**Лабораторная работа №7**

Выполнил

студент І курса

группа ИВ-32

Гутов В.В.

Номер ЗК: 3210

Номер ЗК: 3210

С2=0 – реалізувати інтерфейс List

С3=0 - Масив із початковою кількістю елементів 15 та збільшенням кількості елементів на 30%

Код программы

package OOP.lab7;

import java.util.\*;

/\*\*

\*

\* @version 1 29 April 2014

\* @author Gutov Vladislav

\*

\* Class <code>MyCollection</code> implements interface List with start size 15 and adding 30% of curent capacity if needed

\*/

public class MyCollection<Toy> implements List<Toy> {

private Object[] data;

private int size;

/\*\*

\* Empty constructor

\*/

public MyCollection(){

data = new Object[15];

size = 0;

}

/\*\*

\* Constructor with one start element

\* @param obj - element to be added to this list

\*/

public MyCollection(Toy obj){

data = new Object[15];

size = 0;

data[size++] = obj;

}

/\*\*

\* Constructor with another collection

\* @param c - collection containing elements to be added to this list

\*/

public MyCollection(Collection<? extends Toy> c){

data = new Object[15];

size = 0;

this.addAll(c);

}

/\*\*

\* Returns the number of elements in this list.

\*

\* @return the number of elements in this list

\*/

@Override

public int size() {

return size;

}

/\*\*

\* Returns <tt>true</tt> if this list contains no elements.

\*

\* @return <tt>true</tt> if this list contains no elements

\*/

@Override

public boolean isEmpty() {

return size == 0;

}

/\*\*

\* Returns <tt>true</tt> if this list contains the specified element.

\*

\* @param o element whose presence in this list is to be tested

\* @return <tt>true</tt> if this list contains the specified element

\*/

@Override

public boolean contains(Object o) {

return indexOf(o) >= 0;

}

/\*\*

\* Returns an array containing all of the elements in this list in proper

\* sequence (from first to last element).

\* <p/>

\* <p>The returned array will be "safe" in that no references to it are

\* maintained by this list.

\*

\* @return an array containing all of the elements in this list in proper

\* sequence

\*/

@Override

public Object[] toArray() {

return Arrays.copyOf(data, size);

}

/\*\*

\* Returns an array containing all of the elements in this list in

\* proper sequence (from first to last element);

\* <p/>

\* <p>If the list fits in the specified array with room to spare (i.e.,

\* the array has more elements than the list), the element in the array

\* immediately following the end of the list is set to <tt>null</tt>.

\*

\* @param a the array into which the elements of this list are to

\* be stored, if it is big enough; otherwise, a new array of the

\* same runtime type is allocated for this purpose.

\* @return an array containing the elements of this list

\*/

@Override

public Object[] toArray(Object[] a) {

if (a.length < size)

return Arrays.copyOf(data, size, a.getClass());

System.arraycopy(data, 0, a, 0, size);

if (a.length > size)

a[size] = null;

return a;

}

/\*\*

\* Appends the specified element to the end of this list.

\* <p/>

\*

\* @param t element to be appended to this list

\* @return <tt>true</tt>

\*/

@Override

public boolean add(Toy t) {

if (size >= data.length)

addSpace();

data[size++] = t;

return true;

}

/\*\*

\* private method, witch used to ensure capacity to contain elements in list

\*/

private void addSpace(){

Object[] buffer = Arrays.copyOf(data, size, data.getClass());

int length = (int)(1.3 \* data.length);

data = new Object[length];

System.arraycopy(buffer, 0, data, 0, size);

}

/\*\*

\* Removes the first occurrence of the specified element from this list,

\* if it is present.

\*

\* @param o element to be removed from this list, if present

\* @return <tt>true</tt> if this list contained the specified element

\*/

@Override

public boolean remove(Object o) {

int position = indexOf(o);

if (position < 0)

return false;

for(int i = position; i < size; i++){

data[i] = data[i+1];

}

size--;

return true;

}

/\*\*

\* Returns <tt>true</tt> if this list contains all of the elements of the

\* specified collection.

\*

\* @param c collection to be checked for containment in this list

\* @return <tt>true</tt> if this list contains all of the elements of the

\* specified collection

\*/

@Override

public boolean containsAll(Collection<?> c) {

Object[] buffer = c.toArray();

for (int i = 0; i < c.size(); i++){

if(!this.contains(buffer[i]))

return false;

}

return true;

}

/\*\*

\* Appends all of the elements in the specified collection to the end of

\* this list, in the order that they are returned by the specified

\* collection's iterator.

