UT3 – Resumen lecturas

Listas

Secuencia de cero o más elementos de un tipo determinado.

Sucesión de elementos separados por comas:

a1, a2, …, an donde n >= 0 y ai es del tipo de elemento.

Los elementos pueden estar ordenados en forma lineal de acuerdo a sus posiciones en la lista.

Se trata de un modelo abstracto genérico de amplio uso en la vida cotidiana.

La lista permite agregar un elemento en cualquier lugar:

* Al final, el nuevo elemento será ahora el último de la lista,
* Al principio, el nuevo elemento será ahora el primero de la lista,
* En cualquier posición o de acuerdo a algún criterio (ejemplo: en una posición “x”, en orden alfabético, etc.

Eliminar un elemento de una lista:

* Sea que conozcamos al elemento, o alguna característica del mismo (por ejemplo el número de cédula de identidad),
* Sea que queramos eliminar el elemento de cierta posición “x”.

Acceder a cierto elemento para realizar alguna acción sobre el mismo.

Saber si está vacía. Saber cuantos elementos tiene. Vaciarla.

Una vez implementadas las funcionalidades básicas, a partir de ellas se pueden construir algoritmos más complejos, por ejemplo:

* Eliminar eventuales elementos duplicados de una lista,
* Concatenar una lista con otra,
* Por ejemplo, si las listas implementan conjuntos matemáticos, las operaciones sobre conjuntos (la lista de alumnos inscriptos a Sistemas Digitales “o” a Fundamentos de Gestión, pero no a ambas).
* Etc.

Implementaciones de List de propósito general:

Hay 2 implementaciones: ArrayList y LinkedList. ArrayList ofrece positional Access de tiempo constante y es rápido (sin sincronización).

Si se agregan y quitan elementos con frecuencia, es mejor LinkedList porque estas operaciones requieren tiempo constante, en ArrayList tiempo lineal.

Implementaciones List de propósito específico:

CopyOnWriteArrayList es una implementación de List respaldada por una matriz de copia de escritura.

Si se necesita sincronización un Vector es más rápido que un ArrayList sincronizado con Collections.synchronizedList

Tipos de listas

Interface List

Una lista es una colección ordenada, puede contener elementos duplicados. Las operaciones de la interfaz lista, además de las de Collection son:

* Positional Access: Manipular elementos por su posición numérica en lista (get, set, add, addAll, remove)
* Search: Busca un objeto específico en lista y retorna el índice (indexOf, lastIndexOf)
* Iteration: Extiende la semántica de Iterator (listIterator)
* Range-view: el método de sublista realiza operaciones de rango en lista. La implementación se da en ArrayList (mejor rendimiento) y en LinkedList (en determinadas circunstancias mejor rendimiento).

Remueve al principio y agrega elementos al final

Es la encargada de agrupar una colección de elementos uno a continuación del siguiente.

Tiene la firma de todas las operaciones del TDA Lista.

Similar a la que utilizamos en el curso

* boolean isEmpty();
* boolean add(E e);
* boolean remove(Object o);
* int indexOf(Object o);
* E remove (int index);
* Void clear();

ArrayList

Ofrece positional Access de tiempo constante y es rápido (sin sincronización).

Implementación de Lista con un array (Implementa List).

Array dinámico -> se redimensiona según vaya necesitando más lugares en la lista.

Beneficios ->

* Rápido acceso a los elementos.
* Otras bondades de los arrays.

Problema -> redimensionar

LinkedList

Lista compuesta de nodos donde cada nodo contiene una referencia al próximo nodo, además contiene una unidad de datos llamada carga (contenedor de objetos).

Último nodo tiene como próximo nodo = null, el primer nodo sirve como una referencia a la lista entera.

Implementación de una lista doblemente encadenada (implementa List).

Beneficios ->

* Inserciones y eliminaciones mucho más rápidas que ArrayList.
* Manejo de memoria.

Desventajas -> búsquedas en ArrayList si conozco índices son mucho más rápidas (brinda la posibilidad de acceder directamente al Elemento).

