CSE214 Data Structures

List Abstractions

YoungMin Kwon

List ADT

- List
  - Linearly ordered sequence of elements
- Location
  - The location of an element can be easily described by its index
  - Indexes may change as elements are added and removed from the list
  - The range of a valid index may also change
    - IndexOutOfBoundsException for an invalid index

java.util.List Interface Includes

size( ): Returns the number of elements in the list.

isEmpty( ): Returns a boolean indicating whether the list is empty.

get(i): Returns the element of the list having index i; an error condition occurs if i is not in range $[0, \text{size}( ) - 1]$.

set(i, e): Replaces the element at index i with e, and returns the old element that was replaced; an error condition occurs if i is not in range $[0, \text{size}( ) - 1]$.

add(i, e): Inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list; an error condition occurs if i is not in range $[0, \text{size}( )]$.

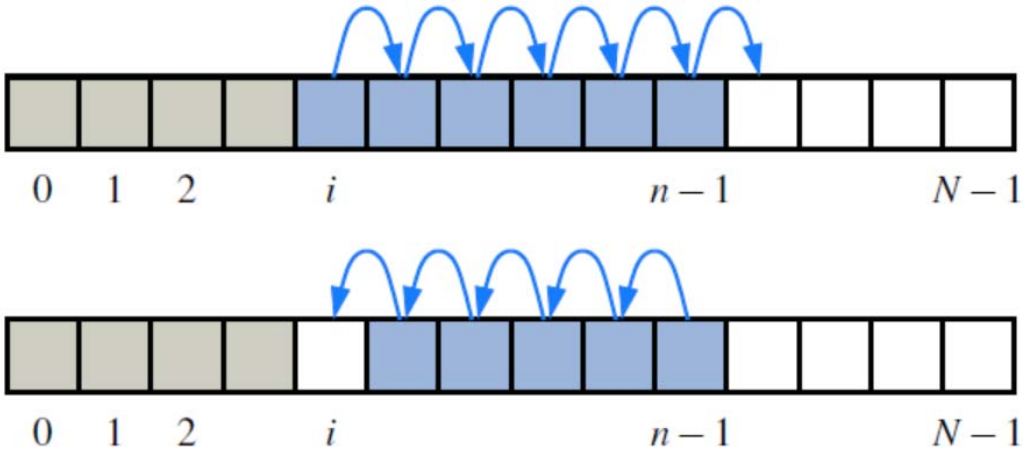
remove(i): Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error condition occurs if i is not in range $[0, \text{size}( ) - 1]$.

Simplified List Interface

```
public interface List<E> {
    //number of elements in the list
    public int size();

    //whether the list is empty
    public boolean isEmpty();

    //get e at index i
    public E get(int i)          throws IndexOutOfBoundsException;

    //set e at index i
    public E set(int i, E e)     throws IndexOutOfBoundsException;

    //add e at index i
    public void add(int i, E e) throws IndexOutOfBoundsException;

    //add e at the last index
    public void add(E e)         throws IndexOutOfBoundsException;

    //remove node at i and return its value
    public E remove(int i)       throws IndexOutOfBoundsException;
}
```

Array-based Lists

- Using an array to implement the List ADT
  - A[i] stores the element at index i
  - Fixed size array or Dynamic array (vector in C++)
- May shift elements to add or to remove



```java
@SuppressWarnings("unchecked")
public class ArrayList<E> implements List<E> {
    protected static final int CAPACITY = 16;
    protected E[] data;
    protected int size;

    public ArrayList()              { this(CAPACITY); }
    public ArrayList(int capacity) { data = (E[])new Object[capacity]; }

    //Interface List
    public int size()               { return size; }
    public boolean isEmpty()        { return size == 0; }

    public E get(int i) throws IndexOutOfBoundsException  {
        checkIndex(i, size);
        return data[i];
    }

    public E set(int i, E e) throws IndexOutOfBoundsException  {
        checkIndex(i, size);
        E old = data[i];    //to return the old element
        data[i] = e;
        return old;
    }
```

