Programación II

Estructuras de Datos y Abstracción de Datos

Universidad Nacional de Rosario.
Facultad de Ciencias Exactas, Ingeniería y Agrimensura.

Introducción

Los tipos de datos definidos en unidades anteriores son **tipos de datos concretos**. Por ejemplo: `Point` (punto) se definió como un par ordenado de flotantes; `Rectangle` (rectángulo) se definio como un punto y dos flotantes.

Introducción

Los tipos de datos definidos en unidades anteriores son **tipos de datos concretos**. Por ejemplo: `Point` (punto) se definió como un par ordenado de flotantes; `Rectangle` (rectángulo) se definio como un punto y dos flotantes.

Otra forma de definir tipos de datos es **mediante sus operaciones**: enumerándolas e indicando su comportamiento (es decir, cuál es su resultado esperado).

Introducción

Los tipos de datos definidos en unidades anteriores son **tipos de datos concretos**. Por ejemplo: `Point` (punto) se definió como un par ordenado de flotantes; `Rectangle` (rectángulo) se definio como un punto y dos flotantes.

Otra forma de definir tipos de datos es **mediante sus operaciones**: enumerándolas e indicando su comportamiento (es decir, cuál es su resultado esperado).

Esta manera de definir datos se conoce como **Tipos Abstractos de Datos** o **TADs** (o *tipo de datos abstracto*, ambas provienen de la traducción del término en inglés *abstract data type*).

Introducción a Tipos Abstractos de Datos

Un **tipo abstracto de datos** o **TAD** es:

Introducción a Tipos Abstractos de Datos

Un **tipo abstracto de datos** o **TAD** es:

- una **colección de datos**

Introducción a Tipos Abstractos de Datos

Un **tipo abstracto de datos** o **TAD** es:

- una **colección de datos**
- acompañada de un **conjunto de operaciones para manipularlos**, de forma tal que queden ocultas la representación interna del nuevo tipo y la implementación de las operaciones, para todas las unidades de programa que lo utilice.

Introducción a Tipos Abstractos de Datos

Un **tipo abstracto de datos** o **TAD** es:

- una **colección de datos**
- acompañada de un **conjunto de operaciones para manipularlos**, de forma tal que queden ocultas la representación interna del nuevo tipo y la implementación de las operaciones, para todas las unidades de programa que lo utilice.

Introducción a Tipos Abstractos de Datos

¿Por qué son **útiles los tipos abstractos de datos**?

- Los programas que los usan hacen referencia a las operaciones que tienen, no a la representación, y por lo tanto ese programa sigue funcionando si se cambia la representación.

Introducción a Tipos Abstractos de Datos

¿Por qué son **útiles los tipos abstractos de datos**?

- Los programas que los usan hacen referencia a las operaciones que tienen, no a la representación, y por lo tanto ese programa sigue funcionando si se cambia la representación.
- Simplifican el desarrollo de algoritmos utilizando las operaciones del tipo abstracto de dato, sin importar cómo las mismas son implementadas.

Introducción a Tipos Abstractos de Datos

¿Por qué son **útiles los tipos abstractos de datos**?

- Los programas que los usan hacen referencia a las operaciones que tienen, no a la representación, y por lo tanto ese programa sigue funcionando si se cambia la representación.
- Simplifican el desarrollo de algoritmos utilizando las operaciones del tipo abstracto de dato, sin importar cómo las mismas son implementadas.
- Dado que una operación puede ser implementada de diferentes formas en un TAD, resulta útil escribir algoritmos que puedan ser usados con cualquiera de sus posibles implementaciones.

Introducción a Tipos Abstractos de Datos

¿Por qué son **útiles los tipos abstractos de datos**?

