Inlämningsuppgift 1
Datastrukturer och algoritmer, 7,5 hp
Systemutveckling II (DA193A), vt11

Innehåll

1 Inledning

Denna inlämningsuppgift syftar till att ge färdigheter i användandet av datastrukturer i programspråket python. Uppgiften går ut på att implementera en FIFO-kö med en cirkulär array. där varje kö-element är objekt av en egendefinierad klass. Via ett grafiskt gränssnitt ska användaren kunna lägga till personer i kön, ta bort personer ur kön och köns innehåll ska visas i en grafisk komponent.

- Literatur: *Cormen, Leiserson, Rivest & Stein: Introduction to Algorithms, 3rd edition, MIT Press, 2009*

Kursboken kommer refereras till med beteckningen [CLR]

1 Redovisning

Laborationen skall redovisas genom att följande material produceras, och levereras via It's learning [1].

- En kort rapport, i pdf format, som beskriver de erfarenheter du gjort under laborationen.
- Källkodsfiler (skriv namn i alla källkodsfiler), med programmet för er implementation av uppgiften.

All kod ni skickar in ska följa god kodstandard (t ex beskrivande namn på variabler och metoder, osv)

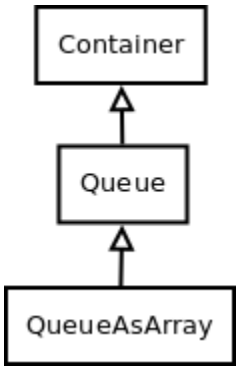
Laborationen ska skickas in senast fredagen den 11/01 kl 23.55 i en zip-fil via It's learning. Obs, inga åäö i filnamnet!

2 Inlämningsuppgiften

Uppgiften går ut på att ni ska implementera en kö av typen FIFO (first in first out) som ska lagra objekt av en egendefinierad klass som ska innehålla minst 3 attribut.
Kön ska implementeras i form av en cirkulär array där alla arrayens element kan utnyttjas utan att någon resurskrävande omförflyttning av elementen krävs.
Via ett grafiskt gränssnitt ska användaren kunna lägga till objekt i kön, ta bort objekt ur kön och köns innehåll ska skrivas ut och uppdateras i en grafisk komponent.

2.1 Implementation av kön

De flesta datastrukturer kan ses som behållare som innehåller och hanterar ett antal element. Datastrukturer har ett antal gemensamma egenskaper som vi samlar i en abstrakt basklass. Klassen Container ska innehålla ett antal metoder med egenskaper som är gemensamma för flera olika datastrukturer. Basklassens egenskaper kan sedan ärvas av de klasser där vi implementerar de olika datastrukturerna. De två första inlämningsuppgifterna kommer att handla om köer vilka också har gemensamma egenskaper. Vi ska därför även deklarera en abstrakt basklass för köer. Vår FIFO kö ska denna vecka implementeras med de klasser och relationer som visas på bilden nedan. Varje klass ska placeras i en egen fil/modul.



2.2 Klassen – Container

Klassen ska innehålla ett antal metoder som ofta används vid implementation av olika datastrukturer. Alla klasser i python måste direkt eller indirekt ärva den inbyggda basklassen object. Klassen Container ska därmed ärva klassen objekt.

- Börja med att importera följande moduler:

```
import sys
from abc import abstractmethod
```

- Klassen Container ska innehålla en heltalsvariabel som används till att räkna antalet element som en datastruktur håller.

- Implemetera följande metoder i klassen:

- purge

Översatt till svenska betyder purge utrensning. Funktionen ska tömma datastrukturen på alla dess element men i denna klass ska metoden purge vara abstrakt, dvs. den ska inte implementeras i klassen Container utan ska deklareras som abstrakt.

Abstrakta metoder deklareras på följande sätt:

```
@abstractmethod
def purge(self):
     pass
```

Notera att exemplet ovan gäller för alla abstrakta metoder.

