4. Colas

Programación II

Grado en Inteligencia Artificial

M. Alonso, M. Cabrero y E. Hernández

fic facultade de informática da coruña

UNIVERSIDADE DA CORUÑA

Contenidos

Introducción

Estructuras lineales: recordemos

- Colección de datos cuyos elementos están ordenados dependiendo de cómo han sido añadidos o eliminados.
- Son estructuras con dos terminaciones que, dependiendo de la estructura, adoptan distintos nombres: izquierda y derecha, frente y cola, cima y fondo.
- Lo que distingue una estructura lineal de otra es el modo en que los elementos se añaden o se eliminan.
  - P.ej. hay estructuras que sólo permiten insertar por sólo un extremo mientras que otras permiten eliminar por cualquier extremo.
- Pilas, colas y listas son los tipos más representativos.

Definición

Concepto

- Metáfora de la cola de gente esperando...la entrada al museo, en la oficina de empleo, en el comedor universitario, al teléfono en contacto con atención al cliente, etc.
- Cuando atienden al que nos precede, nos convertimos en el frente de la cola.

¿Qué es una cola?

- Colección de elementos en la que las operaciones de inserción de nuevos elementos y eliminación de elementos existentes tiene lugar por distintos extremos.
- El principio de la estructura o extremo en el que tienen lugar la eliminaciones se denomina *frente*.
- Los elementos más cercanos al final representan aquellos elementos que llevan menos tiempo en la cola, mientras que los más tiempo llevan son los primeros en ser eliminados.
- Este principio de ordenación (temporal) se llama **FIFO**, *first-in first-out*.

El principio FIFO

- Se aplica donde sea necesario gestionar recursos y atender demandas en función de algún criterio como el momento de la solicitud: trabajos de impresión que se envían a una misma impresora, gestión de procesos de usuario (*scheduling* en un ordenador), un servidor Web respondiendo a solicitudes de clientes, etc.

El TAD Cola

Valores

- Una cola es una colección lineal de elementos que se añaden por un extremo de la estructura y se eliminan por otro extremo distinto denominado *frente*. Los elementos está ordenados por su posición siguiendo un esquema FIFO.
- Una cola puede estar vacía.
- Si no está vacía:
  - Hay un único elemento que ocupa el principio de la cola.
  - Hay un único elemento que ocupa el final de la cola.
  - Para todo elemento, salvo el primero, hay un único elemento ubicado en la posición anterior.
  - Para todo elemento, salvo el último, hay un único elemento ubicado en la posición siguiente.

Operaciones Constructoras Generadoras

- **Queue()**

  Objetivo: Crear una cola vacía
  Nota: en Python se traduce automáticamente en una llamada `self.__init__()`

  Salida: Una cola vacía

- **enqueue(e)**

  Objetivo: Añadir el elemento *e* a la cola

  Entradas: Una cola (*self*) y un elemento *e*

  Salidas: La cola con el elemento *e* añadido al final

Operaciones Destructoras

- **dequeue() $\rightarrow$ e**

  Objetivo: Eliminar un elemento de la cola

  Entradas: Una cola (*self*)

  Salidas: La cola sin el elemento *e* del frente
  El elemento *e* que anteriormente estaba en el frente

  Precondición: La cola no está vacía

Operaciones Observadoras I

- **first() $\rightarrow$ e**

  Objetivo: Devolver el primer elemento *e* de la cola (*frente*)

  Entradas: Una cola (*self*)

  Salidas: El elemento *e* que está en el frente

  Precondición: La cola no está vacía

Operaciones Observadoras II

- **len(S)** $\rightarrow$ n

  Objetivo: Devolver el número de elementos en la cola

  Nota: en Python se implementa con el método especial `__len__`

  Entradas: Una cola (*self*)

  Salidas: Un entero *n*

- **is_empty**() $\rightarrow$ b

  Objetivo: Determinar si la cola está vacía

  Entradas: Una cola (*self*)

  Salidas: Un booleano *b* con valor `True` si la cola está vacía, `False` en otro caso