Ejemplo de recorrer una lista con recurrencia:

imprimirInverso(Nodo lista)

Si lista == null retornar vacio

cabeza = lista

cola = lista.prox

imprimirInverso(cola) // llamada recursiva

imprimir(cabeza)

fin

Solo con pasar el primer nodo como parámetro ya funcionaría.

Stack

Java provee un tipo de objeto preincorporado llamado Stack que implementa el TAD Pila. Antes de usar la clase Stack, tenemos que importarla desde java.util. Las operaciones del TAD en la clase Stack de Java tienen los siguientes nombres:

* Apilar: push
* Desapilar: pop
* estaVacia: isEmpty

Entonces la sintaxis para construir una nueva Stack es:

Stack pila = new Stack();

El tipo de retorno de pop es Object! Esto es porque la implementación de la pila no sabe exactamente de qué tipo son los objetos que contiene. Cuando apilamos los objetos de tipo Nodo, son convertidos automáticamente en Objects. Cuando los obtenemos de nuevo desde el stack tenemos que castearlos de nuevo a Nodo.

Métodos:

* public E push(E item);
* public synchronized E pop();
* public synchronized E peek();
* public boolean empty();

TDA: Tipo de datos abstracto

Un tipo de dato abstracto, o TAD, especifica un conjunto de operaciones (o métodos) y la semántica de las operaciones (que es lo que hacen) pero no especifica la implementación de las mismas. Eso es lo que las hace abstractas.

¿Por qué es útil?

* Simplifica la tarea de especificar un algoritmo si podemos denotar las operaciones que necesitamos sin tener que pensar al mismo tiempo cómo esas operaciones se realizan.
* Ya que hay usualmente muchas maneras distintas de implementar un TAD, podría ser útil escribir un algoritmo que pueda ser usado con cualquiera de las posibles implementaciones.
* Existen TADs comúnmente reconocidos, como el TAD Pila en este capítulo, que a menudo son implementados en bibliotecas estándar de manera que sean escritos una sola vez y luego sean reutilizados por muchos programadores.
* Las operaciones en TADs proveen un lenguaje de alto nivel para especificar y hablar de algoritmos.

Cuando hablamos de TADs, a menudo distinguimos el código que usa al TAD, llamado el código cliente, del código que implementa al TAD, llamado el código proveedor, ya que provee un conjunto de servicios estándar.

Es posible cambiar las implementaciones sin cambiar código cliente.

Implementación del TDA Lista con arreglo

Imagen que contiene Gráfico

Descripción generada automáticamente

Implementación con arreglo -->

* Emplea memoria contigua,
* Se reserva memoria que eventualmente no se usa.
* Puede requerir redimensionamiento en tiempo de ejecución.
* Requiere desplazamientos de elementos al insertar o eliminar, excepto si es al final.
* Se puede acceder a una posición determinada en forma directa.

Implementación del TDA Lista con referencias

Gráfico de cajas y bigotes

Descripción generada automáticamente

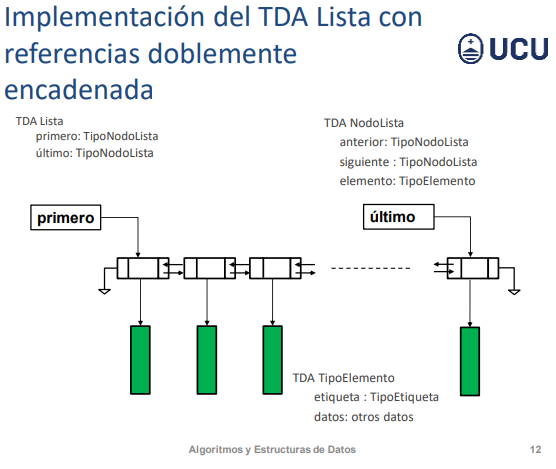
Implementación con referencias -->

* La lista está formada por nodos o celdas; cada nodo contiene un elemento de la lista y una referencia al siguiente nodo,
* El último nodo contiene una referencia nula,
* Evita el empleo de memoria contigua,
* Evita los desplazamientos de elementos al insertar o eliminar,
* Precio adicional: espacio requerido por las referencias,
* Opcionalmente podríamos tener una referencia al último nodo o celda, resulta útil.