- __iter__

Även denna metod ska vara abstrakt och ska då inte implementeras här. Metoden ska används till att iterera igenom datastrukturens alla element.

- getNbrElem

Metoden ska returnera antalet element i datastrukturen, dvs värdet på attributet som håller antalet element i datastrukturen. Deklarera sedan en så kallad property som använder metoden getNbrElem.

Exempel på deklaration av en property:

```
nbrElem = property(
                fget = lambda self: self._getNbrElem())
```

En property förenklar syntaxen vid användandet av get och set metoder.

- getIsEmpty

Metoden ska returnera true om datastrukturen är tom, annars false.

- getIsFull

Metoden ska returnera false på grund av att kommande dynamiska datastrukturer inte har någon övre gräns för antlet element som den kan hålla.
Deklarera även en property som använder denna metod.

2.3 Klassen - Queue

Klassen Queue ska ärva klassen Container och utöver de ärvda metoderna ska klassen dessutom innehålla två abstrakta metoder, nämligen enqueue och dequeue.

Börja med att importera följande moduler:

```
import sys
from container import Container
from abc import abstractmethod
```

- enqueue

Metoden ska användas till att sätta in en nytt element sist i kön, men i och med att den i klassen Queue ska vara abstrakt ska den inte implementeras.

- dequeue

Metoden ska användas till att ta ut det första elementet i kön, men i och med att den i klassen Queue ska vara abstrakt ska den inte implementeras.

2.4 Klassen - QueueAsArray

Klassen QueueAsArray ska ärva klassen Queue och det är i denna subklass som vi ska implementera den slutgiltiga funktionaliteten hos vår datastruktur.
Som argument ska klassen ta ett heltalsvärde för antalet element som list-variabeln ska kunna hålla. Argumentet ska lagras i en datamedlem i klassen.

Börja med att importera följande moduler:

```
import sys
import exceptions
from queue import Queue
```

I python används en datatyp med namnet list som är en motsvarighet till vad som kallas en array i andra språk. Vi ska i implementationen av vår kö använda en list-variabel till att hålla de element som läggs i kön.
Deklarera och initiera en list-variabel på följande vis:

```
self._list = [None for i in xrange(size)]
```

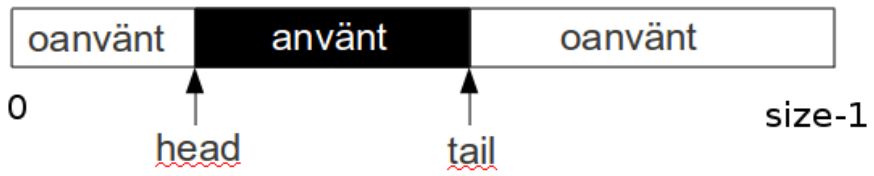
Ytterligare två datamedlemar head och tail ska deklareras som ska hålla index för första respektive sista elementet i kön.

Klassen ska sammanfattningsvis innehålla följande datamedlemmar:

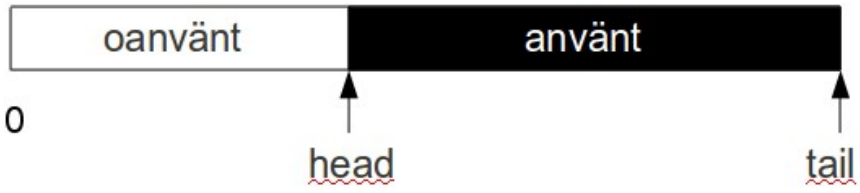
```
self._size
self._list
self._head
self._tail
```

Man bör av effektivitetsskäl inte flytta alla element i list-variabeln framåt när ett element tas ut längst fram i kön. Vi ska istället använda oss av två heltalsvariablerna head och tail till att hålla ordning på positionen (d v s list-index) för det första respektive sista elementet i kön.
De tre följande figurerna visar hur vi med hjälp av heltalsvariablerna head och tail kan utnyttja alla list-variabelns lediga positioner utan att behöva flytta något element.

