Ho Chi Minh City International University

HCMIU CVIP

Nguyen Qui Vinh Quang, Nguyen Tien Cuong

2021-04-06

List (1)

template.cpp

14 lines

```
#include <bits/stdc++.h>
using namespace std;

#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define all(x) begin(x), end(x)
#define sz(x) (int)(x).size()
typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;

signed main() {
  ios::sync_with_stdio(0); cin.tie(0);
  cin.exceptions(cin.failbit);
}
```

ArrayList.java

246 lines

```
package list;
import java.util.Arrays;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;
public class MyArrayList<E> implements java.util.List<E> {
    private static enum MoveType {
        NEXT, PREV
    }
    private static int MAX_CAPACITY = Integer.MAX_VALUE - 8;
    private E[] elements;
    private int size;
    //Constructor:
    public MyArrayList(int capacity) throws
         IllegalArgumentException {
        if ((capacity < 0) || (capacity > MAX_CAPACITY)) {
            String message = String.format("Invalid capacity
                 (=%d)", capacity);
            throw new IllegalArgumentException(message);
        }

        this.elements = (E[]) new Object[capacity];
        this.size = 0;
    }
    public MyArrayList() throws IllegalArgumentException {
  this(10);}
    //Utitilies
    private void checkValidIndex(int index, int min, int max) {
        if ((index < min) || (index > max)) {
            String message = String.format("Invalid index (=%d)
                 ", index);
            throw new IndexOutOfBoundsException(message);
        }
    }
    @Override
    public int size() {return this.size;}
    @Override
    public boolean isEmpty() {return this.size == 0;}
    @Override
    public boolean contains(Object o) {
        boolean found = false;
        for (int idx = 0; idx < this.size; idx++) {
            if (this.elements[idx].equals(o)) {
                found = true; break;
            }
        }
        return found;
    }
    @Override
    public Iterator<E> iterator() {return new MyIterator();}
    @Override
    public Object[] toArray() {return Arrays.copyOf(elements,
         size);}
    @Override
    public <T> T[] toArray(T[] a) {
        /*IMPLEMENTATION: NOT REQUIRED*/
        throw new UnsupportedOperationException("Not supported
             yet."); .
    }
    @Override
    public boolean add(E e) {
        if (e == null) throw new NullPointerException("Can't
             add null pointer");
        checkCapacity(this.size + 1);
        this.elements[this.size++] = e;
        return true;
    }
    private void checkCapacity(int minCapacity) {
        if ((minCapacity < 0) || (minCapacity > MAX_CAPACITY))
            throw new OutOfMemoryError("Not enough memory to
                 store the array");
        //newCapacity maybe a negative becuase of the overflow
        if (minCapacity >= this.elements.length) {
            //grow: oldCapacity >> 1 (= oldCapacity/2)
            int oldCapacity = this.elements.length;
            int newCapacity = oldCapacity + (oldCapacity / 2);
            if (newCapacity < 0) newCapacity = MAX_CAPACITY;
            this.elements = Arrays.copyOf(this.elements,
                 newCapacity);}
    }
    @Override
    public boolean remove(Object o) {
        int index = indexOf(o);
        if (index < 0) return false;
        remove(index);
        return true;
    }
    public boolean containsAll(Collection<?> c) {
        /*IMPLEMENTATION: NOT REQUIRED*/
        throw new UnsupportedOperationException("Not supported
             yet.");
    }
    @Override
    public boolean addAll(Collection<? extends E> c) {
        /*IMPLEMENTATION: NOT REQUIRED*/
        throw new UnsupportedOperationException("Not supported
             yet.");
    }
    @Override
    public boolean addAll(int index, Collection<? extends E> c)
         {
        /*IMPLEMENTATION: NOT REQUIRED*/
        throw new UnsupportedOperationException("Not supported
             yet.");
    }
    @Override
    public boolean removeAll(Collection<?> c) {
        /*IMPLEMENTATION: NOT REQUIRED*/
        throw new UnsupportedOperationException("Not supported
             yet.");
    }
    @Override
    public boolean retainAll(Collection<?> c) {
        /*IMPLEMENTATION: NOT REQUIRED*/
        throw new UnsupportedOperationException("Not supported
             yet.");
    }
    @Override
    public void clear() {
        while (isEmpty() == false) remove(0);
    }
    @Override
    public E get(int index) {
        checkValidIndex(index, 0, size - 1);
        return this.elements[index];
