Lecture 6
Linked Lists vs. Arrays

Zonghua Gu (based on Jianchen Shan)
Department of Computer Science
Hofstra University

Lecture Goals

- Describe the difference between an Abstract Data Type (ADT) and a Data Structure
- Describe and draw the structure of a LinkedList
- Create Generic classes in Java
- Use thrown Exceptions to indicate errors
- Create a doubly linked list with sentinel nodes in Java
- Write tests for a LinkedList
- Describe advantages of a LinkedList over an ArrayList
- Compare advantages in testing methodologies

Key CS Idea: Abstraction

Hiding irrelevant details to focus on the essential features needed to understand and use a thing

Abstraction example: car brakes

driver

Allows us to drive our cars without being a mechanic

Behavior specified

How do they work?

mechanic

Implementation specified

Abstraction Barrier
sets the rules of interaction

Data Abstraction:

<<interface>> List
add(Object) size() etc.

ArrayList

LinkedList

Data Structure Specific implementation

User of libraries

1. language independent
2. interfaces or abstract classes in Java

Abstract Data Type (ADT) No implementation

Library developer

1. fulfill an ADT contract
2. affect the performance

Linked Lists vs. Arrays

ADT

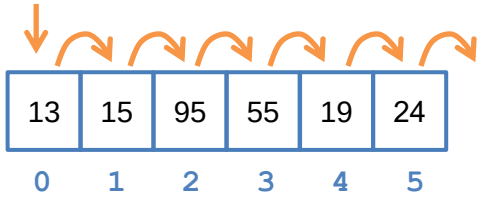
<<interface>>
List
add(Object)
size()
etc.

LinkedList ArrayList

Data Structures

Two ways to implement the same functionality

ADT specifies functionality, but not efficiency!



- sequence of elements in contiguous memory

An ArrayList implements the List interface using an array

- can access elements in constant time

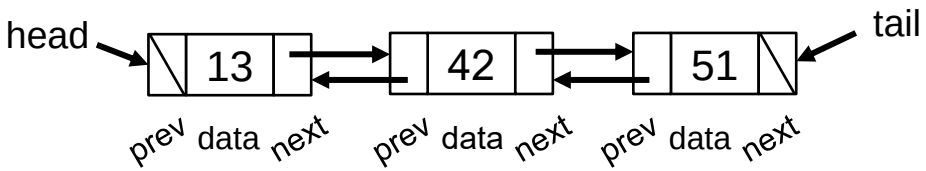
How long does it take to add an element to the front of an ArrayList?

O(n)

- move all elements if array is not full
- copy elements to new array if array is full
- not efficient for add operation

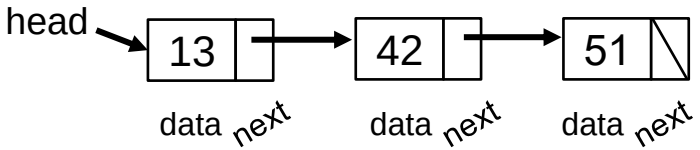
- Doubly Linked List, in pictures

- noncontiguous in memory



A LinkedList implements the List interface using a LinkedList data structure

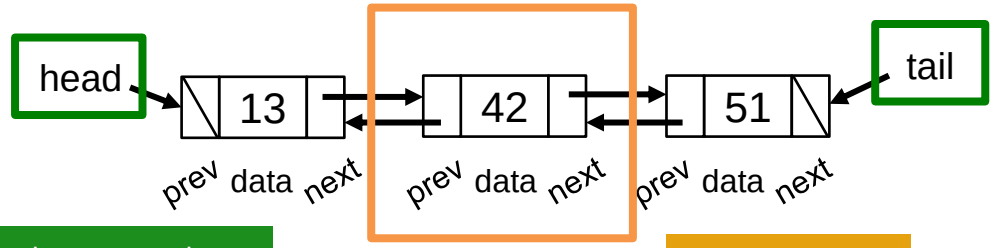
- Singly Linked List, in pictures



- more efficient for inserting elements
- Some functionality is easier to implement with the doubly linked list, **let's implement it in java**