Fall 1. Antag att kön efter en viss tid ser ut enligt följande figur:

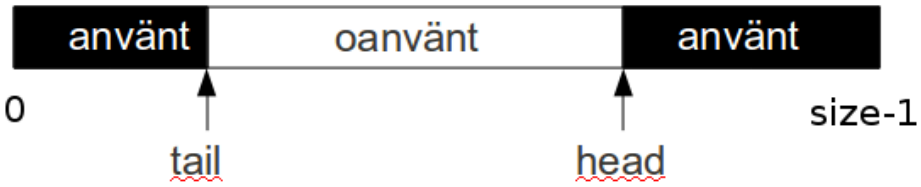


Fall 2. Då tail har index för list-variabelns position size -1 finns lediga positioner i den främre delen av arrayen:



I fall 2 kan list-variabeln göras cirkulär genom att variabeln tail ”hoppar” från position size-1 till position 0. Vi använder modulus operatorn % till att få variablerna tail och head att ”hoppa” från slutet till början av list-variabeln.

Fall 3. Efter ytterligare en tid (d v s element har lagts till och tagits bort) kan då list-variabeln se ut på följande vis:



- **purge**

Metoden ska tömma kön på dess innehåll genom att iterera igenom och sätta alla elementen i list-variablen till None.

- **enqueue**

Metoden ska lägga till ett objekt i kön på positionen för variabeln tail. Först ska det kontrolleras att kön inte är full. Om kön är full ska ett IndexError meddelande genereras.

```
raise IndexError
```

Använd modulus till att göra list-variabeln cirkulär genom att om tail har index size-1 ska värdet på tail ”hoppa” till index 0 nästa gång ett element läggs till i kön.
Inkrementera värdet på variabeln som håller antalet element som finns i kön.

- dequeue

Metoden ska ta bort ett objekt från kön på positionen för variabeln head. Kontrollera först att kön inte är tom. Om kön är tom ska ett IndexError meddelande genereras.

```
raise IndexError
```

Använd modulus på samma sätt som i metoden enqueue för att göra list-variabeln cirkulär. Räkna ner värdet på variabeln som håller antalet element som finns i kön.

- getIsFull

Metoden ska returnera true om kön är full, annars false.

- __iter__

Metoden ska returnera ett iterator objekt innehållande det aktuella QueueAsArray objektet.

```
def __iter__(self):
        return self.Iterator(self)
```

- Iterator - en inre klass

Klassen QueueAsArray ska innehålla en inre klass med namnet Iterator som ska användas till att itererar igenom alla köns element. Den ska som argument ta ett Queue objekt som ska lagras i en datamedlem. Den inre klassen ska även ha en datamedlem som räknar det antal element som iterator-objektet har itererat.

- next

Den inre klassen Iterator ska ha en metod med namnet next som ska returnera objektet för iteratorns aktuella position. Om iteratorn nått slutet på kön ska ett StopIteration meddelande genereras.

```
raise StopIteration
```

Använd modulus på samma sätt som i metoderna enqueue och dequeue så att alla objekten i kön kan skrivas ut även när kön är i så kallat ”wrapped state”.

3. Vald kö-data

Deklarera en valfri klass med minst tre attribut och som ska fungera som kö-data. Klassen ska innehålla get och set metoder samt properties så att attributens värden kan hämtas och tilldelas. Klassen ska även innehålla en metod men namnet __str__ som ska returnera en textsträng med information om de värden som klassens attribut håller.

Ett exempel på en property som använder både get och set metoder:

```
firstName = property(
        fset = lambda self, value: self.setFirstName(value),
        fget = lambda self: self.getFirstName())
```

4. Konstruera ett enkelt grafiskt gränssnitt

Använd ramverket Tkinter till att konstruera ett enkelt grafiskt gränssnitt. Se dokumentet Tkinter som finns att ladda ner från it's learning [1]

Referenser

[1] It’s learning
http://www.itslearning.com/elogin