+1: we may append at the last index

```
public void add(int i, E e) throws IndexOutOfBoundsException  {
    checkIndex(i, size + 1);

    if(size == data.length)
        throw new IllegalStateException("Array is full");

    for(int k = size-1; k >= i; k--) //shift to the right
        data[k+1] = data[k];
    data[i] = e;

    size++;
}

public void add(E e) throws IndexOutOfBoundsException  {
    add(size, e);
}
```

```
public E remove(int i) throws IndexOutOfBoundsException  {
    checkIndex(i, size);

    E old = data[i];

    for(int k = i; k < size-1; k++) //shift to the left
        data[k] = data[k+1];

    size--;

    return old;
}

protected void checkIndex(int i, int n) throws
                                        IndexOutOfBoundsException {
    if(i < 0 || i >= n)
        throw new IndexOutOfBoundsException("Illegal index: " + i);
}
```

```
    //testing...
    protected static void onFalseThrow(boolean b) {
        if(!b)
            throw new RuntimeException("Error: unexpected");
    }

    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(list.size(), 2);
        list.add(list.size(), 3);
        list.add(list.size(), 4);
        list.add(0, 1);
        onFalseThrow(list.remove(list.size()-1) == 4);
        onFalseThrow(list.remove(list.size()-1) == 3);
        onFalseThrow(list.remove(0) == 1);
        onFalseThrow(list.remove(list.size()-1) == 2);
        System.out.println("Success!");
    }
}
```

Performance of ArrayList

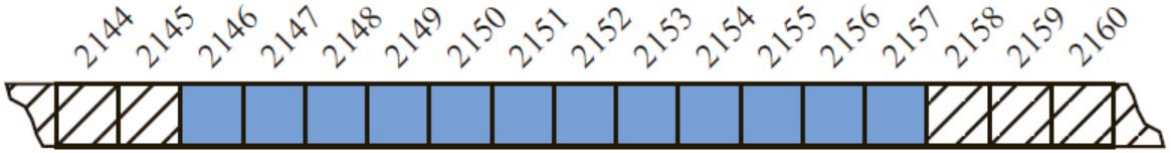
- add() and remove() operations involve shifting
  - On average n/2 elements will be shifted

Method	Running Time
size( )	$O(1)$
isEmpty( )	$O(1)$
get(i)	$O(1)$
set(i, e)	$O(1)$
add(i, e)	$O(n)$
remove(i)	$O(n)$

Dynamic Arrays

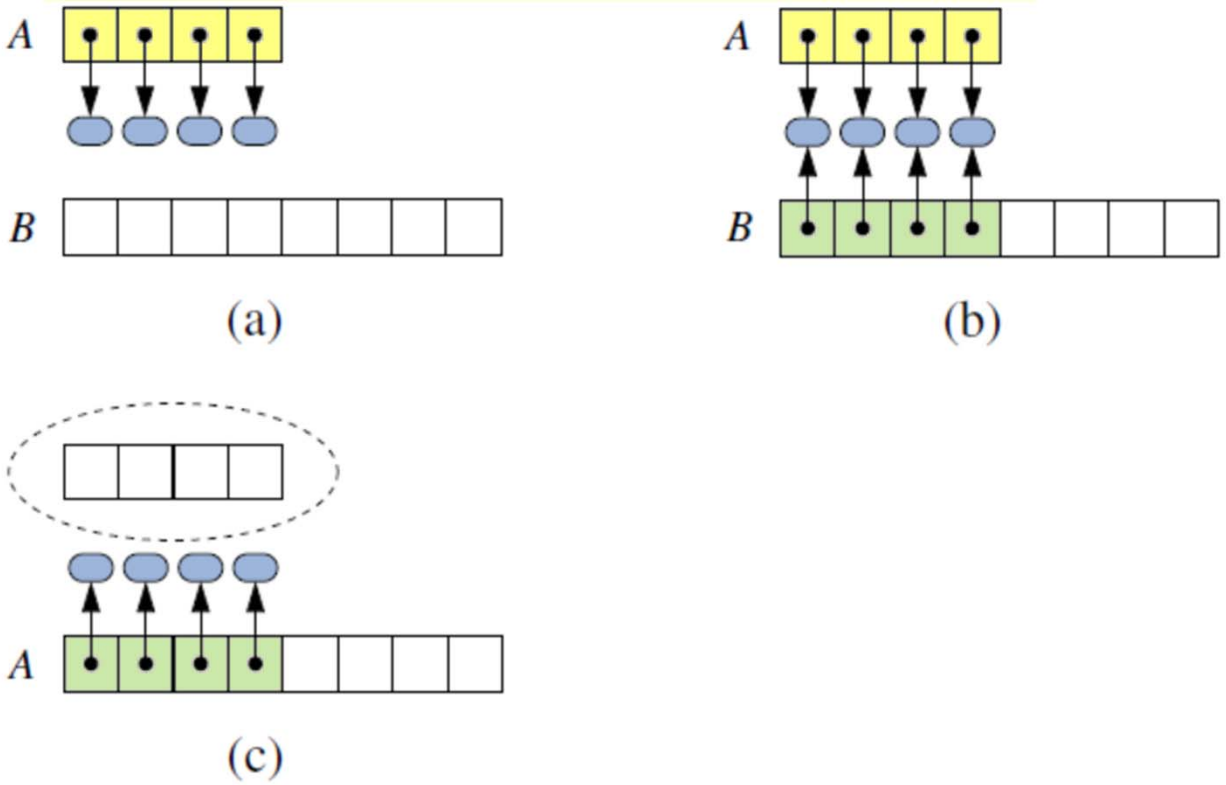
- A drawback of ArrayList
  - A fixed capacity needs to be declared
  - We may not know the maximum list size in advance

