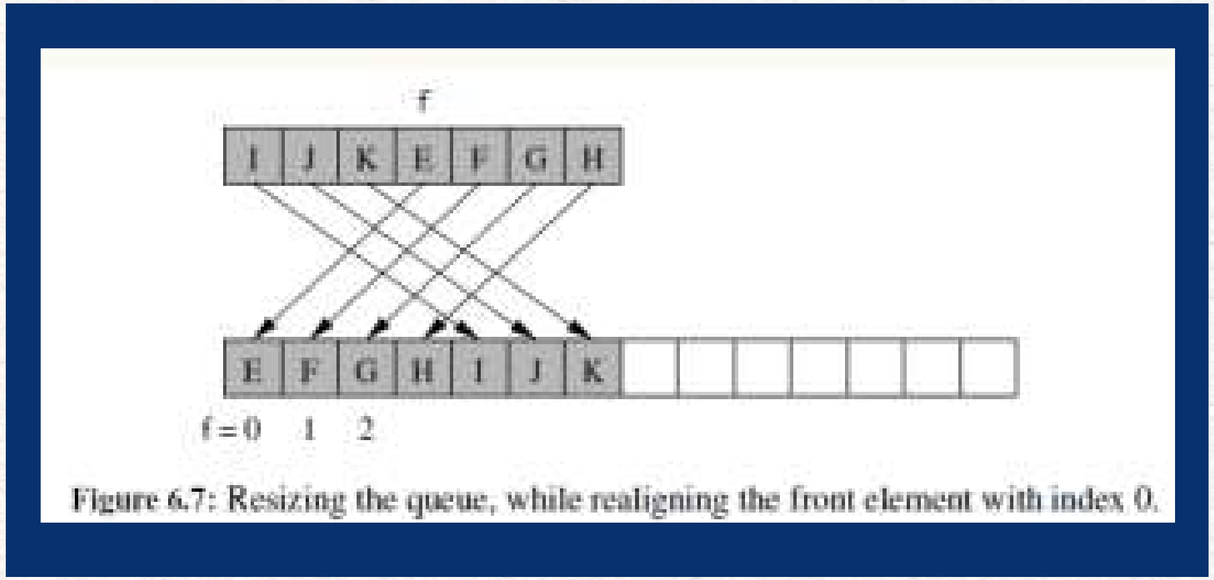


Figure 6.7: Resizing the queue, while realigning the front element with index 0.

6.2.2 Implementasi Queue Berbasis Array

Shrinking the Underlying Array

Implementasi queue menggunakan ArrayQueue (Kode Fragmen 6.6 dan 6.7) tidak memiliki penggunaan ruang yang ideal (Θ(n) dimana n adalah jumlah elemen saat ini). Kapasitas array dasar akan bertambah ketika enqueue dipanggil pada queue yang penuh, namun dequeue tidak pernah mengecilkan array. Akibatnya, kapasitas array dasar menjadi proporsional dengan jumlah maksimum elemen yang pernah disimpan, bukan jumlah elemen saat ini.

Untuk mengatasi hal ini, kita bisa mengurangi ukuran array menjadi setengah dari ukuran saat ini ketika jumlah elemen yang tersimpan kurang dari seperempat kapasitasnya. Strategi ini dapat diterapkan dengan menambahkan dua baris kode berikut dalam method dequeue (setelah baris 38 pada Kode Fragmen 6.6):

```
if 0 < self._size < len(self._data) // 4
        : self._resize(len(self._data) // 2)
```

6.2.2 IMPLEMENTASI QUEUE BERBASIS ARRAY

Tabel di samping menjelaskan kinerja implementasi queue berbasis array, dengan asumsi peningkatan terkadang mengecilkan ukuran array. Dengan pengecualian utilitas _resize, semua metode mengandalkan sejumlah pernyataan konstan yang melibatkan operasi aritmatika, perbandingan, dan penugasan. Oleh karena itu, setiap metode berjalan dalam worst-case O(1) waktu, kecuali enqueue dan dequeue, yang memiliki batas amortisasi O(1) waktu.

Batasan untuk enqueue dan dequeue bersifat amortisasi karena perubahan ukuran array. Penggunaan ruang adalah O(n), di mana n adalah jumlah elemen saat ini dalam antrian.

Analisis Implementasi Queue Berbasis Array

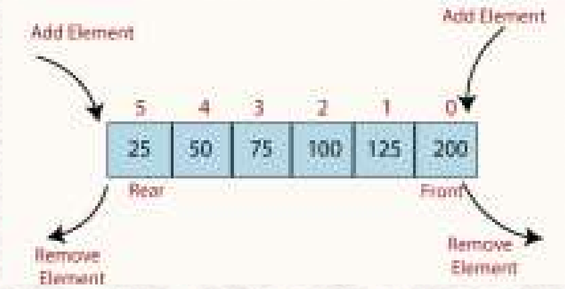
Operation	Running Time
Q.enqueue(e)	$O(1)^*$
Q.dequeue( )	$O(1)^*$
Q.first( )	$O(1)$
Q.is_empty( )	$O(1)$
len(Q)	$O(1)$

*amortized

DOUBLE-ENDED QUEUES (DEQUES)

struktur data seperti antrian yang mendukung penyisipan dan penghapusan elemen baik dari depan maupun belakang antrian. Struktur seperti ini disebut deque (diucapkan "dek"). Deque merupakan kependekan dari Double-Ended Queue.