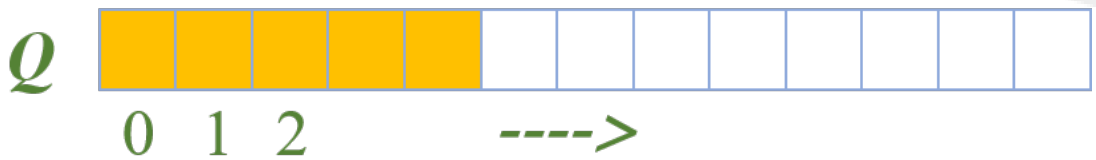
Ejemplo de uso

Operación	Retorno	1º<-Cola<-último
`Q.enqueue(5)`	-	`[5]`
`Q.enqueue(3)`	-	`[5, 3]`
`len(Q)`	2	`[5, 3]`
`Q.dequeue()`	5	`[3]`
`Q.is_empty()`	False	`[3]`
`Q.dequeue()`	3	`[]`
`Q.is_empty()`	True	`[]`
`Q.dequeue()`	"error"	`[]`
`Q.enqueue(7)`	-	`[7]`
`Q.enqueue(9)`	-	`[7, 9]`
`Q.first()`	7	`[7, 9]`
`Q.enqueue(4)`	-	`[7, 9, 4]`
`len(Q)`	3	`[7, 9, 4]`
`Q.dequeue()`	7	`[9, 4]`

Implementación

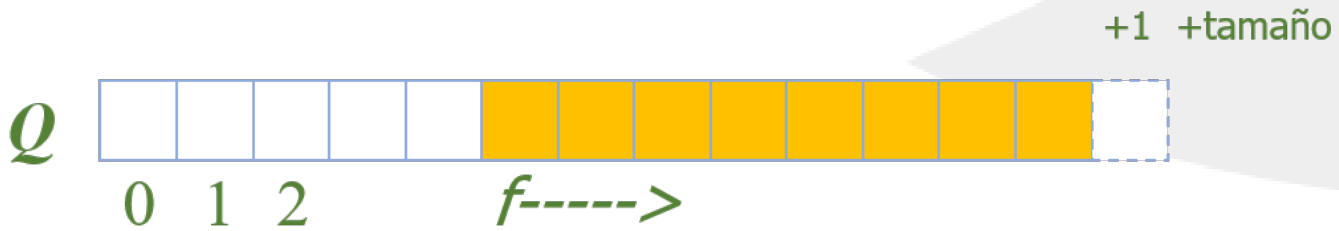
Implementación basada en array I

- Basada en una lista Python.
  - Añadir un elemento e a la cola (*enqueue*) se codifica como una llamada a `append(e)`.
  - Eliminar un elemento e de la cola (*dequeue*) se codifica como una llamada a `pop(0)`, para extraer el primer elemento de la lista.
- Ineficiente: `pop(0)` obliga a desplazar elementos a la izquierda para "cubrir" el hueco.
- Exactamente igual que si se inserta por el principio ("añadir a la cola) y se elimina por el final ("extraer de la cola").



Implementación basada en array II

- Evitar la llamada a `pop(0)`. Reemplazar el elemento desencolado con `None`, y mantener un variable *apuntadora* al índice del elemento actualmente al frente de la cola.
- Desventajas:
  - El tamaño de la lista crece a razón del número de operaciones `enqueue`, no del número de elementos.
  - El coste computacional del desplazamiento de elementos para recuperar espacio sin usar.