- Dynamic Arrays
  - When the list is full, increase the array capacity
  - Copy the contents from old array to the new one
    - We cannot simply allocate the new array at the end of the existing one



Implementing a Dynamic Array

- When adding a new element to a full array



```
public class DynamicArrayList<E> extends ArrayList<E> {
    public DynamicArrayList()              { this(CAPACITY); }
    public DynamicArrayList(int capacity) { super(capacity); }

    public void add(int i, E e) throws IndexOutOfBoundsException {
        checkIndex(i, size + 1);

        if(size == data.length)          //Dynamic array: when full,
            resize(2 * data.length);     //     double the capacity

        super.add(i, e);
    }

    @SuppressWarnings("unchecked")
    protected void resize(int capacity) {
        E[] tmp = (E[]) new Object[capacity];

        for(int k = 0; k < size; k++)
            tmp[k] = data[k];

        data = tmp;
    }
```

Amortization of a Dynamic Array

- amortization (by Oxford dictionary)

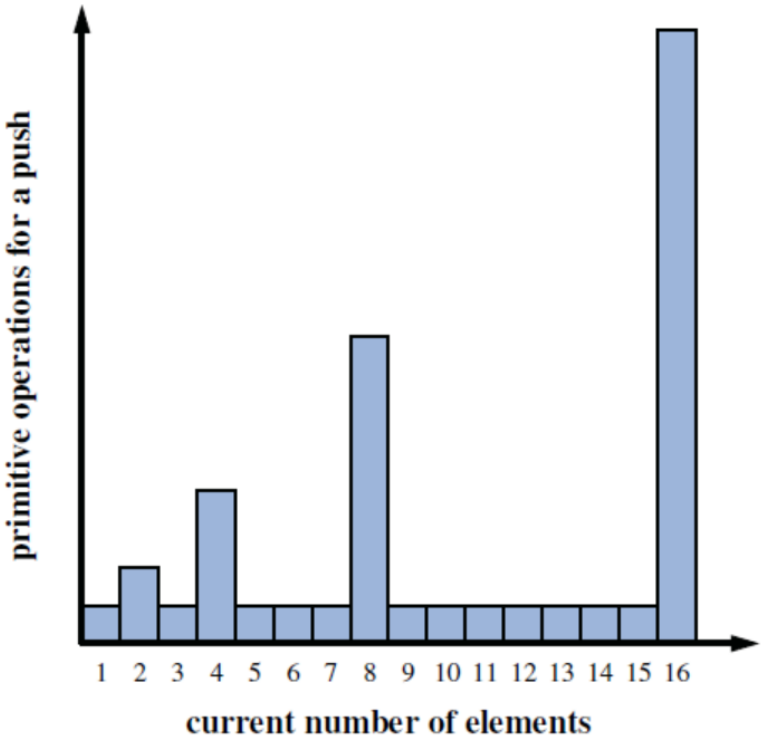
  See amortize

- amortize

  Gradually write off the initial cost (of an asset) over a period...