Tipe data abstrak deque lebih umum daripada ADT stack dan queue. Generalisasi ekstra ini dapat berguna dalam beberapa aplikasi. Misalnya, pada sebuah restoran menggunakan antrian untuk mempertahankan daftar tunggu. Kadang-kadang, orang pertama mungkin dikeluarkan dari antrian hanya untuk mengetahui bahwa meja belum tersedia; biasanya, restoran akan memasukkan kembali orang tersebut di posisi pertama dalam antrian. Mungkin juga pelanggan di akhir antrian menjadi tidak sabar dan meninggalkan restoran. (Kita akan membutuhkan struktur data yang lebih umum lagi jika kita ingin memodelkan pelanggan yang meninggalkan antrian dari posisi lain.



6.3.1 DEQUE UNTUK TIPE DATA ABSTRAK

Untuk memberikan abstraksi yang simetris, ADT deque didefinisikan sehingga deque D mendukung metode berikut:

- D.add_first(e) : Menambahkan elemen e ke depan deque D.
- D.add_last(e) : Menambahkan elemen e ke belakang deque D.
- D.delete_first(): Menghapus dan mengembalikan elemen pertama dari deque D; error terjadi jika deque kosong.
- D.delete_last() : Menghapus dan mengembalikan elemen terakhir dari deque D; error terjadi jika deque kosong. Selain itu, ADT deque akan mencakup accessor berikut:
- D.first() : Mengembalikan (tetapi tidak menghapus) elemen pertama dari deque D; error terjadi jika deque kosong.
- D.last() : Mengembalikan (tetapi tidak menghapus) elemen terakhir dari deque D; error terjadi jika deque kosong.
- D.is_empty() : Mengembalikan True jika deque D tidak mengandung elemen apa pun.
- len(D)` : Mengembalikan jumlah elemen dalam deque D; dalam Python, kita menerapkan ini dengan metode khusus len.

6.3.1 DEQUE UNTUK TIPE DATA ABSTRAK

Tabel disamping menunjukkan serangkaian operasi deque dan pengaruhnya pada deque D bilangan bulat yang awalnya kosong

Operation	Return Value	Deque
D.add_last(5)	–	[5]
D.add_first(3)	–	[3, 5]
D.add_first(7)	–	[7, 3, 5]
D.first( )	7	[7, 3, 5]
D.delete_last( )	5	[7, 3]
len(D)	2	[7, 3]
D.delete_last( )	3	[7]
D.delete_last( )	7	[ ]
D.add_first(6)	–	[6]
D.last( )	6	[6]
D.add_first(8)	–	[8, 6]
D.is_empty( )	False	[8, 6]
D.last( )	6	[8, 6]

6.3.2 IMPLEMENTASI DEQUE DENGAN CIRCULAR ARRAY

Kita dapat mengimplementasikan ADT deque dengan cara yang sangat mirip dengan kelas ArrayQueue yang disediakan dalam Fragmen Kode 6.6 dan 6.7 dari Bagian 6.2.2. Kita tetap mempertahankan tiga variabel instans yang sama: _data, _size, dan _front. Setiap kali kita perlu mengetahui indeks belakang deque, atau slot pertama yang tersedia di luar belakang deque, kita menggunakan aritmatika modular untuk perhitungan. Misalnya, implementasi metode last() menggunakan indeks:

```
back = (self.front + self.size - 1) % len(self.data)
```

Implementasi ArrayDeque.add_last pada dasarnya sama dengan ArrayQueue.enqueue, termasuk ketergantungan pada utilitas_resize. Demikian juga, implementasi ArrayDeque.delete first method sama dengan ArrayQueue.dequeue. Implementasi add_first dan delete_last menggunakan teknik serupa. Satu kehalusan adalah bahwa panggilan ke add_first mungkin perlu membungkus bagian awal array, jadi kita menggunakan aritmatika modular untuk secara melingkar mengurangi indeks, sebagai:

```
self.front = (self.front - 1) % len(self.data) # pergeseran siklik
```

Efisiensi ArrayDeque mirip dengan ArrayQueue, dengan semua operasi memiliki waktu berjalan O(1), tetapi dengan batasan yang diamortisasi untuk operasi yang dapat mengubah ukuran daftar yang mendasarinya.

6.3.3 DEQUE PADA KOLEKSI MODUL PYTHON

Kelas deque secara resmi didokumentasikan untuk menjamin operasi O(1) di kedua ujungnya, tetapi operasi dengan kasus terburuk O(n) saat menggunakan notasi indeks di dekat tengah deque.

collections.deque adalah implementasi deque yang efisien dalam Python, konsisten dengan penamaan list, mendukung panjang tetap, dan memiliki operasi cepat di kedua ujungnya. Namun, operasi di tengah deque memiliki waktu yang lebih lama.

Our Deque ADT	collections.deque	Description
len(D)	len(D)	number of elements
D.add_first( )	D.appendleft( )	add to beginning
D.add_last( )	D.append( )	add to end
D.delete_first( )	D.popleft( )	remove from beginning
D.delete_last( )	D.pop( )	remove from end
D.first( )	D[0]	access first element
D.last( )	D[−1]	access last element
	D[j]	access arbitrary entry by index
	D[j] = val	modify arbitrary entry by index
	D.clear( )	clear all contents
	D.rotate(k)	circularly shift rightward k steps
	D.remove(e)	remove first matching element
	D.count(e)	count number of matches for e

6.3.4 IMPLEMENTASI DEQUE PADA KOLEKSI MODUL PYTHON

MARI BERDISKUSI

Silakan bertanya.

TERIMA KASIH

Mohon maaf atas kesalahan dan kekurangan selama presentasi berlangsung.