UNIVERSIDAD DE MURCIA

Tipos de Datos Abstractos Con Orden Lineal

Tecnología de la Programación

L. Daniel Hernández < *ldaniel@um.es* >

Dpto. Ingeniería de la Información y las Comunicaciones
Universidad de Murcia
4 de octubre de 2023

$$p(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n$$

TDA Polinomio

- Usa: naturales (exponentes), reales (coeficientes)
- Operaciones:
  - `Crear (real *coef) : Polinomio`
    El k-ésimo coeficiente representa al k-ésimo monomio.
  - `suma (Polinomio f, Polinomio g) : Polinomio`
    Retorna la suma de polinomios
  - `termino (Polinomio f, int k) : real`
    Retorna el coeficiente del k-ésimo monomio
  - ...

```
class  Polinomio:
    grado: int
    coef: list

    def suma(p: Polinomio):
        ...
```

Índice de Contenidos

Desarrollo

Qué son los TDAs sin relación de orden

- Los TDA con **ordenación lineal** son aquellos que contienen a un conjunto de objetos pero donde cada objeto tiene exactamente un predecesor inmediato o un sucesor inmediato.
- Destacan el primer objeto (que no tiene predecesor) y el último objeto (que no tiene sucesor).
- **Matemáticamente** quedan definidos mediante una relación binaria $a \preceq b$, que será una ordenación lineal sii cumple las siguientes propiedades:
  Completa o total. $\forall\ a, b$, o bien $a \preceq b$ o bien $b \preceq a$,
  Transitiva. si $a \preceq b$ y $b \preceq c$, esto implica que $a \preceq c$, y
  Antisimétrica. Si se cumple $a \preceq b$ y $b \preceq a$, se sigue que $a = b$
- **Las operaciones** que se pueden lleva a cabo en este tipo de contenedores son:
  - **Acceder** al k-ésimo objeto del orden. En particular, al primer o último objeto en el orden lineal,
  - **Recorrer** todos los objetos del contenedor en dicho orden.
  - **Dado un objeto** a que puede, o no, estar en el contenedor,
    - **buscar** el objeto predecesor inmediato o el sucesor inmediato en el contenedor.
    - **contar** todos los objetos que le preceden o todos los que le suceden.
  - **Inserta** un nuevo objeto o reemplazar el objeto en la k-ésima posición,
  - Dada una referencia al k-ésimo objeto, **insertar** un nuevo objeto antes o después de ese objeto; o eliminar el objeto antes o después del k-ésimo objeto.

Desarrollo

Listas - I

- La **lista abstracta** se define para objetos que se quieren ordenar explícitamente. Hay un primer elemento que se coloca al inicio del contenedor y un último elemento que está al final de la lista. Los elementos centrales se colocan de acuerdo al orden lineal definido.
- Operaciones
  - `List(): Lista`. Crea una nueva lista, inicialmente vacía.
  - `len(): int`. Retorna la longitud o número de elementos en la lista.
    Para una lista cualquiera $< a_0, a_1, \ldots, a_{n-1} >$ retornará el valor n.
  - `contains(value): bool`. Indica si el valor `value` se encuentra na la lista.
  - `getItem(pos): value`. Retorna el elemento o valor almacenado en la posición `pos`.
    Para una lista cualquiera $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ retornará el valor a_{pos}.
  - `setItem(pos, value): None`. Sustituye el `pos`-ésimo valor de la lista por `value`.
    Para una lista cualquiera $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se modificará la lista a la secuencia $< a_0, \ldots, value, \ldots, a_{n-1} >$.