Amortization of a Dynamic Array

- push operation: adding an element to the last position of a list
  - No shift operations



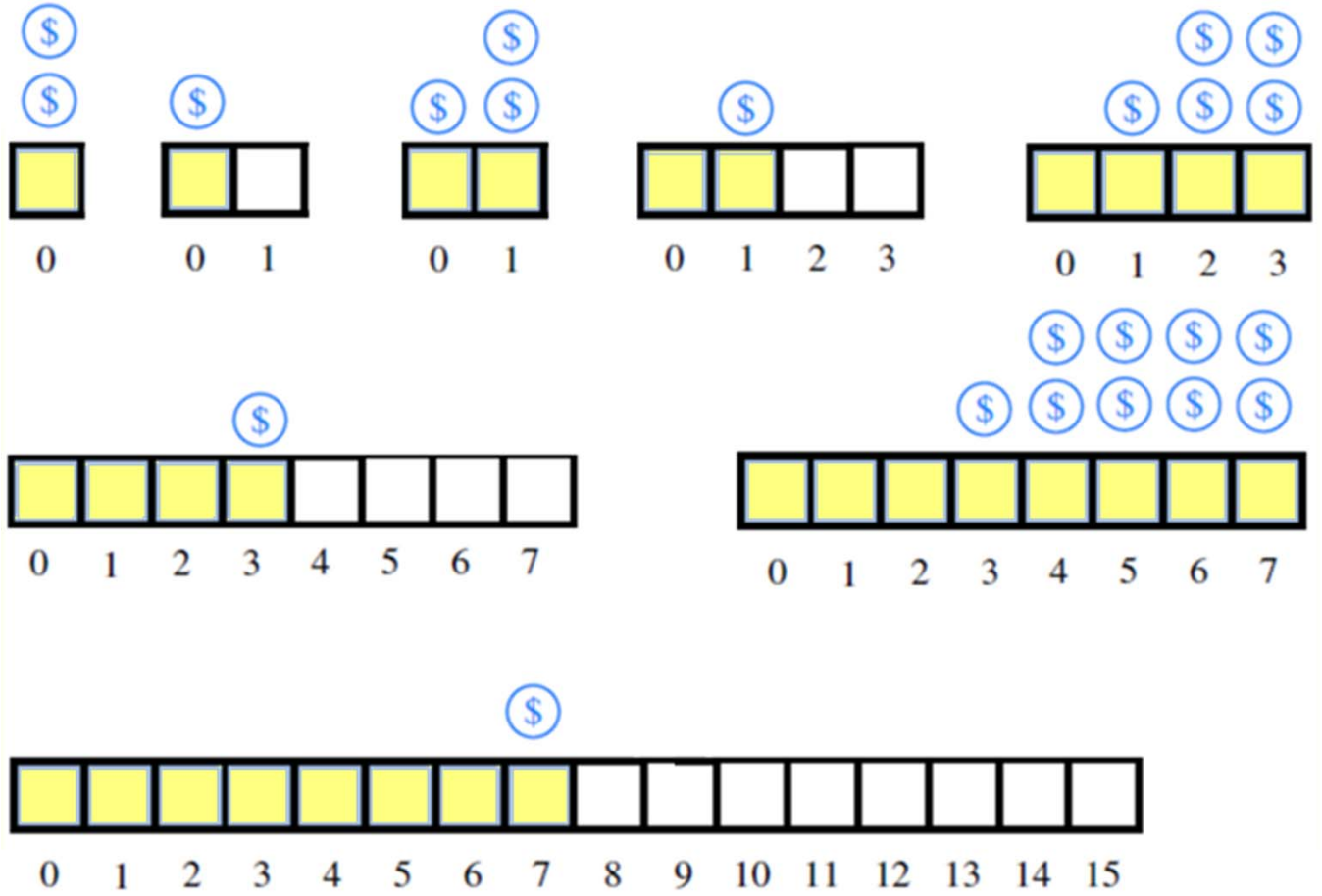
Amortization of a Dynamic Array

- Proposition
  - Let L be an initially empty array list with capacity one, implemented by means of a dynamic array that doubles in size when full
  - The total time to perform a series of n push operations in L is O(n)
  - Question: what is the big O if we increase the array size by 1 for each push operation?