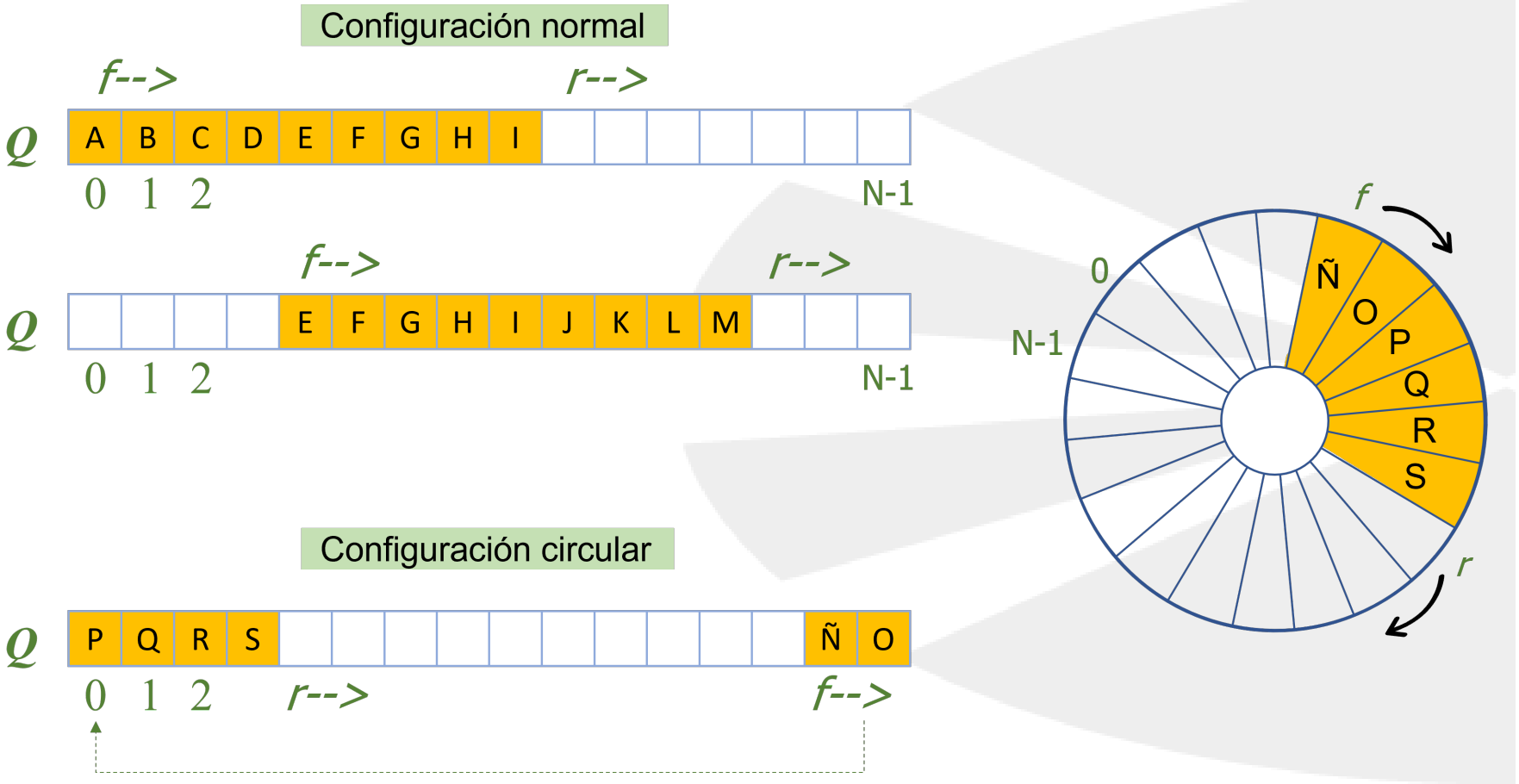
Implementación basada en array III

- Limitar el tamaño de la lista.
- Usar dos variables *apuntadoras*, `f` y `r`, a los índices de la lista donde se almacenan el primer y último elemento de la cola.
- Desventaja: los apuntadores siempre crecen y quedan posiciones sin ocupar a su izquierda.
- Solución: Desplazar los elementos cuando el índice del final de la cola iguale el tamaño de la lista.
- Desventaja: Coste computacional del desplazamiento de elementos.

Implementación basada en array (*circular*) IV

- Se define un tamaño máximo inicial para la cola (array).
- Dos variables `f` y `r` apuntan a las posiciones (índices) de la lista que contienen al primer y último elemento de la cola.
- Añadir elementos incrementa el índice `r`.
- Eliminar elementos incrementa el índice `f`.
- Cualquier apuntador que alcanza el final del array, se inicializa a 0, haciendo que la cola no tenga fin (circular).

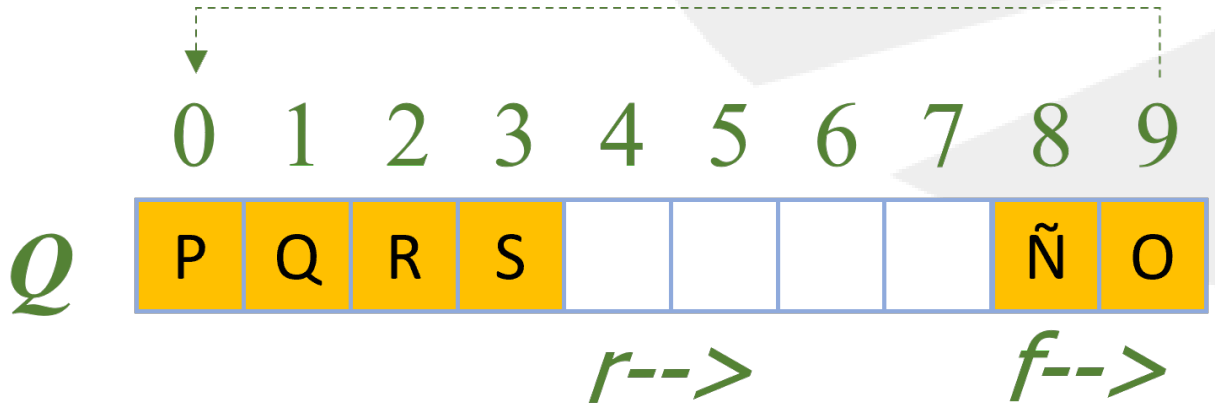
Implementación basada en array (*circular*) IV



Implementación basada en array (*circular*)