- Los programas que los usan hacen referencia a las operaciones que tienen, no a la representación, y por lo tanto ese programa sigue funcionando si se cambia la representación.
- Simplifican el desarrollo de algoritmos utilizando las operaciones del tipo abstracto de dato, sin importar cómo las mismas son implementadas.
- Dado que una operación puede ser implementada de diferentes formas en un TAD, resulta útil escribir algoritmos que puedan ser usados con cualquiera de sus posibles implementaciones.
- Algunos TADs utilizados con frecuencia, son implementados en librerías estándares de manera que puedan ser utilizados por cualquier programador.

Introducción a Tipos Abstractos de Datos

¿Por qué son **útiles los tipos abstractos de datos**?

- Los programas que los usan hacen referencia a las operaciones que tienen, no a la representación, y por lo tanto ese programa sigue funcionando si se cambia la representación.
- Simplifican el desarrollo de algoritmos utilizando las operaciones del tipo abstracto de dato, sin importar cómo las mismas son implementadas.
- Dado que una operación puede ser implementada de diferentes formas en un TAD, resulta útil escribir algoritmos que puedan ser usados con cualquiera de sus posibles implementaciones.
- Algunos TADs utilizados con frecuencia, son implementados en librerías estándares de manera que puedan ser utilizados por cualquier programador.
- Las operaciones de los TADs proveen una especie de lenguaje de alto nivel para discutir y especificar otros algoritmos.

Introducción a Tipos Abstractos de Datos

Dentro del **ciclo de vida de un TAD** hay dos fases:

Introducción a Tipos Abstractos de Datos

Dentro del **ciclo de vida de un TAD** hay dos fases:

la programación del TAD: en la cual se elige una representación, y luego se programa cada uno de los métodos sobre esa representación.

Introducción a Tipos Abstractos de Datos

Dentro del **ciclo de vida de un TAD** hay dos fases:

la programación del TAD: en la cual se elige una representación, y luego se programa cada uno de los métodos sobre esa representación.

la construcción de los programas que lo usan: en esta fase no será relevante para el programador que utiliza el TAD cómo está implementado, sino únicamente los métodos que posee.

Introducción a Tipos Abstractos de Datos

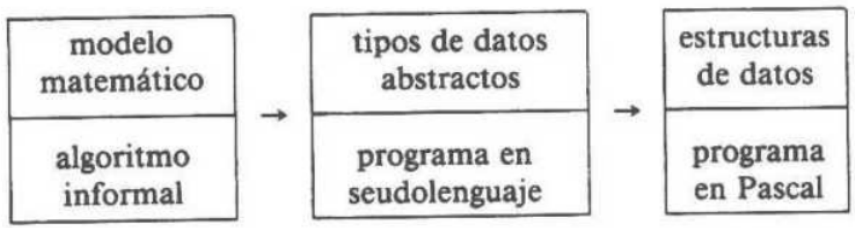


Fig. 1.9. Proceso de solución de problemas.

Figura: Proceso de solución de problemas

Introducción a Tipos Abstractos de Datos

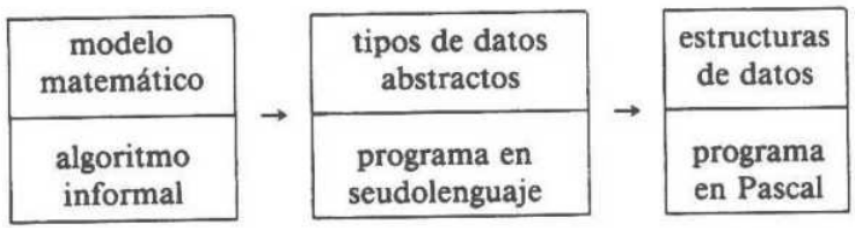


Fig. 1.9. Proceso de solución de problemas.

Figura: Proceso de solución de problemas

Introducción a Tipos Abstractos de Datos

En programación orientada a objetos, una **interfaz** (también llamada **protocolo**) es un medio común para que los objetos no relacionados se comuniquen entre sí.

Introducción a Tipos Abstractos de Datos

En programación orientada a objetos, una **interfaz** (también llamada **protocolo**) es un medio común para que los objetos no relacionados se comuniquen entre sí.