\*

\* @param c collection containing elements to be added to this list

\* @return <tt>true</tt> if this list changed as a result of the call

\*/

@Override

public boolean addAll(Collection<? extends Toy> c) {

Iterator<Toy> iterator = (Iterator<Toy>) c.iterator();

while(iterator.hasNext()){

this.add(iterator.next());

}

return true;

}

/\*\*

\* Inserts all of the elements in the specified collection into this

\* list at the specified position. Shifts the

\* element currently at that position. The new elements

\* will appear in this list in the order that they are returned by the

\* specified collection's iterator.

\*

\* @param index index at which to insert the first element from the

\* specified collection

\* @param c collection containing elements to be added to this list

\* @return <tt>true</tt> if this list changed as a result of the call

\*/

@Override

public boolean addAll(int index, Collection<? extends Toy> c) {

while(size + c.size() > data.length)

addSpace();

if(index < 0)

index = 0;

for(int i = size-1; i >= index; i--){

data[i + c.size()] = data[i];

}

Iterator<Toy> iterator = (Iterator<Toy>) c.iterator();

while(iterator.hasNext()){

data[index++] = iterator.next();

size++;

}

return true;

}

/\*\*

\* Removes from this list all of its elements that are contained in the

\* specified collection.

\*

\* @param c collection containing elements to be removed from this list

\* @return <tt>true</tt> if this list changed as a result of the call

\*/

@Override

public boolean removeAll(Collection<?> c) {

boolean result = false;

Iterator<Toy> iterator = (Iterator<Toy>) c.iterator();

while(iterator.hasNext()){

if(this.remove(iterator.next()))

result = true;

}

return result;

}

/\*\*

\* Retains only the elements in this list that are contained in the

\* specified collection. In other words, removes

\* from this list all of its elements that are not contained in the

\* specified collection.

\*

\* @param c collection containing elements to be retained in this list

\* @return <tt>true</tt> if this list changed as a result of the call

\*/

@Override

public boolean retainAll(Collection<?> c) {

boolean result = false;

MyCollection temp = new MyCollection(this);

Iterator<Toy> iterator = temp.iterator();

while(iterator.hasNext()){

Toy current = iterator.next();

if(!c.contains(current)){

this.remove(current);

result = true;

}

}

return result;

}

/\*\*

\* Removes all of the elements from this list.

\* The list will be empty after this call returns.

\*/

@Override

public void clear() {

data = new Object[15];

size = 0;

}

/\*\*

\* Returns the element at the specified position in this list.

\*

\* @param index index of the element to return

\* @return the element at the specified position in this list

\* @throws IndexOutOfBoundsException if the index is out of range

\*/

@Override

public Toy get(int index) {

if(index >= size)

throw new IndexOutOfBoundsException();

return (Toy) data[index];

}

/\*\*

\* Replaces the element at the specified position in this list with the

\* specified element.

\*

\* @param index index of the element to replace

\* @param element element to be stored at the specified position

\* @return the element previously at the specified position

\* @throws IndexOutOfBoundsException if the index is out of range

\*/

@Override

public Toy set(int index, Toy element) {

if(index >= size)

throw new IndexOutOfBoundsException();

Toy result = (Toy) data[index];

data[index] = element;

return result;

}

/\*\*

\* Inserts the specified element at the specified position in this list.

\* Shifts the element currently at that position

\*

\* @param index index at which the specified element is to be inserted

\* @param element element to be inserted

\*/

@Override

public void add(int index, Toy element) {

if(size >= data.length)

addSpace();

for(int i = size-1; i >= index; i++){

data[i] = data[i-1];

}

data[index] = element;

}

/\*\*

\* Removes the element at the specified position in this list.

\* Shifts any subsequent elements to the left.

\* Returns the element that was removed from the list.

\*

\* @param index the index of the element to be removed

\* @return the element previously at the specified position

\*/

@Override

public Toy remove(int index) {

Toy result = (Toy) data[index];

for(int i = index; i < size; i++){

data[i] = data[i+1];

}

size--;

return result;

}

/\*\*

\* Returns the index of the first occurrence of the specified element

\* in this list, or -1 if this list does not contain the element.

