6. Listas

Programación II

Grado en Inteligencia Artificial

M. Alonso, M. Cabrero y E. Hernández

fic facultade de informática da coruña

UNIVERSIDADE DA CORUÑA

Contenidos

Definición

¿Qué es una lista?

¿Qué es una lista?

- Las pilas solo permiten insertar y borrar por un único extremo
- Las colas permiten insertar por un extremo y borrar por el otro

- Las **listas** permiten insertar y borrar en cualquier lugar
  - Aportan flexibilidad
  - Su manejo es más complejo: más operaciones, más situaciones de error, ...
  - Su implementación también es más compleja
  - Algunas operaciones pueden ser poco eficientes

¿Y las listas de Python?

El lenguaje Python incorpora un **tipo de dato concreto** `lista`:

- Implementación basada en arrays
- El acceso a los elementos almacenados se hace con índices enteros
- Problemas de eficiencia en ciertas operaciones

Nos abstraeremos de los detalles de implementación de las listas, las posiciones, etc., para definir un TAD que denominaremos **listas posicionales** (para evitar confundirlas con las listas de Python)

Listas posicionales

Trataremos con:

- Listas
- Posiciones
- Elementos que se almacenan en las listas

Ejemplos de uso:

- `pl = PositionalList()`
- `pl.add_first(10)`
- `pl.replace(pl.before(pl.last()), 28)`
-
```
for x in pl:
    print(x)
```

El TAD Lista

El TAD Lista Posicional: Valores

Una lista posicional es una secuencia lineal de elementos organizados por su posición

- Una lista posicional puede estar vacía
- Si no está vacía:
  - Hay un único elemento que ocupa la primera posición
  - Hay un único elemento que ocupa la última posición
  - 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 creadoras

- **PositionalList**()

  Objetivo Crear una lista posicional vacía
  Nota: en Python se traduce automáticamente en una llamda `self.__init__()`

  Salida Una lista posicional vacía

Operaciones mutadoras

- **add_last**(e) $\rightarrow$ p

  Objetivo: Añadir el elemento *e* al final

  Entradas: Una lista posicional (*self*) y un elemento *e*

  Salidas: La lista posicional con el elemento *e* añadido al final;
  La posición *p* del elemento *e* en la lista

- **add_first**(e) $\rightarrow$ p

  Objetivo: Añadir el elemento *e* al inicio

  Entradas: Una lista posicional (*self*) y un elemento *e*

  Salidas: La lista posicional con *e* añadido al inicio;
  La posición *p* del elemento *e* en la lista

  Poscondición: Todas las posiciones de la lista dejan de ser válidas

Operaciones mutadoras

- **add_before**(p, e) $\rightarrow$ p'

  Objetivo Añadir el elemento *e* antes de la posición *p*

  Entradas Una lista posicional (*self*), una posición *p* y un elemento *e*

  Salidas Lista posicional con *e* añadido antes de *p*; La posición *p'* que ocupa el elemento *e* en la lista

  Precondición *p* es una posición válida de la lista posicional

  Poscondición La posición *p* y las siguientes dejan de ser válidas

Operaciones mutadoras

- **add_after**(p, e) $\rightarrow$ p'

  Objetivo Añadir el elemento *e* después de la posición *p*

  Entradas Una lista posicional (*self*), una posición *p* y un elemento *e*

  Salida La lista posicional con *e* añadido después de *p*;
  La posición *p'* que ocupa el elemento *e* en la lista

  Precondición *p* es una posición válida de la lista posicional

  Poscondición La posición *p* y siguientes dejan de ser válidas

Operaciones mutadoras

- **replace**(p, e)

Objetivo	Reemplazar el elemento en la posición *p* por *e*
Entradas	Una lista posicional (*self*), una posición *p* y un elemento *e*
Salidas	La lista posicional con *e* añadido en posición *p*; El elemento que anteriormente ocupaba la posición *p*
Precondición	*p* es una posición válida de la lista posicional

Operaciones mutadoras

- **delete**(p)

  Objetivo: Elimina el elemento en la posición *p*

  Entradas: Una lista posicional (*self*) y una posición *p*

  Salidas: La lista posicional sin el elemento en la posición *p*;
  El elemento que anteriormente ocupaba la posición *p*

  Precondición: *p* es una posición válida de la lista posicional

  Poscondición: La posición *p* y siguientes dejan de ser válidas

Operaciones observadoras

- **first**() → p
  - Objetivo: Devuelve la primera posición
  - Entradas: Una lista posicional (*self*)
  - Salidas: La primera posición *p* o `None` si está vacía
- **last**() → p
  - Objetivo: Devuelve la última posición
  - Entradas: Una lista posicional (*self*)
  - Salidas: La última posición *p* o `None` si está vacía
- **is_empty**() → b
  - Objetivo: Determinar si la lista está vacía
  - Entradas: Una lista posicional (*self*)
  - Salidas: un booleano *b* `True` si lista vacía, `False` en otro caso