Listas - II

- `insertItem(pos, value): None`. Inserta el `pos`-ésimo valor de la lista por `value`.
  Para una lista cualquiera $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se modificará la lista a la secuencia $< a_0, \ldots, value, a_{pos}, \ldots, a_{n-1} >$.
- `removeItem(pos): None`. Elimina el elemento de la posición dada, si existe en la lista.
  Dada una lista cualquiera $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se modificará la lista a la secuencia $< a_0, \ldots, a_{pos-1}, a_{pos+1}, \ldots, a_{n-1} >$.
- `clear(): None`. Limpia la lista. Se convierte en una lista vacía.
  Modifica cualquier lista a la lista $<>$.
- `isEmpty(): bool`. Indica si la lista está vacía o no.
- `first(): pos`. Retorna la posición del primer elemento de la lista. Si la lista está vacía retornará `last()`
  Dada una lista $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se retornará la posición donde se localiza a_0.
- `last(): pos`. Retorna la posición del último elemento de la lista. Si la lista está vacía retornará `last()`
  Dada una lista $< a_0, \ldots, a_{pos}, \ldots, a_{n-1} >$ se retornará la posición donde se localiza a_{n-1}.

Listas - III

- `next(pos): pos`. Retorna la posición del elemento siguiente al elemento de la posición dada.
  Dada una lista retornará la posición donde se localiza el elemento a_{pos+1}.
- `previous(pos): pos`. Retorna la posición del elemento anterior al elemento de la posición dada.
  Dada una lista retornará la posición donde se localiza el elemento a_{pos-1}.
- `iterator(): Lista`: Crea y retorna un iterador para la lista.

Profundizamos en ...

Con más detalle ..

Implementación de Listas: Arrays

- **Arrays**: cada elemento del array contiene un elemento de la lista, indexados/ordenados de acuerdo al orden lineal.
- Un array 'puro' presenta inconvenientes de gestión de memoria.
- Alternativamente se usa esta representación.

```
struct rep
    int pos; # la primera posición vacía
    int len; # longitud de la lista
    object[] list;  # la lista
```

  - La **función de abstracción**: $Abst$: **rep** $\longrightarrow \mathcal{A}$ la definiremos como
    $Abst$(r) =<`r.list[0], r.list[1], ..., r.list[len-1]`>
  - El **invariante de la representación** I : **rep** $\longrightarrow \mathbb{B}$, lo definimos de esta manera:

$$I(\text{r}) = \begin{cases} \forall p, p < \text{r.pos} \Rightarrow \text{r.list[r.p]} \neq \mathit{NULL}\ \mathit{and} \\ \forall p, p \geq \text{r.pos} \Rightarrow \text{r.list[r.p]} = \mathit{NULL}\ \mathit{and} \\ \text{r.list} \neq \mathit{NULL}\ (\text{r.len} > 0) \end{cases}$$

Con este invariante hay una lista que no se puede representar ¿cuál?

Con más detalle ..

Implementación de Listas: Nodos

- **Nodo**: Estructura con 1 valor y 1 o más referencias a nodos.
- **Nodo Simple**: Estructura con 1 valor y 1 referencia.
- **Estructura enlazada**: Estructura de datos formada por nodos.
- **Estructura simplemente enlazada**: Estructura enlazada formada por nodos simples.
- Una lista se puede representar mediante una estructura simplemente enlazada.
- **Función de abstracción:** $Abst$: **node** $\longrightarrow \mathcal{A}$

  $Abst(\texttt{r}) =$<`r.valor, r.next.valor, ..., r.next....next.valor`>
- El **invariante de la representación** I : **node** $\longrightarrow \mathbb{B}$, que se define así:
  - Consta de una colección de nodos donde cada uno tiene una referencia a otro nodo (cuyo campo llamaremos `siguiente` o `next`).
  - En dos nodos diferentes la referencia `next` son diferentes.
  - Se tiene una variable externa que hace referencia a aquel nodo tal que a partir de él se puede recorrer todos los elementos de la lista pasando por la referencia de `next` de cada uno.

