Algoritmos y Estructuras de Datos I

Universidad Católica del Uruguay

Listas en Java

Interface List

- Es la encargada de agrupar una colección de elementos uno a continuación del siguiente.
- Tiene la firma de todas las operaciones del TDA Lista visto en clase (y alguna más).

Universidad Católica del Uruguay

Interface List

- 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();

Interface List

- Java -> PODEMOS VER EL CÓDIGO!!!

```
/**
 * Returns the number of elements in this list.  If this list contains
 * more than <tt>Integer.MAX_VALUE</tt> elements, returns
 * <tt>Integer.MAX_VALUE</tt>.
 *
 * @return the number of elements in this list
 */
int size();

/**
 * Returns <tt>true</tt> if this list contains no elements.
 *
 * @return <tt>true</tt> if this list contains no elements
 */
boolean isEmpty();
```

ArrayList

- 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 no es GRATIS!!!

ArrayList

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final long serialVersionUID = 8683452581122892189L;

    /**
     * Default initial capacity.
     */
    private static final int DEFAULT_CAPACITY = 10;

    /**
     * Shared empty array instance used for empty instances.
     */
    private static final Object[] EMPTY_ELEMENTDATA = {};

    /**
```

ArrayList

```
 */
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

```
 * Increases the capacity to ensure that it can hold at least the
 * number of elements specified by the minimum capacity argument.
 *
 * @param minCapacity the desired minimum capacity
 */
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

LinkedList

- 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 accede directamente al Elemento).

LinkedList

Universidad Católica del Uruguay

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
    transient int size = 0;

    /**
     * Pointer to first node.
     * Invariant: (first == null && last == null) ||
     *            (first.prev == null && first.item != null)
     */
    transient Node<E> first;

    /**
     * Pointer to last node.
     * Invariant: (first == null && last == null) ||
     *            (last.next == null && last.item != null)
     */
    transient Node<E> last;

    /**
     * Constructs an empty list.
     */
    public LinkedList() {
    }
```

LinkedList

Universidad Católica del Uruguay

```
 * Links e as last element.
 */
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}

/**
 * Inserts element e before non-null Node succ.
 */
void linkBefore(E e, Node<E> succ) {
    // assert succ != null;
    final Node<E> pred = succ.prev;
    final Node<E> newNode = new Node<>(pred, e, succ);
    succ.prev = newNode;
    if (pred == null)
        first = newNode;
    else
        pred.next = newNode;
    size++;
    modCount++;
```

LinkedList

Universidad Católica del Uruguay

```java
/**
 * Appends the specified element to the end of this list.
 *
 * <p>This method is equivalent to {@link #addLast}.
 *
 * @param e element to be appended to this list
 * @return {@code true} (as specified by {@link Collection#add})
 */
public boolean add(E e) {
    linkLast(e);
    return true;
}
```

Stack

- La clase ***Stack*** es una lista en la que el acceso a sus elementos es de tipo LIFO (Last In - First Out, o último en entrar - primero en salir).
- Es la implementación del TDA Pila de Java
- Hereda de *Vector* (otra implementación de ***List***, muy similar a *ArrayList* pero Synchronized).
- *Stack* hereda de *Vector* (es una ***List***).

Stack

- Métodos
  - public E push(E item)
  - public synchronized E pop()
  - public synchronized E peek()
  - public boolean empty()

Stack

```
class Stack<E> extends Vector<E> {
    /**
     * Creates an empty Stack.
     */
    public Stack() {
    }

    /**
     * Pushes an item onto the top of this stack. This has exactly
     * the same effect as:
     * <blockquote><pre>
     * addElement(item)</pre></blockquote>
     *
     * @param    item    the item to be pushed onto this stack.
     * @return   the <code>item</code> argument.
     * @see      java.util.Vector#addElement
     */
    public E push(E item) {
        addElement(item);

        return item;
    }
```

Stack

```
     * @return  The object at the top of this stack (the last item
     *          of the <tt>Vector</tt> object).
     * @throws  EmptyStackException  if this stack is empty.
     */
    public synchronized E pop() {
        E       obj;
        int     len = size();

        obj = peek();
        removeElementAt(len - 1);

        return obj;
    }

    /**
     * Looks at the object at the top of this stack without removin
     * from the stack.
     *
     * @return  the object at the top of this stack (the last item
     *          of the <tt>Vector</tt> object).
     * @throws  EmptyStackException  if this stack is empty.
     */
    public synchronized E peek() {
        int     len = size();

        if (len == 0)
            throw new EmptyStackException();
        return elementAt(len - 1);
    }
```

Resumiendo

Universidad Católica del Uruguay

- Java utiliza varios de los algoritmos que utilizamos durante la primera mitad del curso.

- Estamos en condiciones entender que es lo que está pasando ahí adentro (ya no es más una caja negra).

Universidad Católica del Uruguay

¿Preguntas?