- Eliminar (añadir) un ítem, "avanza" el apuntador `f` (`r`).
- `f = (f + 1) % N`, siendo N el tamaño del array.
- Ejemplo para N=10:

f	Nuevo valor de f
0	$(0+1)\,\%10 = 1$
...	...
7	$(7+1)\,\%10 = 8$
...	...
9	$(9+1)\,\%10 = 0$



Implementación basada en array (*circular*)

```
class ArrayQueue:
  DEFAULT_CAPACITY = 10

  def __init__(self):
    self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
    self._size = 0
    self._front = 0
```

- `data`: instancia lista. Inicialmente es una lista de tamaño reducido (la cola tendría tamaño 0).
- `size`: número de elementos almacenados en la cola (en contraposición a la longitud de la lista).
- `front`: índice en `_data` del primer elemento de la cola (si la cola no está vacía).

Implementación basada en array (*circular*)

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

def is_empty(self):
  return self._size == 0

def first(self):
  if self.is_empty():
    raise Empty('Queue is empty')
  return self._data[self._front]
```

- `first` devuelve el elemento situado al frente de cola.
- `_front` informa de la localización en la lista `_data` del elemento al frente (si la lista no está vacía).

Implementación basada en array (*circular*)

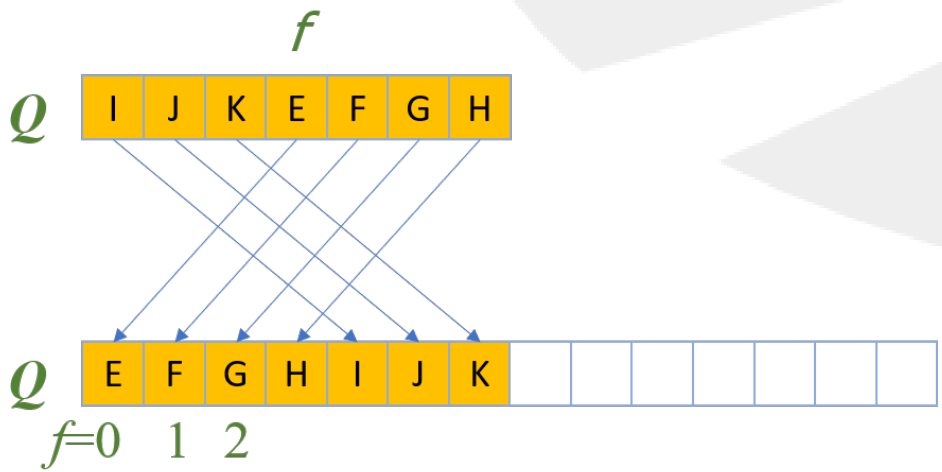
```
def enqueue(self, e):
  if self._size == len(self._data):
    self._resize(2 * len(self.data))
  avail = (self._front + self._size) % len(self._data)
  self._data[avail] = e
  self._size += 1
```

- Si se alcanza el tamaño máximo, se duplica y se reorganizan elementos.
- El índice (*avail*) del nuevo elemento se calcula añadiendo el tamaño de la cola a la posición del frente.
- Ejemplo para una cola con capacidad de 10 elementos.

$\texttt{\_front}_t$	`_size`	posiciones	`avail`	Comportamiento
5	3	5, 6, 7	8	normal
8	3	8, 9, 0	1	circular

Implementación basada en array (*circular*)

```
def _resize(self, cap):                 # cap >= len(self)
  old = self._data                      # keep existing list
  self._data = [None] * cap             # new list
  walk = self._front
  for k in range(self._size):
    self._data[k] = old[walk]
    walk = (1 + walk) % len(old)        # old size as modulus
  self._front = 0                       # front realigned
```



Implementación basada en array (*circular*)

```
def dequeue(self):
  if self.is_empty():
    raise Empty('Queue is empty')
  answer = self._data[self._front]
  self._data[self._front] = None    # help garbage

  self._front = (self._front + 1) % len(self._data)
  self._size -= 1
  return answer
```

- `_front` es el índice del valor a eliminar y devolver. Se actualiza y con ello el segundo elemento promociona al primer puesto.
- La asignación de `None` elimina la referencia al objeto extraído (Python reclamará el espacio pues no está referenciado por la lista).

Bibliografía

Bibliografia

- Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. 2013. Data Structures and Algorithms in Python (1st edition). Wiley Publishing.
- Kenneth A. Lambert. 2018. Fundamentals of Python: Data Structures (2nd edition). Cengage.

4. Colas

Programación II

Grado en Inteligencia Artificial

M. Alonso, M. Cabrero y E. Hernández

fic facultade de informática da coruña

UNIVERSIDADE DA CORUÑA