Operaciones básicas con Estructuras enlazadas - I

1. **Recorrer los nodos.**

```
actual = primer nodo de la lista
mientras actual is not None:
  trabajar con actual
  actual = actual.next
```

2. **Buscar un nodo.**

```
actual = primer nodo de la lista
mientras actual is not None y actual.value != target:
  actual = actual.next
trabajar con actual
```

Operaciones básicas con Estructuras enlazadas - II

3. **Añadir un nodo como siguiente de otro.**

```
pos = referencia # Pondremos uno delante de este
nuevo_nodo = Node(VALUE)
nuevo_nodo = pos.next
pos.next = nuevo_nodo
```

4. **Eliminar el nodo siguiente de otro.**

```
pos = referencia # Eliminaremos el siguiente a este
borrar = pos.next
pos.next = borrar.next
gc.collect()
```


Modificaciones sobre una Estructuras Enlazada Simple I

- Considerar un **nodo cabecera**. El nodo cabecera NO forma parte de la lista, pero apunta al primer elemento de la lista.

```
struct node  # Estructura del nodo
    TD valor
    node next

struct List  # Lista simplemente enlazada
  # Descomentar el adecuado, en su caso:
  # node head # Apunta a una cabecera
  # node list # Apunta al primer elemento de la lista
```

- Para recorrer la lista desde cualquier punto, considerar **listas circulares**.

Modificaciones sobre una Estructuras Enlazada Simple II

- Si se quiere añadir **elementos al final**, tener un campo `ultimo` al último elemento de la lista.

```
struct node  # Estructura del nodo
    TD valor
    node next

struct List  # Lista simplemente enlazada
  node list
  node last # con referencia al último elemento de la lista
```

- Si hay que hacer muchos recorridos, considerar una **lista doblemente enlazada**.

```
struct node
   TD valor
   node next
   node previous
```

Profundizamos en ...

Con más detalle ..

Búsqueda en listas

- Un **algoritmo de búsqueda** es aquel que está diseñado para localizar un elemento con ciertas propiedades dentro de una estructura de datos. Por ejemplo, encontrar el registro correspondiente a cierta canción en un array de canciones.
- Dependiendo de cómo se encuentren ordenados los elementos de la colección, podemos aplicar uno de los dos siguientes algoritmos:
  - o **búsqueda secuencial**, cuando los elementos de la colección no están ordenados
  - o **búsqueda binaria**, para cuando los elementos están ordenados.

Con más detalle ..

Ordenación en listas

- Un **algoritmo de ordenación** u ordenamiento es aquel que está diseñado para ordenar los elementos de una lista.
- Será necesario aclarar qué se entiende por la expresión `vec[pos1]<vec[pos2]`: operación construida por el programador.
- Algoritmos conocidos de ordenación, y que se consideran **ya aprendidos**.
  - **Ordenación de Burbuja.** Se revisa la lista una y otra vez, intercambiando los valores de dos posiciones consecutivas si dichos valores no están ordenados.

    ```
    4 5 3 2 (swap)
    4 3 5 2
    4 3 2 5
    ```
  - **Ordenación por Selección.** Selecciona el elemento más pequeño de la lista y lo coloca en su posición. Después busca el siguiente más pequeño de entre el resto de la lista y lo coloca en su sitio, y así se repite una y otra vez.

    ```
    4 5 3 2 (min = 2)
    2 3 5 4
    ```
  - **Ordenación por inserción.** En el paso k, se considera el valor de la posición k y el anterior (inicialmente en la posición k-1). Mientras que el anterior sea menor que el valor se intercambian los valores (actualizando en cada bucle el valor de anterior).

    ```
    4 5 3 2 (k=2)
    3 4 5 2
    ```

Con más detalle ..