https://www.youtube.com/watch?v=R9PTBwOzceo

Two Classes in a LinkedList

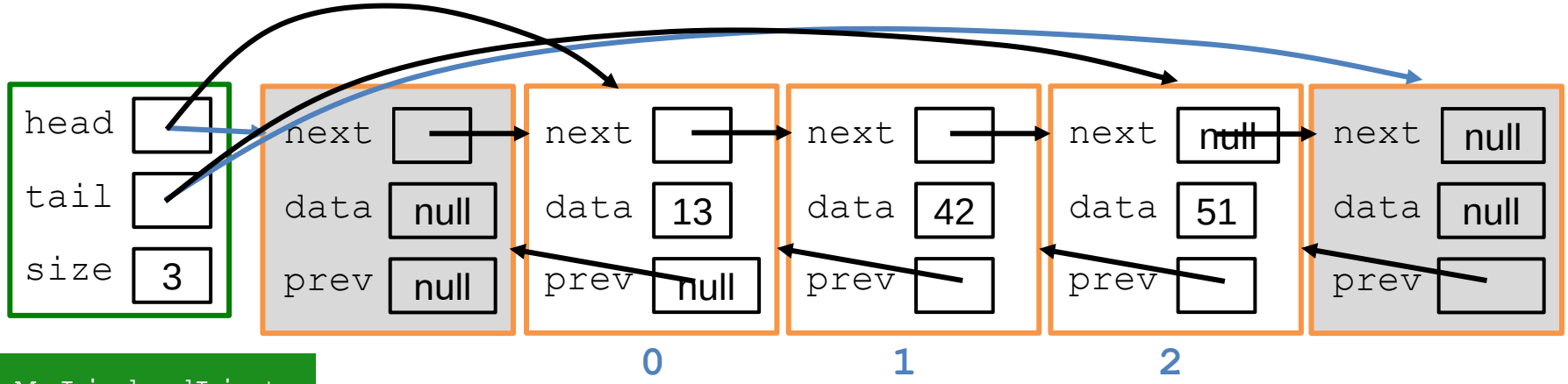


MyLinkedList

ListNode

What will be the type of the fields prev and next in the ListNode class?

These fields store references to other list elements, each of which is an object of type ListNode.



MyLinkedList object

ListNode Objects

Sentinel or dummy nodes

- They make implementation of the LinkedList functionality, slightly easier

How long does it take to access an element in a LinkedList implementation (in the worst case)?

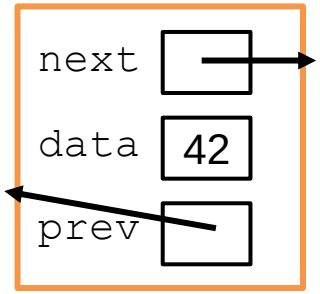
O(n)

In the worst case, we are accessing an element in the middle of the list and so we need to follow O(n/2) references from the head (or the tail).

Use Type Parameter to Create Generic Classes

```
public class ListNode<E> {
        ListNode<E> next;
prev;
        E data;
}
```

ListNode<E>



The ListNode class is the backbone of the linked list structure

What is E?

Type parameter. Our ListNode is "generic"

Meaning that type can be passed in when we create one of these ListNodes and it allows us to make our list structure be generic

```
public class RememberLast<T> {
        private T lastElement;
        private int numElements;
        public RememberLast () {
                numElements = 0;
                lastElement = null;
        }
        public T add(T element) {
                T prevLast = lastElement;
                lastElement = element;
                numElements++;
        return prevLast;
        }
}
```

```
// Somewhere else...
RememberLast<Integer> rInt =
  new RememberLast<Integer>();
RememberLast<String> rStr =
  new RememberLast<String>();
rInt.add(3);
rStr.add("Happy");
```

Java autoboxes ints into objects

We can't instantiate a generic class with primitive types. It has to be an object type though.

Example: Parameterized types

- Integer is class, int is a primitive type

byte has Byte
short has Short
int has Integer
long has Long
boolean has Boolean
char has Character
float has Float
double has Double

Handle Bad Inputs with Exceptions

```
public class RememberLast<T> {
        // Code omitted here
        public T add(T element) {
                if(element == null) {
                        <<WHAT GOES HERE?>>
                }
                T prevLast = lastElement;
        lastElement = element;
                numElements++;
        return prevLast;
        }
}
```

A. Return -1 to flag the bad input.

B. Return null to flag the bad input.

C. Cause an error that stops normal program execution.

A. Doesn't work. Must return a T

B. Not enough for fatal error

C. ✓

Throw exceptions to indicate fatal problems

Not required since NPE is unchecked, but OK

```
public class RememberLast<T> {
        public T add(T element) throws NullPointerException {
                if(element == null) {
                        throw new NullPointerException("Handled
                            by compiler: the element is empty");
                }
                T prevLast = lastElement;
                lastElement = element;
                numElements++;
        return prevLast;
        }
        public static void main(String args[]){
                RememberLast<Integer> rInt = new
                                        RememberLast<Integer>();
                rInt.add(null);
        }
}
```