Estas son definiciones de métodos y valores sobre los cuales los objetos están de acuerdo para cooperar.

Introducción a Tipos Abstractos de Datos

En programación orientada a objetos, una **interfaz** (también llamada **protocolo**) es un medio común para que los objetos no relacionados se comuniquen entre sí.

Estas son definiciones de métodos y valores sobre los cuales los objetos están de acuerdo para cooperar.

Utilizando el concepto de interfaz, podemos decir que a **quien utilice el TAD, sólo le interesará la interfaz que éste ofrezca.**

TAD - Lista (List)

El **TAD Lista** representa un conjunto de valores ordenados y numerables, con las siguientes operaciones:

`__str__` para obtener una representación de la lista como cadena de texto.

`__len__` para calcular la longitud de la lista.

`append(x)` para agregar un elemento al final de la lista.

`insert(i, x)` para agregar el elemento x en la posición i.

`remove(x)` para eliminar la primera aparición de x en la lista (si x no se encuentra presente, imprimirá un error y detendrá la ejecución inmediatamente.)

TAD - Lista (List)

El **TAD Lista** representa un conjunto de valores ordenados y numerables, con las siguientes operaciones:

`pop(i)` para eliminar el elemento que está en la posición i y devolver su valor. Si no se especifica el valor de i, pop() elimina y devuelve el elemento que está en el último lugar de la lista (si el indice es inválido, imprimirá un error y detendrá la ejecución inmediatamente.)

`index(x)` devuelve la posición de la primera aparición de x en la lista (si x no se encuentra presente, imprimirá un error y detendrá la ejecución inmediatamente.)

Las listas de Python son una implementación del TAD Lista. Esto significa que el tipo de datos nativo `list` en Python provee una implementación para (al menos) las operaciones mencionadas en la sección anterior

Arreglos y Performance

La representación interna de las listas nativas de Python utiliza una estructura de datos llamada **arreglo**. Un arreglo es la forma más simple de almacenar una secuencia de datos en la memoria, ya que los elementos están contiguos.

Acceder a un elemento cualquiera de un arreglo dada su posición `i` (es decir, lo que hacemos en Python con la operación `lista[i]`) es una operación de **tiempo constante**, por lo cual, lo que tarda la operación no depende de la cantidad de elementos que contiene el arreglo.

Las listas de Python son una implementación del TAD Lista. Esto significa que el tipo de datos nativo `list` en Python provee una implementación para (al menos) las operaciones mencionadas en la sección anterior

Arreglos y Performance

La representación interna de las listas nativas de Python utiliza una estructura de datos llamada **arreglo**. Un arreglo es la forma más simple de almacenar una secuencia de datos en la memoria, ya que los elementos están contiguos.

En cambio, dado que los elementos deben estar siempre contiguos en la memoria, agregar o quitar elementos en una posición dada del arreglo es una operación de **tiempo lineal**: esta operación va a tardar más o menos dependiendo del tamaño de la lista, y la posición donde queramos agregar o quitar elementos.