Amortization of a Dynamic Array

- Justification
  - Suppose that every time you push, you are using $3 of operations although the required operations is $1
    - $1 for the push operation now
    - $1 for the copy of this element on resizing
    - $1 for the copy of a previous element on resizing
  - When the array of size 2^i is resized to 2^{i+1}
    - $\$2^{i-1}$ is used for copying the second 2^{i-1} elements to the new array
    - $\$2^{i-1}$ is used for copying the first 2^{i-1} elements that already consumed their savings

Amortization of a Dynamic Array



Position-Based List

- Integer indexes
  - Provide an excellent way to describe locations
  - Efficient way to access array elements
  - Not so efficient for linked lists

- Position
  - Abstracts a position in a list
  - Works well for linked lists

```
public interface Position<E> {
    E getElement() throws IllegalStateException;
}

public interface PositionalList<E>
                  extends Iterable<Position<E>>
{
    public int size();
    public boolean isEmpty();

    //position update methods
    public Position<E> first();
    public Position<E> last ();
    public Position<E> before(Position<E> pos);
    public Position<E> after (Position<E> pos);

    //list update methods
    public Position<E> addFirst(E e);
    public Position<E> addLast (E e);
    public Position<E> addBefore(Position<E> pos, E e);
    public Position<E> addAfter (Position<E> pos, E e);
    public E set(Position<E> pos, E e);
    public E remove(Position<E> pos);
}
```

With Position, these are possible

```
public class PositionalDblLinkedList<E> implements PositionalList<E> {
    private static class Node<E> implements Position<E> {
        private E e;
        private Node<E> next, prev;
        public Node(E e, Node<E> p, Node<E> n)
                                        { this.e = e; prev = p; next = n; }
        public E getElement()           { return e; } //interface Position
        public void setElement(E e)     { this.e = e; }
        public Node<E> getPrev()        { return prev; }
        public Node<E> getNext()        { return next; }
        public void setPrev(Node<E> p) { prev = p; }
        public void setNext(Node<E> n) { next = n; }
    }

    private Node<E> head;
    private Node<E> tail;
    private int size;

    public PositionalDblLinkedList() {
        head = new Node<E>(null, null, null);
        tail = new Node<E>(null, head, null);
        head.setNext(tail);
    }
```

```
//helper methods
private Position<E> addBetween(E e, Node<E> pred, Node<E>succ) {
    Node<E> newest = new Node<E>(e, pred, succ);
    pred.setNext(newest);
    succ.setPrev(newest);
    size++;
    return newest;
}
```

Implicit cast to Position<E>

```
private Position<E> toPosition(Node<E> node) {
    if(node == head || node == tail)
        return null;
    return node;
}

private Node<E> castOrThrow(Position<E> pos) {
    Node<E> node = (Node<E>) pos;
    if(node.getNext() == null)
        throw new IllegalArgumentException("pos is not in the list");
    return node;
}
```

```
//interface PositionalList
public int size()        { return size; }

public boolean isEmpty() { return size == 0; }

//position update methods
public Position<E> first() {
    return toPosition(head.getNext());
}

public Position<E> last() {
    return toPosition(tail.getPrev());
}

public Position<E> before(Position<E> pos) {
    Node<E> node = castOrThrow(pos);
    return toPosition(node.getPrev());
}

public Position<E> after(Position<E> pos) {
    Node<E> node = castOrThrow(pos);
    return toPosition(node.getNext());
}
```

```
//list update methods
public Position<E> addFirst(E e) {
    return addBetween(e, head, head.getNext());
}

public Position<E> addLast(E e) {
    return addBetween(e, tail.getPrev(), tail);
}

public Position<E> addBefore(Position<E> pos, E e) {
    Node<E> node = castOrThrow(pos);
    return addBetween(e, node.getPrev(), node);
}

public Position<E> addAfter(Position<E> pos, E e) {
    Node<E> node = castOrThrow(pos);
    return addBetween(e, node, node.getNext());
}

public E set(Position<E> pos, E e) {
    Node<E> node = castOrThrow(pos);
    E old = node.getElement();
    node.setElement(e);
    return old;
}
```

```
public E remove(Position<E> pos) {
    Node<E> node = castOrThrow(pos);
    Node<E> pred = node.getPrev();
    Node<E> succ = node.getNext();
    pred.setNext(succ);
    succ.setPrev(pred);
    size--;
    E old = node.getElement();
    node.setNext(null); //invalidate the position
    node.setPrev(null);
    node.setElement(null);
    return old;
}
```

Method	Running Time
size( )	$O(1)$
isEmpty( )	$O(1)$
first( ), last( )	$O(1)$
before(p), after(p)	$O(1)$
addFirst(e), addLast(e)	$O(1)$
addBefore(p, e), addAfter(p, e)	$O(1)$
set(p, e)	$O(1)$
remove(p)	$O(1)$