Comparación de métodos -->

* La realización con arreglos exige especificar el tamaño máximo de la lista: se puede estar desperdiciando espacio de almacenamiento,
* Ciertas realizaciones son más lentas en una representación que en otra,
* La realización con referencias utiliza para los elementos sólo el espacio real requerido por ellos, pero debe agregarse el espacio necesario para cada referencia.

Implementación del TDA Lista con referencias doblemente encadenada



Implementación del TDA Lista con referencias -->

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Ejemplo:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Seudocódigo

Objetivo -> especificar claramente soluciones algorítmicas, independiente del lenguaje de programación.

Especificación de algoritmos en seudocódigo:

1. Descripción de la solución en lenguaje natural. Se escribe abstracto y alto nivel pareciéndose a la realidad.
2. Especificación del algoritmo en seudocódigo formal detallado. Se comienza por indicar las precondiciones y postcondiciones, luego se escribe el seudocódigo.

* PRECONDICIONES
  + Estado en que debe encontrase el objeto antes de comenzar el algoritmo,
  + Lenguaje natural,
  + Especificación rigurosa,
  + Mapeo con el código,
  + Ayudan a escribir los casos de prueba
* POSTCONDICIONES
  + Estado en que ha de quedar el objeto después de terminar el algoritmo
  + Facilitan la escritura de los casos de prueba.

Las pre/postcondiciones sirven para:

* Completar la comprensión del algoritmo,
* Identificar y entender las condiciones de borde,
* Diseñar pruebas para verificar la corrección de los resultados esperados.

1. Escritura del programa en algún lenguaje de programación.

Ejemplo:

**Si** condición **Entonces**

Bloque de sentencias

**Si no**

Bloque de sentencias

**Fin si**

**Mientras** condición **hacer**

Bloque de sentencias

Actualizar vcb

**Fin mientras**

**Variables**

* Nombres significativos, excepto de las vcb. vcb => variable de control de bucle (se debe inicializar antes de entrar al bucle y actualizarla dentro).
* Notación camello (unAlumno)
* Indicación de tipo de dato (puede omitirse en caso de ser obvio o que no aporten al conocimiento).

Ejemplo:

Inicializar vcb

**Hacer**

Bloque de sentencias

Actualizar vcb

**Mientras (**condición de la vcb)

Operadores: +, -, \*, /, AND, OR, XOR, NOT.

Comparación: <, >, <=, <>, >=, =

Asignación: ß

Ordenar una Lista

Podemos distinguir 3 métodos de ordenación o clasificación (sort) sencillos e intuitivos, llamados “directos”, para ordenar una secuencia de elementos:

* **Intercambio:** recorrer la lista “n” veces intercambiando los elementos contiguos que no respeten el orden,
* **Selección:** en cada paso obtener el elemento de menor clave e insertarlo en el lugar que le corresponde en el área de salida,
* **Inserción:** en cada paso quitar el siguiente elemento del área de entrada y colocarlo en el lugar que le corresponde en el área de salida.

Ordenar una Lista Enlazada

El método de intercambio está especialmente desaconsejado para ordenar una lista enlazada.

Los métodos directos que pueden usarse son: selección, inserción.

Selección directa:

* Crear una nueva lista vacía, sea “ordenada”,
* Hasta que la lista original quede vacía:
  + Obtener el elemento de mayor (menor) clave,
  + Quitarlo de la lista,
  + Insertarlo en la primera (última) posición de la lista ordenada
* Devolver la lista ordenada.

Inserción directa:

* Crear una nueva lista vacía, sea “ordenada”,
* Hasta que la lista original quede vacía:
  + Obtener el primer elemento,
  + Quitarlo de la lista,
  + Insertarlo en la posición que le corresponde de la lista ordenada.
* Devolver la lista ordenada

Pilas

Una pila es una colección, lo cual significa que es una estructura de datos que contiene múltiples elementos.