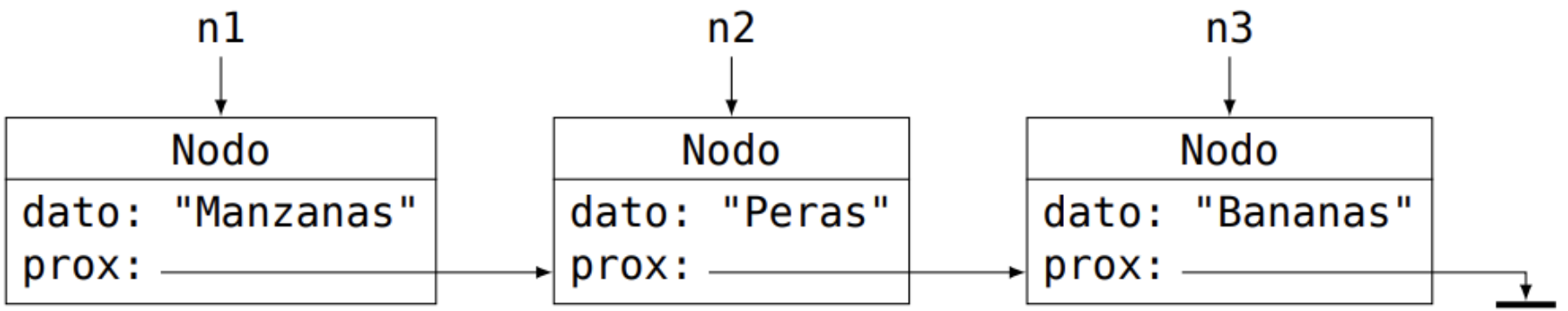
Una lista enlazada está formada por nodos, y cada nodo guarda un elemento y una referencia a otro nodo, como si fueran vagones en un tren.

```
class Nodo:
    def __init__(self, dato: Any =None,
                 prox: "Nodo" | None =None):
        self.dato = dato
        self.prox = prox
    def __str__(self):
        return str(self.dato)
```

TAD - Lista (List)

Implementación: Lista Enlazada

```
>>> n3 = Nodo("Bananas")
>>> n2 = Nodo("Peras", n3)
>>> n1 = Nodo("Manzanas", n2)
>>> str(n1)
'Manzanas'
>>> str(n2)
'Peras'
>>> str(n3)
'Bananas'
```



Hemos creado una lista en forma manual. Si nos interesa recorrerla, podemos hacer lo siguiente:

```
def ver_lista(nodo: Nodo | None) -> None:
    """Recorre todos los nodos a través de sus
    enlaces, mostrando sus contenidos."""
    while nodo is not None:
        print(nodo)
        nodo = nodo.prox
```

Podemos utilizar esta función de la siguiente forma:

```
>>> ver_lista(n1)
Manzanas
Peras
Bananas
```

Si se desea «desenganchar» un nodo del medio de la lista, alcanza con cambiar la referencia `proximo`, por ejemplo:

```
>>> n1.prox = n3
>>> ver_lista(n1)
Manzanas
Bananas
>>> n1.prox = None
>>> ver_lista(n1)
Manzanas
```

Puede suceder que querramos quitar a `n1` y continuar con el resto de la lista como la colección de nodos a tratar.
Una solución muy simple es asociar una referencia al principio de la lista, que llamaremos lista, y que mantendremos independientemente de cuál sea el nodo que está al principio de la lista, por ejemplo:

```
>>> n3 = Nodo("Bananas")
>>> n2 = Nodo("Peras", n3)
>>> n1 = Nodo("Manzanas", n2)
>>> lista = n1
>>> ver_lista(lista)
Manzanas
Peras
Bananas
```

Ahora sí estamos en condiciones de eliminar el primer elemento de la lista sin perder la identidad de la misma:

```
>>> lista = lista.prox
>>> ver_lista(lista)
Peras
Bananas
```

Ventaja importante de las listas enlazadas: eliminar el primer elemento es una operación de tiempo constante.
Desventaja: el acceso a la posición i se realiza en un tiempo proporcional a i.

Clase ListaEnlazada

Basándonos en los nodos implementados anteriormente, pero buscando desligar al programador que desea usar la lista de la responsabilidad de manipular las referencias, definiremos ahora la clase ListaEnlazada, de modo tal que no haya que operar mediante las referencias internas de los nodos, sino que se lo pueda hacer a través de operaciones de lista.

Recordamos a continuación las operaciones que inicialmente deberá cumplir la clase ListaEnlazada:

`__str__` para obtener una representación de la lista como cadena de texto.

`__len__` para calcular la longitud de la lista.

`append(x)` para agregar un elemento al final de la lista.

`insert(i, x)` para agregar el elemento x en la posición i.