Iterator

- Will be explained later

```
private class NodeIterator implements Iterator<Position<E>>{
    private Node<E> curr;
    private NodeIterator()    { curr = head; }
    //true if any element is not returned by next()
    public boolean hasNext()  { return curr.next != tail; }
    //move the position and return the next unhandled element
    public Position<E> next() { curr = curr.next; return curr; }
}

public Iterator<Position<E>> iterator() {
    return new NodeIterator();
}
```

```
    //testing...
    private static void onFalseThrow(boolean b) {
        if(!b)
            throw new RuntimeException("Error: unexpected");
    }

    public static void main(String[] args) {
        PositionalDblLinkedList<Integer> list =
                                    new PositionalDblLinkedList<Integer>();
        Position<Integer> pos;
        pos = list.addLast(2);
        pos = list.addBefore(pos, 1);
        pos = list.after(pos);
        pos = list.addAfter(pos, 3);
        pos = list.last();
        pos = list.addLast(4);
        pos = list.first();
        onFalseThrow(list.remove(list.first()) == 1);
        onFalseThrow(list.remove(list.last())  == 4);
        onFalseThrow(list.remove(list.first()) == 2);
        onFalseThrow(list.remove(list.first()) == 3);
        System.out.println("Success!");
    }
}
```

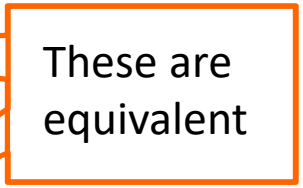
Iterator

- Iterator
  - A software design pattern that abstracts the process of scanning through a sequence of elements
- java.until.Iterator interface
  - hasNext(): returns true iff there is at least one additional element
  - next(): returns the next element
- java.lang.Iterable interface for a collection
  - iterator(): returns an iterator of the elements

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class IterableDynArrayList<E>
             extends DynamicArrayList<E>
             implements Iterable<E> {

    private class ArrayIterator implements Iterator<E> {
        private int i;

        //interface Iterator
        public boolean hasNext() {  //whether there is additional element
            return i < size;        //size is from DynamicArrayList
        }

        public E next() {           //return the next element
            if(i >= size)
                throw new NoSuchElementException("no such elements");

            return data[i++];       //data is from DynamicArrayList
        }
    }
```

```
//constructors
public IterableDynArrayList() {
    this(CAPACITY);
}

public IterableDynArrayList(int capacity) {
    super(capacity);
}

//interface Iterable
public Iterator<E> iterator() {
    return new ArrayIterator();
}
```

```
public static void main(String[] args) {
    IterableDynArrayList<Integer> list =
                            new IterableDynArrayList<Integer>(1);
    list.add(list.size(), 2);
    list.add(list.size(), 3);
    list.add(list.size(), 4);
    list.add(0, 1);
    list.add(0, 0);

    int i = 0;
    for(Integer j : list)
        onFalseThrow(j == i++);

    i = 0;
    Iterator<Integer> iter = list.iterator();
    while(iter.hasNext()) {
        Integer j = iter.next();
        onFalseThrow(j == i++);
    }
…
```



```
…
        onFalseThrow(list.remove(list.size()-1) == 4);
        onFalseThrow(list.remove(list.size()-1) == 3);
        onFalseThrow(list.remove(1) == 1);
        onFalseThrow(list.remove(0) == 0);
        onFalseThrow(list.remove(list.size()-1) == 2);
        System.out.println("Success!");
    }
}
```

Programming Assignment 3

- Implement the followings
  - `CircularlyDblLinkedList`, which is a circularly list based on a doubly linked list
  - Using `CircularlyDblLinkedList`, implement `SetImpl`, which is a `Set`
  - Implement `Subset` which is a `BooleanAlgebra`
  - Your implementation should pass all unit test cases included in the zip file at the minimum
- Submit CircularlyDblLinkedList.java, SetImpl.java and Subset.java in a single zip file
- Due date: 4/5/2022