\*

\* @param o element to search for

\* @return the index of the first occurrence of the specified element in

\* this list, or -1 if this list does not contain the element

\*/

@Override

public int indexOf(Object o) {

if (o == null) {

for (int i = 0; i < size; i++)

if (data[i]==null)

return i;

} else {

for (int i = 0; i < size; i++)

if (o.equals(data[i]))

return i;

}

return -1;

}

/\*\*

\* Returns the index of the last occurrence of the specified element

\* in this list, or -1 if this list does not contain the element.

\*

\* @param o element to search for

\* @return the index of the last occurrence of the specified element in

\* this list, or -1 if this list does not contain the element

\*/

@Override

public int lastIndexOf(Object o) {

if (o == null) {

for (int i = size-1; i >= 0; i--)

if (data[i]==null)

return i;

} else {

for (int i = size-1; i >= 0; i--)

if (o.equals(data[i]))

return i;

}

return -1;

}

/\*\*

\* Returns an iterator over the elements in this list in proper sequence.

\*

\* @return an iterator over the elements in this list in proper sequence

\*/

@Override

public Iterator<Toy> iterator() {

return new Iterator<Toy>() {

private int currentIndex = 0;

@Override

public boolean hasNext() {

return currentIndex < size;

}

@Override

public Toy next() {

if (!hasNext())

throw new NoSuchElementException();

return (Toy) data[currentIndex++];

}

@Override

public void remove() {

throw new UnsupportedOperationException();

}

};

}

/\*\*

\* Returns a list iterator over the elements in this list (in proper

\* sequence).

\*

\* @return a list iterator over the elements in this list (in proper

\* sequence)

\*/

@Override

public ListIterator<Toy> listIterator() {

return new LstIterator();

}

/\*\*

\* Returns a list iterator over the elements in this list (in proper

\* sequence), starting at the specified position in the list.

\* The specified index indicates the first element that would be

\* returned by an initial call to {@link java.util.ListIterator#next next}.

\* An initial call to {@link java.util.ListIterator#previous previous} would

\* return the element with the specified index minus one.

\*

\* @param index index of the first element to be returned from the

\* list iterator (by a call to {@link java.util.ListIterator#next next})

\* @return a list iterator over the elements in this list (in proper

\* sequence), starting at the specified position in the list

\*/

@Override

public ListIterator<Toy> listIterator(int index) {

return new LstIterator(index);

}

private class LstIterator implements ListIterator<Toy>{

private int currentIndex = 0;

public LstIterator(){

currentIndex = 0;

}

public LstIterator(int index){

currentIndex = index;

}

@Override

public boolean hasNext() {

return currentIndex < size;

}

@Override

public Toy next() {

if (!hasNext())

throw new NoSuchElementException();

return (Toy) data[currentIndex++];

}

@Override

public boolean hasPrevious() {

return currentIndex > 0;

}

@Override

public Toy previous() {

if (!hasPrevious())

throw new NoSuchElementException();

return (Toy) data[--currentIndex];

}

@Override

public int nextIndex() {

return currentIndex+1;

}

@Override

public int previousIndex() {

return currentIndex-1;

}

@Override

public void remove() {

throw new UnsupportedOperationException();

}

@Override

public void set(Toy toy) {

MyCollection.this.set(currentIndex, toy);

}

@Override

public void add(Toy toy) {

MyCollection.this.add(currentIndex, toy);

}

}

/\*\*

\* Returns a view of the portion of this list between the specified

\* <tt>fromIndex</tt>, inclusive, and <tt>toIndex</tt>, exclusive. (If

\* <tt>fromIndex</tt> and <tt>toIndex</tt> are equal, the returned list is

\* empty.) The returned list is backed by this list, so non-structural

\* changes in the returned list are reflected in this list, and vice-versa.

\* The returned list supports all of the optional list operations supported

\* by this list.<p>

\* <p/>

\*

\* @param fromIndex low endpoint (inclusive) of the subList

\* @param toIndex high endpoint (exclusive) of the subList

\* @return a view of the specified range within this list

\*/

@Override

public List<Toy> subList(int fromIndex, int toIndex) {

List<Toy> result = new MyCollection();

for(int i = fromIndex; i < toIndex; i++){

result.add(this.get(i));

}

return result;

}

}

Выводы

В данной лабораторной работе я реализовал интерфейс List. Все методы работают согласно документации. Начальный размер списка по необходимости увеличивается на 30% от текущего размера. Код был детально задокументирован.