`remove(x)` para eliminar la primera aparición de x en la lista (si x no se encuentra presente, imprimirá un error y detendrá la ejecución inmediatamente.)

Recordamos a continuación las operaciones que inicialmente deberá cumplir la clase ListaEnlazada:

`pop(i)` para eliminar el elemento que está en la posición i y devolver su valor. Si no se especifica el valor de i, pop() elimina y devuelve el elemento que está en el último lugar de la lista (si el indice es inválido, imprimirá un error y devolverá inmediatamente.)

`index(x)` devuelve la posición de la primera aparición de x en la lista (si x no se encuentra presente, imprimirá un error y detendrá la ejecución inmediatamente.)

TAD - Lista (List)

Implementación: Lista Enlazada

Valen ahora algunas consideraciones más antes de empezar a implementar la clase:

TAD - Lista (List)

Implementación: Lista Enlazada

Valen ahora algunas consideraciones más antes de empezar a implementar la clase:

- La lista deberá tener como atributo la referencia al primer nodo

TAD - Lista (List)

Implementación: Lista Enlazada

Valen ahora algunas consideraciones más antes de empezar a implementar la clase:

- La lista deberá tener como atributo la referencia al primer nodo
- Una implementación trivial del método `__len__` es recorrer todos los nodos de la lista y contar la cantidad de elementos. Para mejorar la eficiencia alcanza con agregar un atributo numérico que contenga la cantidad de nodos `len`.

TAD - Lista (List)

Implementación: Lista Enlazada

Valen ahora algunas consideraciones más antes de empezar a implementar la clase:

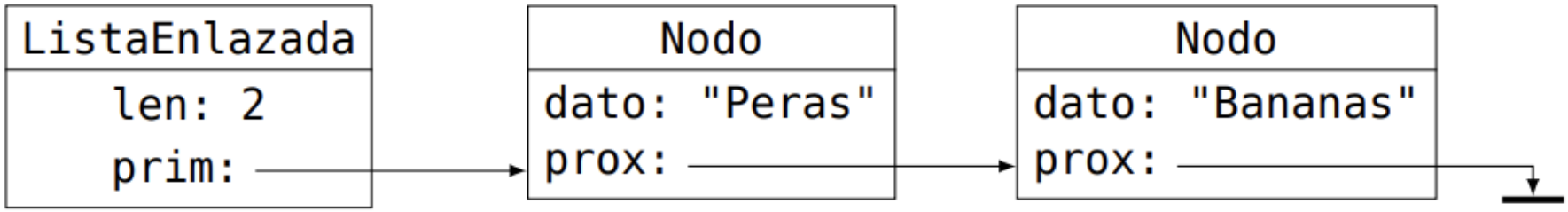
- La lista deberá tener como atributo la referencia al primer nodo
- Una implementación trivial del método `__len__` es recorrer todos los nodos de la lista y contar la cantidad de elementos. Para mejorar la eficiencia alcanza con agregar un atributo numérico que contenga la cantidad de nodos `len`.
- Por otro lado, como vamos a incluir todas las operaciones de listas que sean necesarias para operar con ellas, no es necesario que la clase `Nodo` esté disponible para que otros programadores puedan modificar (y romper) las listas a voluntad usando operaciones de nodos.
  Para hacer que los nodos sean privados, cambiaremos el nombre de la clase a `_Nodo`. Esto es sólo una convención entre programadores, nada impide que utilicemos la clase `_Nodo` desde fuera.

Clase ListaEnlazada

Empezamos escribiendo la clase con su constructor.

```
class ListaEnlazada:
    """Modela una lista enlazada."""
    def __init__(self) -> None:
        """Crea una lista enlazada vacía."""
        # referencia al primer nodo (None si la
        # lista está vacía)
        self.prim = None
        # cantidad de elementos de la lista
        self.len = 0
```

Si la lista contiene dos elementos, su estructura será como la siguiente:



Ejercicio

Escribir los métodos `__str__` y `__len__` para la lista enlazada.