Tipo especial de lista en que todas las inserciones y eliminaciones se hacen en el mismo extremo, denominado tope.

Listas LIFO (Last In First Out).

Operaciones normales para el TDA Pila:

* Anula
* Tope
* Saca –“pop”
* Mete (unElemento) –“push”
* Vacía

Realización de pila con arreglo

Diagrama

Descripción generada automáticamente

Realización de pila con referencias

Diagrama

Descripción generada automáticamente

El TDA cola y el TDA cola de prioridad tienen el mismo conjunto de operaciones y sus interfaces son las mismas. La diferencia esta en la semántica de las operaciones, la cola usa la política FIFO, mientras que la cola de prioridad usa la política de encolado de prioridades (cliente con prioridad más alta va primero).

Colas

Tipo especial de lista en el que los elementos se insertan en un extremo (el posterior) y se extraen por el otro (en anterior o frente).

Listas FIFO (First In First Out).

Operaciones:

* Anula
* Frente
* PoneEnCola(unElemento)
* QuitaDeCola(unElemento)
* Vacía
* Constructor(crea cola vacía)

Un objeto cola contiene una única variable de instancia que es la lista que la implementa.

Colas enlazadas

Es igual a la lista enlazada con la diferencia que mantiene una referencia del primer y último nodo. En una cola enlazada vacía tanto el primer nodo como el último son null.

Realización de colas con referencias

Igual que en las pilas, cualquier operación de listas es válida para las colas.

Se puede mantener una referencia al principio y otra al final de la cola.

Se puede utilizar como encabezamiento una celda adicional en la cual se ignora el campo elemento.

Implementación de colas con arreglos “circulares”

La representación de listas por medio de arreglos puede también usarse para las colas, pero no es muy eficiente.

Para mejorar, utilizamos un arreglo como si fuera un círculo, es decir, después del último elemento viene nuevamente el primero.

La cola se encuentra en alguna parte del círculo, utilizando posiciones consecutivas.

Diagrama, Dibujo de ingeniería

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

Imagen que contiene Diagrama

Descripción generada automáticamente

Otros comportamientos interesantes de las listas

* Invertir
* Ordenar
* Mezclar dos listas

Formula prolija para evitar “irse” de índice:

prox = (prox + 1) % arreglo.length

**Instructivo de Genéricos**

Los Generics (genéricos) permiten que los tipos (clases e interfaces) sean parámetros cuando se definen clases, interfaces y métodos. Como los parámetros formales, más familiares, usados en la declaración de métodos, los parámetros de tipos proveen una forma de reutilizar el mismo código con diferentes entradas. La diferencia es que las entradas de los parámetros formales son valores, mientras que las entradas de los parámetros de tipos son tipos.

El código que utiliza Generics tiene muchos beneficios sobre el código que no los utiliza:

* Chequeos más fuertes de tipos en tiempo de compilación. El compilador de java aplica el chequeo fuerte de tipos al código genérico y muestra los errores si el código viola el chequeo. Arreglar los errores en tiempo de compilación es más fácil que arreglar errores de tiempo de ejecución, que pueden ser difíciles de encontrar.
* Se elimina la utilización de casts.

El siguiente snippet sin Generics requiere de casting:

Texto, Carta

Descripción generada automáticamente

Cuando se lo reescribe utilizando Generics, el código no requiere de casting:

Texto

Descripción generada automáticamente con confianza media

* Permite a los desarrolladores implementar algoritmos genéricos: Mediante el uso de los Generics, los desarrolladores pueden implementar algoritmos genéricos que funcionan con colecciones de diferentes tipos, pueden ser adaptados, son seguros en cuanto al chequeo de tipos y son fáciles de leer.

**Tipos genéricos**

Un tipo genérico es una clase o interfaz genérica que es parametrizada mediante tipos. La siguiente clase Box será modificada para demostrar el concepto.