```
public class App {
    public static void main(String[] args) {
        UnitTest.testBool();
        UnitTest.testSubset();
    }
}

public class UnitTest {
...
    public static void testSubset() {
        System.out.println("testSubset...");
        Int _1 = new Int(1);
        Int _2 = new Int(2);
        Int _3 = new Int(3);
        SetImpl<Int> x = new SetImpl<Int>(new Int[] {_1, _1, _1});
        SetImpl<Int> y = new SetImpl<Int>(new Int[] {_1, _2, _1, _2});
        SetImpl<Int> z = new SetImpl<Int>(new Int[] {_3, _2, _3, _3});
        SetImpl<Int> u = new SetImpl<Int>(new Int[] {_1, _2, _3});
        Subset<Int> a = new Subset<Int>(x, u);
        Subset<Int> b = new Subset<Int>(y, u);
        Subset<Int> c = new Subset<Int>(z, u);

        testAll(a, b, c);
        testAll(new Duel(a), new Duel(b), new Duel(c));
        System.out.println("testSubset done");
    }
```

```java
//interface List
public interface List<E> {
    //get the size of the list
    public int size();

    //is the list empty?
    public boolean isEmpty();

    //get element at index i
    public E get(int i) throws IndexOutOfBoundsException;

    //set e at index i
    public E set(int i, E e) throws IndexOutOfBoundsException;

    //add e at index i
    public void add(int i, E e) throws IndexOutOfBoundsException;

    //add e at the last index
    public void add(E e) throws IndexOutOfBoundsException;

    //remove node a index i and return its element
    public E remove(int i) throws IndexOutOfBoundsException;
}
```

```java
//interface Set

public interface Set<E extends Ordered> {
    //whether this is equal to set
    public boolean isEqual(Set<E> set);

    //return this union set
    public Set<E> union(Set<E> set);

    //return this intersection with set
    public Set<E> intersection(Set<E> set);

    //return this - set
    public Set<E> difference(Set<E> set);
}
```

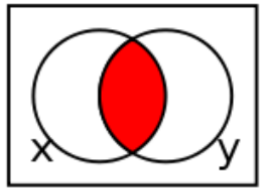
```
//interface BooleanAlgebra

public interface BooleanAlgebra {
    //or operation
    public BooleanAlgebra or(BooleanAlgebra a);

    //and operation
    public BooleanAlgebra and(BooleanAlgebra a);

    //not operation
    public BooleanAlgebra not();

    //identity of the or operation
    public BooleanAlgebra orIdentity();

    //identity of the and operation
    public BooleanAlgebra andIdentity();

    //Additional methods: whether a is equal to this
    public boolean isEqual(BooleanAlgebra a);
}
```

```
import java.util.Iterator;

public class CircularlyDblLinkedList<E> implements List<E>, Iterable<E> {
    protected static class Node<E> {
        public E e;
        public Node<E> prev, next;
        public Node() {
            this.e = null; this.prev = this; this.next = this;
        }
        public Node(E e, Node<E> prev, Node<E> next) {
            this.e = e; this.prev = prev; this.next = next;
        }
    }
    public static class NodeIterator<E> implements Iterator<E> {
        private Node<E> head, curr;
        public NodeIterator(Node<E> head) {
            this.head = head; this.curr = head.next;
        }
        //TODO: implement Iterator<E>
    }

    protected Node<E> head;
    protected int size;
```

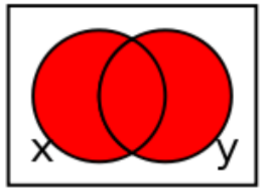
```
//constructor
public CircularlyDblLinkedList() {
    head = new Node<E>();
    size = 0;
}

//TODO: implement interface List
public E get(int i) {
    return findNode(i).e;
}

//TODO: implement interface Iterable

//helper methods
protected Node<E> findNode(int i) {
    if(i < 0 || i >= size)
        throw new IndexOutOfBoundsException(
          "invalid index: " + i + " is not in [ 0, " + size + ")");

    //TODO: find the node at index i and return it
}
```

```
public class SetImpl<E extends Ordered> implements Set<E> {
    private CircularlyDblLinkedList<E> list;
    public SetImpl() {
        list = new CircularlyDblLinkedList<E>();
    }
...
    public SetImpl(E[] arr) {
        list = new CircularlyDblLinkedList<E>();
        for(int i = 0; i < arr.length; i++)
            list.add(i, arr[i]);
        dedupe();
    }

    //TODO: implement interface Set

    //helper methods
    private CircularlyDblLinkedList<E> copyList() {
...
    }

    //TODO: remove duplicated elements (do not sort the list)
    private void dedupe() {
    }
```