TAD - Lista (List)

Implementación: Lista Enlazada

```
def insert(self, i: int, x: Any) -> None:
    """Inserta el elemento x en la posición i.
    Si la posición es inválida, imprime un error
    y retorna inmediatamente"""
    if i < 0 or i > self.len:
        print("Posición inválida")
        return

    nuevo = _Nodo(x)
    if i == 0:
        # Caso particular: insertar al principio
        nuevo.prox = self.prim
        self.prim = nuevo
```

```
    else:
        # Buscar el nodo anterior a la posición
        # deseada
        n_ant =self.prim
        for pos in range(1, i):
            n_ant = n_ant.prox

        # Intercalar el nuevo nodo
        nuevo.prox = n_ant.prox
        n_ant.prox = nuevo

    self.len += 1
```

TAD - Lista (List)

Implementación: Lista Enlazada

```
def pop(self, i: int | None = None) -> Any:
    """Elim nodo de la posición i, y devuel
    ve el dato contenido. Si i fuera de rango,
    muestra un mensaje de error y termina. Si no
    se recibe la posición, devuelve el último
    elemento."""
    if i is None:
        i = self.len - 1
    if i < 0 or i >= self.len:
        print("Posición inválida")
        return
    if i == 0:
        # Caso particular: saltear la cabecera de
        # la lista
        dato = self.prim.dato
        self.prim = self.prim.prox
```

```
    else:
        # Buscar los nodos en las posiciones (i-1)
        # e (i)
        n_ant = self.prim
        n_act = n_ant.prox
        for pos in range(1, i):
            n_ant = n_act
            n_act = n_ant.prox
        # Guardar el dato y descartar el nodo
        dato = n_act.dato
        n_ant.prox = n_act.prox
    self.len -= 1
    return dato
```

```
def remove(self, x: Any) -> None:
    """Borra la primera aparición del valor x en
    la lista. Si x no está en la lista, imprime
    un mensaje de error y retorna inmediatamente."""
    if self.len == 0:
        print("La lista esta vacía")
        return
    if self.prim.dato == x:
        # Caso particular: saltear la cabecera de
        # la lista
        self.prim = self.prim.prox
```

TAD - Lista (List)

Implementación: Lista Enlazada

```
    else:
        # Buscar el nodo anterior al que contiene
        # a x (n_ant)
        n_ant = self.prim
        n_act = n_ant.prox
        while n_act is not None and n_act.dato != x
            n_ant = n_act
            n_act = n_ant.prox
        if n_act == None:
            print("El valor no está en la lista.")
            return

        # Descartar el nodo
        n_ant.prox = n_act.prox
        self.len -= 1
```

Invariantes de objeto

Los **invariantes** son condiciones que deben ser siempre ciertas. Las **invariantes de objetos**, que son condiciones que deben ser ciertas a lo largo de toda la existencia de un objeto.

La clase `ListaEnlazada` presentada en la sección anterior, cuenta con dos invariantes que siempre debemos mantener. Por un lado, el atributo `len` debe contener siempre la cantidad de nodos de la lista. Por otro lado, el atributo `prim` referencia siempre al primer nodo de la lista.

¿PREGUNTAS?

Referencias

Apunte de Cátedra

Elaborados por el staff docente.

Será subido al campus virtual de la materia.

Apunte de la Facultad de Ingeniería de la UBA, 2da Edición, 2016.

Algoritmos y Programación I: Aprendiendo a programar usando Python como herramienta.

Unidades 15 y 16

A. Downey et al, 2002.

How to Think Like a Computer Scientist. Learning with Python.

Capítulos 12 a 16

Alfred V. Aho, John E. Hopcroft, Jefrey D. Ullman con la colaboración de Guillermo Levine Gutiérrez; versión en español de Américo Vargas y Jorge Lozano

Estructura de datos y algoritmos