Handled by compiler

```
Exception in thread "main" java.lang.NullPointerException:
Handled by compiler: the element is empty
        at RememberLast.add(RememberLast.java:11)
        at RememberLast.main(RememberLast.java:23)
```

```
public class RememberLast<T> {
        public T add(T element) throws NullPointerException {
                try {
                        if(element == null) {
                                throw new NullPointerException("Handled
                                        by compiler: the element is empty");
                        }
                }
                catch(NullPointerException e) {
                        System.out.println("Handled by program: cannot
                                                store null pointers");
                }
                T prevLast = lastElement;
        lastElement = element;
                numElements++;
        return prevLast;
        }
        public static void main(String args[]){
                RememberLast<Integer> rInt = new RememberLast<Integer>();
                rInt.add(null);
        }
```

Handled by programmer

```
Handled by program: cannot store null pointers
```

[1] https://www.geeksforgeeks.org/throw-throws-java/ [2] https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/

Java Code for a Linked List

Default value is null

```
class ListNode<E> {
        ListNode<E> next;
        ListNode<E> prev;
        E data;
        public ListNode(E theData) {
                this.data = theData;
    }
}
```

Recursive data type!

No type parameter in the constructor header

using references of the class itself inside the class we're defining.

- Why can we use a class when we're not even done defining?

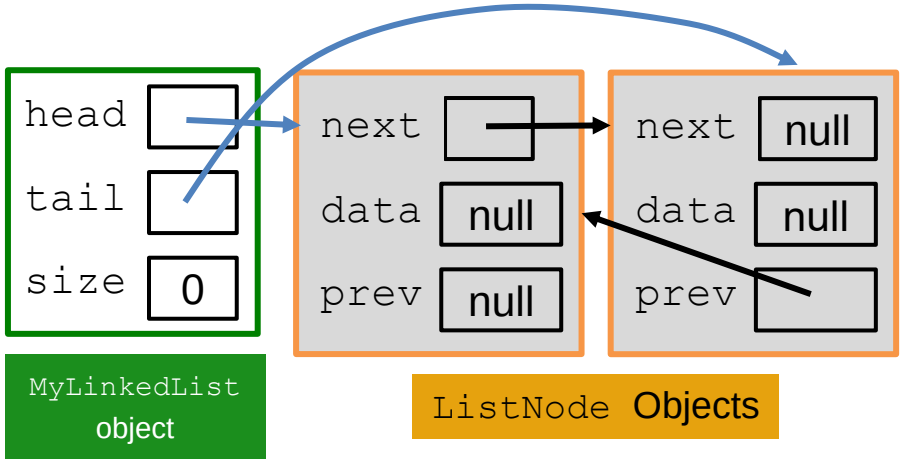
Because by the time Java actually creates any of these ListNode objects, the class definition will be finished.

```
public class MyLinkedList<E>
{
        private ListNode<E> head;
        private ListNode<E> tail;
        private int size;
        public MyLinkedList() {
                size = 0;
                head = new ListNode<E>(null);
                tail = new ListNode<E>(null);
                head.next = tail;
                tail.prev = head;
        }
}
```

Now we've correctly setup an empty linked list! Let's implement size, get, set, add, remove

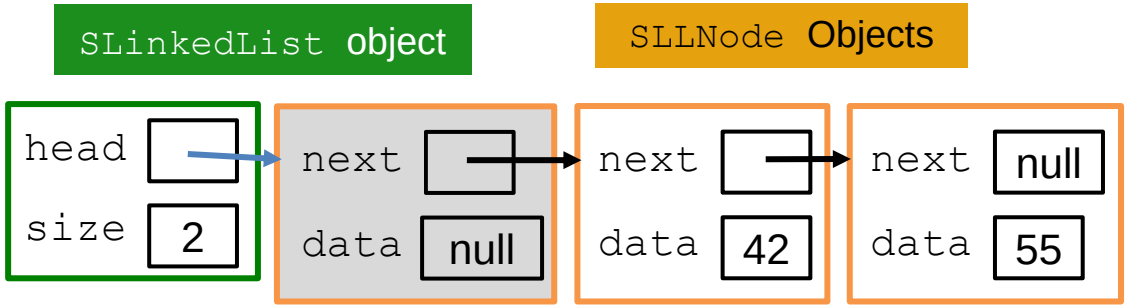
Does this constructor correctly create the diagram as shown below