Operaciones observadoras

- **before**(p) $\rightarrow$ p'

  Objetivo: Devuelve la posición anterior a *p*

  Entradas: Una lista posicional (*self*) y una posición *p*

  Salidas: La posición *p'* anteriopr a *p* o `None` si es la primera

  Precondición: *p* es una posición válida de la lista posicional

- **after**(p) $\rightarrow$ p'

  Objetivo: Devuelve la posición posterior a *p*

  Entradas: Una lista posicional (*self*) y una posición *p*

  Salidas: La posición *p'* posterior a *p* o `None` si es la última

  Precondición: *p* es una posición válida de la lista posicional

Operaciones observadoras

- **get_element**(p) $\rightarrow$ e

  Objetivo: Devuelve el elemento de la lista en la posición *p*

  Entradas: Una lista posicional (*self*) y una posición *p*

  Salidas: El elemento *e* en posición *p*

  Precondición: *p* es una posición válida de la lista posicional

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

  Objetivo: Devuelve el número de elementos de la lista

  Nota: en Python se traduce automáticamente en una llamada a `__len__`

  Entradas: Una lista posicional (*self*)

  Salidas: El número *n* de elementos de la lista

Operaciones observadoras iteradoras

- **iter**() $\rightarrow$ I

  Objetivo: Devuelve un iterador para los elementos de la lista
  Nota: en Python se traduce automáticamente en una llamada a `__iter__`

  Entradas: Una lista posicional (*self*)

  Salidas: Un iterador *I*

Implementación con arrays

Implementación con arrays

Poco más que un mero interfaz con las listas de Python

```
1  class ArrayPositionalList:
2    def __init__(self):
3      self._data = []
4
5    def add_last(self, e):
6      self._data.append(e)
7      return len(self._data) - 1
8
9    def add_first(self, e):
10     self._data.insert(0,e)
11     return 0
12
13   def add_before(self, p, e):
14     if 0 <= p < len(self._data):
15         self._data.insert(p, e)
16         return p
17     else:
18         raise IndexError('p no válido')
```

```
1
2  def add_after(self, p, e):
3    if 0 <= p < len(self._data):
4        self._data.insert(p+1, e)
5        return p+1
6    else:
7        raise IndexError('p no válido')
8
9  def delete(self, p):
10   return self._data.pop(p)  #exception?
11
12 def replace(self, p, e):
13   old_value = self._data[p] #exception?
14   self._data[p] = e
15   return old_value
```

Implementación con arrays

```
1  def first(self):
2    if self.is_empty():
3        return None
4    else:
5        return 0
6
7  def last(self):
8    if self.is_empty():
9        return None
10   else:
11       return len(self._data) - 1
12
13 def before(self, p):
14   if 0 < p < len(self._data):
15       return p-1
16   elif p == 0:
17       return None
18   else:
19       raise IndexError('p inválido')
```

```
1  def after(self, p):
2    if 0 <= p < (len(self._data) - 1):
3        return p+1
4    elif p == (len(self._data) - 1):
5        return None
6    else:
7        raise IndexError('p inválido')
8
9  def get_element(self, p):
10   if 0 <= p < len(self._data):
11       return self._data[p]
12   else:
13       raise IndexError('p inválido')
14
15 def __len__(self):
16   return len(self._data)
17
18 def is_empty(self):
19   return (len(self._data)) == 0
20
21 def __iter__(self):
22   return iter(self._data)
```

Implementación con una secuencia enlazada

Implementación con una secuencia enlazada

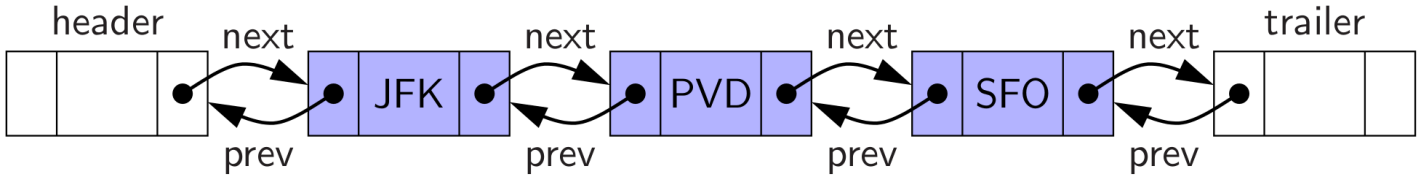
- En secuencias basadas en array, el índice entero sirve para localizar un elemento o el punto de inserción o borrado de un elemento.
- En secuencias enlazadas no sirven:
  - No hay acceso eficiente por índice: se necesita recorrer toda la lista contando elementos.
  - No son una buena abstracción: cambian debido a inserciones o borrados en posiciones previas en la secuencia.