**Una simple clase Box**

Empiece por examinar una clase Box no genérica que opera con objetos de cualquier tipo. Sólo necesita de dos métodos: set, que agrega un objeto a la caja, y get, que lo obtiene:

**Interfaz de usuario gráfica, Texto

Descripción generada automáticamente**

Como sus métodos aceptan o devuelven un Object, eres libre de pasarle cualquier cosa que quieras, siempre y cuando no sea uno de los tipos primitivos (en Java todas las clases heredan de Object). No hay forma de verificar, en tiempo de compilación, cómo se utiliza la clase. Una parte del código podría poner un Integer en la caja y esperar obtener Integers de la misma, mientras otra parte del código podría, por error, pasarle un String, resultando en un error en tiempo de ejecución.

**Una versión genérica de la clase Box**

Una clase genérica es definida mediante el siguiente formato:

****

La sección de parámetros de tipo, delimitada por los brackets (<>), se encuentra a continuación del nombre de la clase. Especifica los parámetros de tipo (también llamados variables de tipo) T1, T2 … Tn.

Para actualizar la clase Box y utilizar Generics, creas una declaración de un tipo genérico, cambiando el código “public class Box” a “public class Box<T>”. Esto introduce la variable de tipo, T, que puede ser utilizada en cualquier lugar dentro de la clase.

Con este cambio, la clase Box cambia a:

**Texto, Carta

Descripción generada automáticamente**

**Convención de nombres para parámetros de tipo**

Por convención, los parámetros de tipo se nombran con una única letra mayúscula. Esto contrasta mucho con las convenciones de nombres de variables que ya conocen de Java, y por una buena razón: Sin esta convención, sería difícil distinguir mediante una variable de tipo y el nombre de una clase o interfaz.

Los nombres más comunes usados para parámetros de tipo son:

* E – Elemento (usado extensivamente en Java Collections)
* K – Key (clave)
* N – Number (número)
* T – Tipo
* V – Valor
* S, U, V etc – 2do, 3er y 4to tipo

**Invocando e instanciando un Tipo Genérico**

Para hacer referencia a la clase Box desde dentro de tu código, deberás realizar una invocación de tipo genérico, que reemplaza T con algún valor concreto, como Integer.

Terminología: Parámetro de Tipo y Argumento de Tipo: muchos desarrolladores usan los términos parámetro de tipo y argumento de tipo como si fueran sinónimos, pero estos términos no son iguales. Cuando uno programa, provee argumentos de tipo para crear un tipo parametrizado o parametrizable. Entonces, la T en Foo es un parámetro de tipo y String en Foo es un argumento de tipo. En este instructivo se respeta esta definición cuando se utilizan estos términos.

Como cualquier otra declaración de variable, este código no crea nuevo objeto Box. Simplemente declara que integerBox va a mantener una referencia a “Una Caja de Integer”, que es como se leería Box.

La invocación a un tipo genérico se conoce comúnmente como parameterized type (tipo parametrizado).

Para instanciar esta clase, utiliza la palabra reservada “new”, como siempre, pero pon en medio del nombre de la clase y los paréntesis:

****

**El Diamante**

A partir de Java SE 7 y posterior, se puede reemplazar los argumentos de tipo requeridos para invocar el constructor de una clase genérica con un conjunto vacío de argumentos de tipo (<>), siempre y cuando el compilador pueda determinar, o inferir, los argumentos de tipo a través del contexto. Este par de brackets, <>, se conocen informalmente como “el diamante”. Por ejemplo, se puede crear una instancia de Box con la siguiente sentencia:

****

**Múltiples parámetros de tipo**

Como se mencionó anteriormente, una clase genérica puede tener múltiples parámetros de tipo. Por ejemplo, la clase genérica OrderedPair, que implementa la interfaz genérica Pair:

**Texto

Descripción generada automáticamente**

Las siguientes sentencias crean dos instancias de la clase OrderedPair:

**Interfaz de usuario gráfica, Texto

Descripción generada automáticamente**

El código, new OrderedPair, instancia K como un String y V como un Integer. De esta forma, los parámetros de tipos de OrderedPair son String e Integer, respectivamente.