    }
    @Override
    public E set(int index, E element) {
        checkValidIndex(index, 0, size - 1);
        E oldElement = this.elements[index];
        this.elements[index] = element;
        return oldElement;
    }
    @Override
    public void add(int index, E element) {
        checkValidIndex(index, 0, size);
        if (element == null)
            throw new NullPointerException("Can not add null
                 pointer");
        checkCapacity(this.size + 1);
        int copyCount = (this.size - 1) - index + 1;
        System.arraycopy(this.elements, index, this.elements,
             index + 1, copyCount);
        this.elements[index] = element; this.size++;
    }
    @Override
    public E remove(int index) {
        checkValidIndex(index, 0, size - 1);
        E oldElement = this.elements[index];
        int copyCount = (this.size - 1) - (index + 1) + 1;
        System.arraycopy(this.elements, index + 1, this.
             elements, index,
                 copyCount);
        this.size--;
        return oldElement;
    }
    @Override
    public int indexOf(Object o) {
        int foundIdx = -1;
        for (int idx = 0; idx < this.size; idx++) {
            if (this.elements[idx].equals(o)) { //== not
                foundIdx = idx;
                break;
            }}
        return foundIdx;
    }
    @Override
    public int lastIndexOf(Object o) {
        int foundIdx = -1;
        for (int idx = this.size - 1; idx >= 0; idx--) {
            if (this.elements[idx].equals(o)) {
                foundIdx = idx;
                break;
            }}
        return foundIdx;
    }
    @Override
    public ListIterator<E> listIterator() {return new
         MyListIterator();}
    @Override
    public ListIterator<E> listIterator(int index) {
  return new MyListIterator(index);}
    @Override
```

```
    public List<E> subList(int fromIndex, int toIndex) {
        /*IMPLEMENTATION: NOT REQUIRED*/
        throw new UnsupportedOperationException("Not supported
             yet.");}
    @Override
    public String toString() {
        String desc = "[";
        Iterator<E> it = this.iterator();
        while (it.hasNext()) {
            E e = it.next();
            desc += String.format("%s,", e);
        }
        if (desc.endsWith(","))
            desc = desc.substring(0, desc.length() - 1);
        return desc + "]";
    }
    //Definition of Inner Class
    public class MyIterator implements Iterator<E> {
        int cursor = 0;
        MoveType moveType = MoveType.NEXT;
        boolean afterMove = false;
        @Override
        public boolean hasNext() {
            return this.cursor != MyArrayList.this.size; }
        @Override
        //Move cursor to next + return preivous element
        public E next() {
            cursor += 1;
            moveType = MoveType.NEXT;
            afterMove = true;
            return MyArrayList.this.elements[cursor - 1];
        }
        @Override
        public void remove() {
            if (!afterMove) return;
            MyArrayList.this.remove(cursor - 1);
            cursor -= 1;
            afterMove = false;
        }
    }//End of MyIterator
    public class MyListIterator extends MyIterator implements
         ListIterator<E> {
        public MyListIterator(int index) {cursor = index;}
        public MyListIterator() {}
        @Override
        public boolean hasPrevious() {return this.cursor != 0;}
        @Override
        public void remove() {
            if (!afterMove) return;
            if (moveType == MoveType.NEXT) super.remove();
             else {
                MyArrayList.this.remove(cursor);
                afterMove = false;
            }}
        @Override
        public E previous() {
            cursor -= 1;
            moveType = MoveType.PREV;
            afterMove = true;
            return MyArrayList.this.elements[cursor];
        }
        @Override
        public int nextIndex() { return cursor;}
        @Override
        public int previousIndex() { return cursor - 1;}
        @Override
        public void set(E e) {
            if (!afterMove) return;
            if (moveType == MoveType.NEXT) MyArrayList.this.set
                 (cursor - 1, e);
            else MyArrayList.this.set(cursor, e);
        }
        @Override
        public void add(E e) {
            if (!afterMove) return;
            if (moveType == MoveType.NEXT) {
                MyArrayList.this.add(cursor - 1, e);
            } else MyArrayList.this.add(cursor, e);
            cursor += 1;
            afterMove = false;
        }
    }
}
```