Implementación con una secuencia enlazada

Implementación con una secuencia enlazada

- Debemos definir una clase interna para las posiciones: `Position`
- El resto es poco más que una interfaz con una secuencia enlazada:

```
1 class LinkedPositionalList(_DoublyLinkedBase):
```



- En este caso hemos utilizado como base la secuencia doblemente enlazada, pero puede sustituirse por otra secuencia (enlazada simple, circular, . . .)

La clase Position

```
class Position:
    """An abstraction representing the location of a single element.

    Note that two position instaces may represent the same inherent
    location in the list.  Therefore, users should always rely on
    syntax 'p == q' rather than 'p is q' when testing equivalence of
    positions."""

    def __init__(self, container, node):
      """Constructor should not be invoked by user."""
      self._container = container      # la secuencia enlazada
      self._node = node                # el nodo indicado por la posición

    def element(self):
      """Return the element stored at this Position."""
      return self._node._element

    def __eq__(self, other):
      """Return True if other is a Position representing the same location."""
      return type(other) is type(self) and other._node is self._node

    def __ne__(self, other):
      """Return True if other does not represent the same location."""
      return not (self == other)       # la negación de __eq__
```

Métodos auxiliares de lista para manejar posiciones

```
def _validate(self, p):
    """Return position's node, or raise error if invalid."""
    if not isinstance(p, self.Position):
      raise TypeError('p must be proper Position type')
    if p._container is not self:
      raise ValueError('p does not belong to this container')
    if p._node._next is None: # Removed nodes sets_next to None
      raise ValueError('p is no longer valid')
    return p._node
```

- Cuando un método acepta una posición como parámetro, se comprueba su validez y se determina el nodo asociado.
- Internamente la posición apunta a un nodo y a la instancia de la lista que contiene el nodo específico.
- Con la referencia se puede detectar el envío de una instancia de posición que no pertenece a la lista.

Métodos auxiliares de lista para manejar posiciones

```
def _make_position(self, node):
    """Return Position instance for given node (or None if sentinel)."""
    if node is self._header or node is self._trailer:
      return None                  # comprobación de límites
                                   # pensado para first() y last()
    else:
      return self.Position(self, node)        # crea una posición válida

# override de la versión heredada para devolver Position en lugar de Node
def _insert_between(self, e, predecessor, successor):
    """Add element between existing nodes and return new Position."""
    node = super()._insert_between(e, predecessor, successor)
    return self._make_position(node)
```

Operaciones mutadoras

```
# No se necesita __init__ porque se hereda de _DoublyLinkedBase

def add_first(self, e):
    return self._insert_between(e, self._header, self._header._next)

def add_last(self, e):
    return self._insert_between(e, self._trailer._prev, self._trailer)

def add_before(self, p, e):
    original = self._validate(p)                    # exception?
    return self._insert_between(e, original._prev, original)

def add_after(self, p, e):
    original = self._validate(p)                    # exception?
    return self._insert_between(e, original, original._next)

def replace(self, p, e):
    original = self._validate(p)                    # exception?
    old_value = original._element
    original._element = e
    return old_value
```

Operaciones mutadoras

```
def delete(self, p):
    original = self._validate(p)          # exception?
    return self._delete_node(original)    # call to inherited method
                                          # that returns the stored element
                                          # and sets the _next field to None val
```

Operaciones observadoras

```
def first(self):
    return self._make_position(self._header._next)

def last(self):
    return self._make_position(self._trailer._prev)

def before(self, p):
    node = self._validate(p)          # exception?
    return self._make_position(node._prev)

def after(self, p):
    node = self._validate(p)          # exception?
    return self._make_position(node._next)

# __len__ se hereda de _DoublyLinkedBase

def get_element(self, p):
    self._validate(p)
    return p.element()
```

Iterador

```
def __iter__(self):
  cursor = self.first()
  while cursor is not None:
    yield self.get_element(cursor)
    cursor = self.after(cursor)
```

- Este iterador es realmente independiente de la implementación (sólo usa las operaciones definidas para la lista posicional)
- Por tanto se podría utilizar también en la implemenatción con arrays
- Nos sirve como ejemplo típico de cómo se recorre una lista posicional

Ejemplos de uso

Uso de listas posicionales

- Son independientes de la implementación:

```
from array_positional_list import ArrayPositionalList as PositionalList
from linked_positional_list import LinkedPositionalLIst as PositionalLis
```