Gracias al autoboxing (el mecanismo por el cual un tipo primitivo es “convertido” en un objeto que lo envuelve), es válido pasarle un String y un int (en vez de un Integer directamente) a la clase.

Como se mencionó en la sección del Diamante, como el compilador de Java puede inferir los tipos de K y V a partir de la declaración OrderedPair, estas sentencias se pueden acortar utilizando la notación de diamante:

****

**Tipos parametrizados**

También se puede sustituir un parámetro de tipo (ej K o V) con un tipo parametrizado (por ejemplo List). Por ejemplo, usando OrderedPair:

****

**Métodos Genéricos**

Los métodos genéricos son métodos que introducen sus propios parámetros de tipo. Esto es similar a declarar un tipo genérico, pero el alcance del parámetro de tipo estará limitado al método en el que se declare. Se permite tanto para métodos estáticos como no estáticos, así como también para constructores de clases genéricas.

La sintaxis para un método genérico introduce un parámetro de tipo, dentro de los brackets, y aparece antes del tipo de retorno del método. Para métodos estáticos, la sección de parámetro de tipo debe aparecer antes del tipo de retorno.

La clase Util, incluye un método genérico, compare, que compara dos objetos Pair:

**Texto

Descripción generada automáticamente**

La sintaxis completa para invocar este método sería:

**Texto

Descripción generada automáticamente con confianza baja**

El tipo ha sido provisto de forma explícita, como se muestra en negrita. Generalmente esto se puede dejar para que el compilador infiera el tipo que se necesita:

**Texto

Descripción generada automáticamente**

**Parámetros de Tipo “Ligados/Limitados”**

Hay veces en que podés querer restringir los tipos que se pueden usar como argumentos de tipo en un tipo parametrizado. Por ejemplo, un método que opera con números puede querer aceptar sólo instancias de Number o sus subclases. Para esto que son los parámetros de tipo ligados (Bounded Type Parameters en inglés).

Para declarar un parámetro de tipo ligado, debes listar el nombre del parámetro, seguido de la palabra reservada extends (extiende/hereda), seguido de su límite superior, que en este ejemplo es Number. Nótese que en este contexto, extends, es usado en el sentido general para expresar que extends (hereda/extiende como en clases) o implements (implementa interfaz).

**Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente**

Al modificar nuestro método para incluir un parámetro de tipo ligado, la compilación ahora fallará, ya que nuestra invocación de inspect todavía incluye un String:

**Texto

Descripción generada automáticamente**

Además de limitar los tipos que se pueden usar para instanciar un tipo genérico, los parámetros ligados permiten invocar métodos definidos dentro de los límites:

**Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente**

isEven invoca el método intValue definido en Integer a través de n.

**Múltiple**

El ejemplo anterior muestra el uso de un parámetro de tipo con un único límite/ligamiento, pero un parámetro de tipo puede tener múltiples:

****

Una variable con múltiples ligamientos es un subtipo de todos los tipos listados en el ligamiento. Si uno de ellos es una clase, debe ser especificado primero:

**Gráfico

Descripción generada automáticamente con confianza baja**

Si A no es especificado antes que los demás, se obtendrá un error en tiempo de compilación:

****

**Métodos genéricos y parámetros ligados**

Los parámetros de tipo ligados son clave para la implementación de algoritmos genéricos. Considere el siguiente método que cuenta la cantidad de elementos en un array T[] que son más grandes que un elemento “elem”.

**Texto

Descripción generada automáticamente**

La implementación del método es bastante simple, pero no compila porque el operador (>) sólo opera con tipos primitivos como short, int, double, long, float, byte y char. No se puede usar el operador (>) para comparar objetos. Para arreglar el problema, unamos un parámetro de tipo ligado por la interfaz Comparable<T>:

**Texto

Descripción generada automáticamente**

El código resultante será:

**Interfaz de usuario gráfica, Texto

Descripción generada automáticamente con confianza media**