```
//a = Subset({1,2}, {1,2,3,4}), b = Subset({2,3}, {1,2,3,4});
//a and b = Subset({2},     {1,2,3,4}) : intersection
//a or  b = Subset({1,2,3}, {1,2,3,4}) : union
//not a   = Subset({3,4},   {1,2,3,4}) : complement

public class Subset<E extends Ordered> implements BooleanAlgebra {
    //a subset of univ
    private Set<E> subset;

    //universal set
    private Set<E> univ;

    public Subset(Set<E> subset, Set<E> univ) {
        this.subset = subset.union(new SetImpl<E>());
        this.univ = univ.union(new SetImpl<E>());
    }

    //interface BooleanAlgebra
    public BooleanAlgebra or(BooleanAlgebra a) {
        //TODO: return the union of this and a
    }

    public BooleanAlgebra and(BooleanAlgebra a) {
        //TODO: return the intersection of this and a
    }
```



x∧y



x∨y

¬x

```
public BooleanAlgebra not() {
    //TODO: return univ - this
}

public BooleanAlgebra orIdentity() {
    //TODO: return the or identity
}

public BooleanAlgebra andIdentity() {
    //TODO: return the and identity
}

public boolean isEqual(BooleanAlgebra a) {
    Subset<E> s = castOrThrow(a);
    return subset.isEqual(s.subset) && univ.isEqual(s.univ);
}
```

Optional: Boolean Algebra

- Let a structure $(B, +, \cdot, \sim, 0, 1)$ be an algebra
  - B is a set that contains 0 and 1
  - $+$ and $\cdot$ are binary operators and $\sim$ is a unary operator

- The structure is a Boolean algebra if for all $x, y, z \in B$
  - $x + y = y + x$
  - $x \cdot (y + z) = x \cdot y + x \cdot z$
  - $x + 0 = x$
  - $x + \sim x = 1$
  - $0 \neq 1$
  - $x \cdot y = y \cdot x$
  - $x + (y \cdot z) = (x + y) \cdot (x + z)$
  - $x \cdot 1 = x$
  - $x \cdot \sim x = 0$

```
public class Bool implements BooleanAlgebra {
    public boolean b;

    public Bool(boolean b) { this.b = b; }

    //interface BooleanAlgebra
    public BooleanAlgebra or(BooleanAlgebra a) {
        return new Bool(b || ((Bool)a).b);
    }
    public BooleanAlgebra and(BooleanAlgebra a) {
        return new Bool(b && ((Bool)a).b);
    }
    public BooleanAlgebra not() {
        return new Bool(!b);
    }
    public BooleanAlgebra orIdentity() {
        return new Bool(false);
    }
    public BooleanAlgebra andIdentity() {
        return new Bool(true);
    }
    public boolean isEqual(BooleanAlgebra a) {
        return b == ((Bool)a).b;
    }
}
```

```
//a = Prime(10, 30), b = Prime(6, 30); 10 = 2*5, 6 = 2*3, 30 = 1*2*3*5
//a and b = Prime(2, 30);  2  = gcd(10, 6)
//a or  b = Prime(30,30);  30 = lcm(10, 6)
//not a   = Prime(3, 30);  3  = 30/10

public class Primes implements BooleanAlgebra {
    private int n, p;

    //constructor: primes is products of prime numbers
    public Primes(int n, int p) {
        this.n= n; this.p = p;
    }

    //interface BooleanAlgebra
    public BooleanAlgebra or(BooleanAlgebra a) {
        Primes i = castOrThrow(a);
        return new Primes(lcm(n, i.n), p);
    }
    public BooleanAlgebra and(BooleanAlgebra a) {
        Primes i = castOrThrow(a);
        return new Primes(gcd(n, i.n), p);
    }
```

```
    public BooleanAlgebra not() {
        return new Primes(p/n, p);
    }
    public BooleanAlgebra orIdentity() {
        return new Primes(1, p);
    }
    public BooleanAlgebra andIdentity() {
        return new Primes(p, p);
    }
    public boolean isEqual(BooleanAlgebra a) {
        Primes i = (Primes)a;
        return n == i.n && p == i.p;
    }

    //privates
    private Primes castOrThrow(BooleanAlgebra a) {
        Primes i = (Primes)a;
        if(p != i.p)
            throw new IllegalArgumentException("Unmatched primes");
        return i;
    }
    private int gcd(int a, int b) { ... }
    private int lcm(int a, int b) { return a * b / gcd(a, b); }
}
```