Ordenación por Casilleros y por Mezclas

- **Ordenación por Casilleros.**
  1. Crear una colección de casilleros vacíos, que ya está "ordenados" entre sí.
  2. Colocar cada elemento a ordenar de la lista en un único casillero.
  3. Ordenar los elementos de cada casillero según el algoritmo que se considere más adecuado.
  4. Devolver los elementos de cada casillero "concatenados".
- **Ordenación por Mezclas.** Dada una lista,
  1. Si la lista tienen un elemento retornar la lista.
  2. En otro caso, dividir (recursivamente) la lista en dos partes 'simétricas'.
  3. Mezclar las dos listas:
     - Mientras que alguna de las listas tenga elementos hacer:
       1) Se comparan los dos primeros elementos de la lista y se quita el más pequeño.
       2) El más pequeño se añade a la lista solución.
     - Añadir los elementos de la lista no vacía a la lista solución.
  4. Retornar la lista solución.

Desarrollo

Pilas

- Una **pila** es una lista con el criterio LIFO.
- Operaciones
  - `Stack() : Stack`. Crea una nueva pila, inicialmente vacía.
  - `peek() : value`. Retorna el valor del tope. También se suele usar la signatura `top() : value`.
    Para una pila $< a_0, a_1, \ldots, >$ retornará el valor a_0.
  - `pop() : value`. Retorna el valor del tope y además borra el primer nodo de la pila.
    Para una pila $< a_0, a_1, a_2, \ldots, >$ retornará el valor a_0 y la nueva pila es $< a_1, a_2, \ldots, >$.
  - `push(value) : None`. Inserta un nuevo nodo en el tope de la lista.
    Para una pila $< a_0, a_1, a_2, \ldots, >$ y un valor *value* la pila se modifica para conseguir la pila $< \mathit{value}, a_0, a_1, a_2, \ldots, >$.
  - `len() : int`. Retorna el número de elementos de la pila.
  - `isEmpty() : Bool`. Indica si la pila está vacía o no.
  - `clear() : None`. Limpia la pila y la deja sin elementos.
- MUY IMPORTANTE: No se puede acceder directamente a ningún elemento central.

Desarrollo

Colas y Colas de Prioridad

- Una **cola** es una lista con el criterio FIFO.
- Operaciones
  - `Queue() : Queue`. Crea una nueva cola, inicialmente vacía.
  - `peek() : value`. Retorna el valor del primer elemento de la lista, pero no lo borra. También es usual esta signatura `front() : value`.
  - `dequeue() : value`. Retorna el primer elemento de la cola borrandolo de la cola. También es usual esta signatura `top() : value`.
    Para la cola $< a_0, a_1, \ldots, >$ retornará el valor a_0. Se genera un error si la cola está vacía.
  - `enqueue(value) : None`. Añade un nuevo elemento al final de la cola
    Para la cola $< a_0, a_1, \ldots, a_{n-1} >$ se modificará a la lista
    $< a_0, a_1, \ldots, a_{n-1}, value >$
  - `len() : int`. Retorna el número de elementos de la cola.
  - `isEmpty() : Bool`. Indica si la cola está vacía o no.
  - `clear() : None`. Limpia la cola y la deja sin elementos.
- Una **cola de prioridad** es una cola pero que el encolar a un elemento nuevo lo coloca en la posición que le corresponda en el orden. Las demás operaciones son iguales.
- MUY IMPORTANTE: No se puede acceder directamente a ningún elemento central. Acceder directamente es suspender el examen.

Desarrollo

Ordenación

```
def search(initial, goal_test, successors):
  # Nodo = {estado, nodo_padre}

  frontera = [Nodo(initial, None)]  # LISTA de nodos que
                    # contienen a los estados a expandir
  explorados = {initial} # CONJUNTO de estados analizados

  mientras que la frontera no esté vacía:
     nodo_actual = extraer un nodo de frontera (y
      ↪  borrarlo) # PILA, COLA (de prioridad)
     estado_actual = estado del nodo_actual
     si goal_test(estado_actual):
        retornar camino solución para el nodo_actual
     para cada estado de successors(estado_actual):
         si estado está en explorados:
             pasar al siguiente estado
         añadir estado a explorados
         añadir el nodo(estado, nodo_actual) a frontera
  retornar None
```