- Debe respetarse siempre la especificación de los operadores
- Especial cuidado en satisfacer siempre las precondiciones al llamar a los operadores

Mostrar una lista

Igual que mostrar una lista de Python:

```
def print_list(pl):
    """ Show a positional list in a terminal. """
    print("[", end=" ")
    for x in pl:
        print(x, end=" ")
    print("]")
```

Copiar una lista

```
def copy_list(pl):
    """ Returns a copy of a positional list."""
    result = PositionalList()
    for x in pl:
        result.add_last(x)
    return result
```

Mostrar una lista "al revés"

Ya no podemos utilizar el iterador, que siempre va "hacia adelante".
Recurrimos a un bucle clásico

```
def print_list_reversed(pl):
    """ Show a positional list, in reverse order."""
    print("[", end=" ")
    marker = pl.last()                  # None if empty list
    while marker != None:
        print(pl.get_element(marker), end=" ")
        marker = pl.before(marker) # None if first position
    print("]")
```

Buscar un elemento

```
def search_element(pl, e):
    """ Return the position of the first instance of e
        in a positional list pl. Return None if e
        is not an element of pl."""
    marker = pl.first()                 # None if empty list
    while marker != None and pl.get_element(marker) != e:
        marker = pl.after(marker)  # None if last positio
    return marker
```

Listas ordenadas

Listas ordenadas

- Los elementos no se pueden insertar en cualquier posición
- Se eliminan las operaciones `add_*`
- En su lugar, tenemos una única operación `add(e)`: busca la posición adecuada de inserción para que la lista se mantenga ordenada
- En caso de elementos repetidos, se añade antes de los existentes
- `replace` necesita mantener el orden

Especificación

- **add**(e)

  Objetivo: Añadir el elemento *e* en orden

  Entradas: Una lista posicional ordenada (*self*) y un elemento *e*

  Salida: La lista posicional ordenada con *e* añadido
  La posición que ocupa el elemento *e* en la lista

  Poscondición: La lista está ordenada

Implementación con arrays

```
def add(self, e):
    """Insert element e into list and return new Position."""
    if self.is_empty() or e > self._data[len(self._data) - 1]:
        self._data.append(e)
        return len(self._data) - 1
    else:
        # incrementamos el tamaño de la lista de datos
        self._data.append(None)
        # recorremos los elementos originales desde el final
        # hacia el principio
        p = len(self._data) - 2
        while p >= 0 and self._data[p] >= e:
            self._data[p + 1] = self._data[p]
            p -= 1
        self._data[p + 1] = e     # insertar e en su posición
        return p + 1
```

Implementación con secuencias enlazadas

Donde _add_first, _add_last y _add_after son ahora métodos privados

```
def add(self, e):
    """Insert element e into list and return new Position."""
    if self.is_empty() or e > self.get_element(self.last()):
        return self._add_last(e)
    else:
        # recorremos del final hacia el principio
        p = self.last()
        while p != None and self.get_element(p) >= e:
            p = self.before(p)
        if p == None:
            return self._add_first(e)
        else:
            return self._add_after(p, e)
```

Implementación con arrays/secuencias enlazadas

Modificación independiente de la implementación subyacente

```
def replace(self, p, e):
  """Replace the element at Position p with e.

  Return the element formerly at Position p.
  """
  old_value = self.delete(p)   # temporarily store old element
  self.add(e)                  # add new element
  return old_value             # return the old element value
```

Secuencias basadas en arrays vs enlazadas

Ventajas de la representación basada en arrays

- El acceso a un elemento es directo basado en índice (en las enlazadas es secuencial desde el primer o último elemento hasta llegar al de la posición deseada).
- Generalmente usan menos memoria: sólo para elementos (en las enlazadas para elementos y referencia a nodos).

Ventajas de la representación enlazada

- Uso más eficiente de la memoria: su tamaño no está predefinido, crecen dinámicamente.
- Las operaciones de inserción y eliminación son óptimas en tiempo. No así las basadas en array ya que las llamadas subyacentes a los métodos `insert()` y `pop()` requieren desplazar todos los elementos posteriores.

Bibliografía

Bibliografia

- Capítulo 7 de: Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. 2013.Data Structures and Algorithms in Python (1st edition). Wiley Publishing.
- Capítulo 9 de: Kenneth A. Lambert. 2018. Fundamentals of Python: Data Structures (2nd edition). Cengage.

6. Listas

Programación II

Grado en Inteligencia Artificial

M. Alonso, M. Cabrero y E. Hernández

fic facultade de informática da coruña

UNIVERSIDADE DA CORUÑA