

Clase adicional 9

Temas

- Listas enlazadas
- Árboles
- Problemas de la clase adicional
- Ejercicios de diseño

Listas enlazadas

Previamente en este curso, ya habrá trabajado con dos de las estructuras de datos más básicas: los arrays y los vectores. En esta clase adicional, estudiaremos estructuras de datos más avanzadas: las listas enlazadas, los árboles y los grafos.

Aunque los arrays son una buena solución para almacenar un número de conjunto del mismo tipo de objetos, y aunque los vectores nos permiten controlar dinámicamente el tamaño del espacio de almacenamiento, ambas estructuras de datos tienen un inconveniente importante. Si queremos insertar o eliminar una entrada, es preciso desplazar hacia arriba o hacia abajo todos los elementos situados "debajo" de dicha entrada copiándolos en los elementos adyacentes. A pesar de que es posible salvar esta dificultad (por ejemplo, estableciendo los elementos no utilizados en null), la cantidad de memoria utilizada en un sistema de estas características podría ser ingente y sería necesario modificar el código para manipular todos los espacios vacíos. Un método más eficaz para almacenar los datos que puedan requerir numerosas adiciones y eliminaciones es la lista enlazada.

Definición

Una lista enlazada es una serie de objetos que "saben" dónde se encuentra el siguiente miembro de la lista en la memoria del ordenador. El último miembro de la lista suele indicar que el miembro siguiente es un "null". Para lograrlo, cada objeto de la lista debe disponer de un miembro que pueda almacenar la ubicación en la memoria del siguiente objeto de la lista. Este tipo de listas son listas enlazadas simples, ya que el código puede desplazarse en la lista en una sola dirección. También existen listas enlazadas dobles. Una referencia al objeto de la lista suele indicar el comienzo de la misma.

Operaciones en listas enlazadas

Algunas de las operaciones básicas que pueden realizarse en una lista enlazada son:

- Crear una lista
- Buscar un elemento en la lista
- Insertar un elemento al final de la lista
- Eliminar un elemento de la lista

Implementación en Java

El siguiente código es una implementación sencilla de una lista enlazada: (NOTA: para resumir, el código para buscar un elemento en la lista se incluye en el método de eliminación). Aquí está la clase que declara los datos básicos en cada enlace de la lista:

```
class Check {
    private static int totalChecks=0; // una variable para el seguimiento de todas las comprobaciones

    private int checkNumber;
    private double checkAmount;
    private Check nextCheck; // referencia a la siguiente comprobación de la lista

public Check ( double amount ) {
    checkNumber = ++totalChecks;
    checkAmount = amount;
    nextCheck = null;
}
    public static int getTotalChecks() {
        return totalChecks;
    }
    public int getCheckNumber() {
        return checkNumber;
    }
}
    public Check getNextCheck() {
        return nextCheck;
    }
}
    public void setNextCheck(Check c) {
        nextCheck = c;
    }
}
    public void print() {
        System.out.println("Número de comprobación: #" + checkNumber + "; Check
        Amount : $" +
checkAmount);
    }
}
```

Éste es el código que proporciona los operadores de la lista (añadir, eliminar, está vacío, etc.) Se incluye una excepción como ejemplo en el método deleteChecks. Sería mejor que el código comprobase los null antes de buscar otro enlace, y no que se estancase en la excepción para tratar la excepción NullPointerException que se genera al intentar leer un objeto que no existe.

```
class ListOfChecks {
    // crea una referencia a nuestra lista
    private Check head;
    public ListOfChecks() {
        // establece la referencia a la lista en null, ya que la lista
        // no tiene ningún miembro
        Check head = null;
    }
    public boolean isEmpty() {
        return (head == null); // si la lista está vacía, devolver true
    }
}
    public boolean addToEnd ( double amount ) {
        if ( isEmpty() )
            // si la lista está vacía, comenzar una nueva
```

```

        head = new Check(amount);
    else {
        // "Recorrer la lista" hasta llegar al final (esto es, cuando la siguiente
        // comprobación es null)
        Check temp = head;
        while (temp.getNextCheck() != null)
            temp = temp.getNextCheck();
    ...    temp.setNextCheck(new Check(amount));    // Colocar una marca al final
    }
}    return (true);    // ... y devolver true

public boolean deleteCheck ( int number ) {
    if ( isEmpty() )
        return (false);    // si no hay lista, no se elimina nada
    else if (head.getCheckNumber() == number)
        head = head.getNextCheck();
    // La memoria "eliminada" se ha perdido porque no hay ninguna referencia
    // la recolección de basura devuelve la memoria al sistema.

    else {
        Check temp = head;
        Check previous = null;
        // necesito recordar la comprobación que hay antes de la que
        // quiero eliminar
        while (temp.getCheckNumber() != number) {
            // antes de seguir, recordar de dónde viene
            previous = temp;
            temp = temp.getNextCheck();
        }
        // cuando termina el bucle, temp será la comprobación
        // que debe eliminarse y previous será la comprobación
        // anterior. Así que queremos que la comprobación que ahora
        // sigue a temp, siga a previous
        System.out.println("La comprobación " + temp.getCheckNumber() + " se
        acaba de eliminar.");
        previous.setNextCheck(temp.getNextCheck());

    }
    return (true);
}

}
public void print() {
    int total = 0;
    if (!isEmpty())
    {
        Check temp = head;
        while (temp != null){
            temp.print();
            temp = temp.getNextCheck();
            total++;
        }
    }
    System.out.println("Número total de comprobaciones: " +total);
}

```

```

        System.out.println("Número de comprobaciones escritas: " +
        Check.getTotalChecks());
    }
}

```

El código siguiente prueba las dos clases anteriores.

```

public class ListOfChecksTest {

    public static void main(String args[]) {
        ListOfChecks myList = new ListOfChecks();
        myList.addToEnd(1830.50);
        myList.addToEnd(255.68);
        myList.addToEnd(99.99);
        myList.print();
        myList.deleteCheck(2);
        myList.addToEnd(140.50);
        myList.print();
    }
}

```

Árboles

Los árboles son estructuras de datos cuya apariencia hace exactamente honor a su nombre, con la salvedad de que se dibujan al revés, como las raíces. Después de las listas, son el siguiente método importante para organizar los datos.

Terminología

A continuación incluimos una lista de la terminología que se utiliza al referirse a los árboles:

Nodo	Parte más pequeña de un árbol que almacena datos abstractos.
Nodo raíz	El árbol empieza desde un nodo raíz y se ramifica en varios nodos hijos que, a su vez, se extienden para formar otros nodos hijos. Un nodo raíz puede no tener nodos hijos; se trataría de un nodo único del árbol y recibe el nombre de nodo de hoja.
Nodo hijo	Nodo que se ramifica a partir de un nodo padre. Un nodo hijo siempre tiene un nodo padre.
Nodo padre	Nodo que se ramifica en nodos hijos. Los nodos padres pueden tener uno o varios nodos hijos.
Nodo de hoja	Nodo sin hijos.
Subárbol	Parte del árbol que nace desde un nodo concreto, es decir, desde dicho nodo y todos sus descendientes.
Antecesor	Un antecesor de un nodo A es cualquier nodo entre el nodo raíz (incluido) y el padre de A.
Descendiente	Un descendiente de un nodo A es cualquier nodo hijo de A o un descendiente. Todos los nodos del subárbol que nace en A serán descendientes de A.
Profundidad	Indica la profundidad del árbol; el número máximo de pasos necesarios para alcanzar cualquier nodo de hoja desde el nodo raíz.
Árbol ordenado	Un árbol ordenado es un árbol en el que los hijos de un nodo se ordenan de modo específico.

factor de
ramificación

Número máximo de hijos que puede tener un nodo. Aplicado a árboles, se dice que cada el número de nodos que cada nodo de dicho árbol no puede tener no puede superar el de hijos.

Los datos se almacenan en nodos. El árbol comienza desde un nodo raíz y se ramifica en uno o más nodos que, a su vez, se extienden para crear uno o varios nodos y así sucesivamente hasta terminar en nodos de hoja. Los nodos padres se ramifican en nodos hijos.

Existen distintas secuencias posibles para recorrer un árbol. Las más comunes son:

Preorder:

- raíz
- subárbol izquierdo
- subárbol derecho

Inorder:

- subárbol izquierdo
- raíz
- subárbol derecho

Postorder:

- subárbol izquierdo
- subárbol derecho
- raíz

A continuación se muestra un algoritmo recursivo para realizar el recorrido *Inorder* de un árbol binario.

```
public class BinaryTree
{
    private Object value;
    private BinaryTree left = null;
    private BinaryTree right = null;
    public Vector getInorder()
    {
        Vector vec = new Vector();
        return traverseInorder( this, vec );
    }
    private Vector traverseInorder( BinaryTree b, Vector v )
    {
        if ( b != null )
        {
            traverseInorder( left, v );
            v.addElement( value );
            traverseInorder( right, v );
        }
    }
}
```

```
    }  
    return v;  
  }  
}
```

Problemas de la clase adicional

Problema 1

Todos los elementos de cada lista enlazada siempre tienen:

- Una referencia a otro elemento de la lista o un puntero NULL.
- Ninguna referencia.
- Una referencia al primer elemento de la lista.
- Una referencia al último elemento de la lista.

Problema 2

Si un árbol binario tiene un valor de clave padre inferior que el del hijo izquierdo, pero superior que el del hijo derecho, ¿cuál de los siguientes recorridos imprimirá los datos en orden descendiente?

- Preorder*.
- Inorder*.
- Postorder*.
- Ninguno de los anteriores.

Ejercicios de diseño

1. Complete la clase `BinaryTree` con tres tipos distintos de recorridos (por ejemplo, *preorder*, *inorder* y *postorder*) y un método `main()` para realizar la comprobación. En el método `main()`, construya en primer lugar el árbol binario y, a continuación, recórralo con los tres métodos distintos e imprima el resultado de los recorridos.

2. Los profesores adjuntos del curso 1.00 han decidido redactar una aplicación de Java para realizar el seguimiento de las calificaciones de los estudiantes en el cuestionario 2. En concreto, han decidido utilizar una lista enlazada como estructura de datos para almacenar las calificaciones de todos los estudiantes. Ayúdeles a escribir un método llamado `count()` que devuelva el número de estudiantes que han obtenido una calificación por encima de un umbral determinado.

En primer lugar, ayude a los profesores adjuntos a definir una clase llamada `Student` para almacenar los identificadores y las calificaciones de todos los estudiantes y el acceso que devuelve la calificación del estudiante.

A continuación, ayude a los profesores adjuntos a escribir una clase `StudentListTest` con una función `main` y un método `count()`. En la función `main`, cree una lista que almacene la información de varios estudiantes. Posteriormente, llame al método `count()` para encontrar los estudiantes con calificaciones por encima de la nota umbral. El método `count()` tiene la siguiente firma:

```
public static int count (List s, int threshold)
```

El primer argumento, `s`, es una referencia a una lista enlazada de objetos `Student`. El segundo argumento especifica el umbral de calificaciones. Queremos contar el número de estudiantes que han obtenido una calificación igual o superior al umbral. El método devolverá el resultado del recuento como un `integer`. No debe modificar `List s`.

Sugerencia: se recomienda utilizar la clase `java.util.LinkedList` y la interfaz `java.util.Iterator` para este problema.