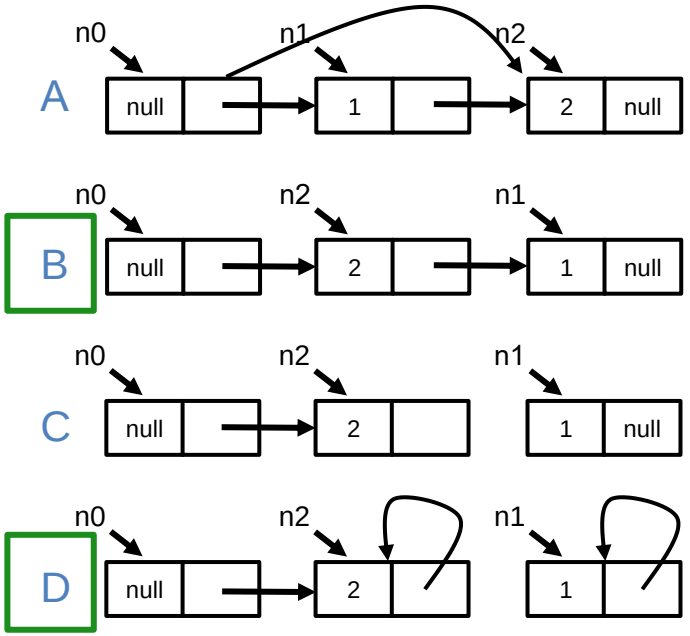
Need to link the two sentinel nodes to each other



This list is empty. It has zero data nodes, but two sentinel nodes

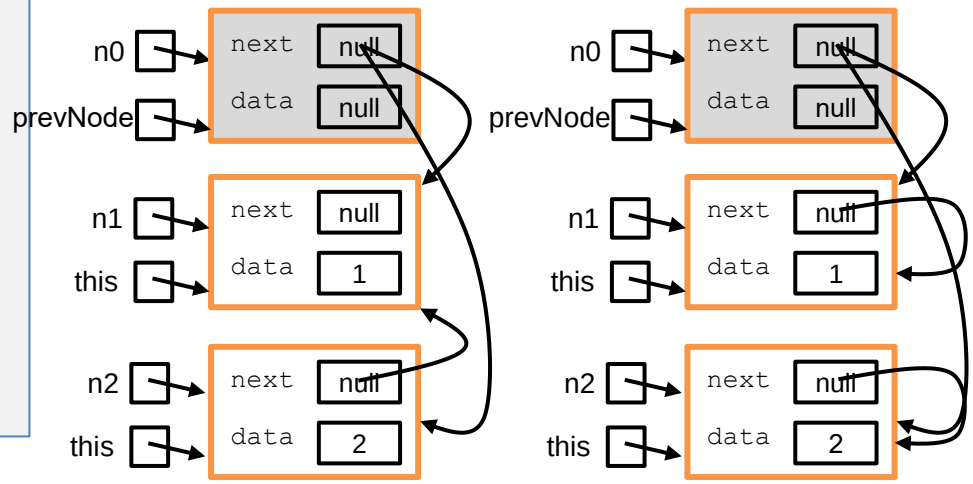
Practices with Singly Linked List






```
public class SLLNode<E> {
        SLLNode<E> next;
        E data;
        public SLLNode() {
                this.next = null; this.data = null;
        }
        public SLLNode(E theData) {
                this.data = theData;
        }
        public SLLNode(E theData, SLLNode<E> prevNode)
        {
                this(theData);
                prevNode.next = this;
                this.next = prevNode.next;
        }
        public static void main(String[] args) {
          SLLNode<Integer> n0 = new SLLNode<Integer>();
SLLNode<Integer> n1= new SLLNode<Integer>(1,n0);
          SLLNode<Integer> n2 = new SLLNode<Integer>(2,n0);
        }
```

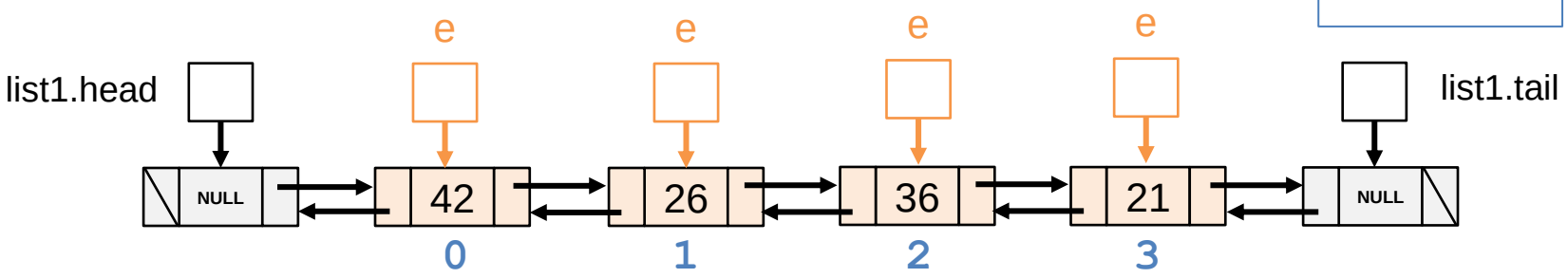
three constructors



Let's draw the diagram one line at a time for step-by-step walkthrough

Get Operation

index = 4




```
list1.getNode(2);
```

1. Check if index n is legal

2. Traverse the list to locate the node

Option A – always iterate from the head

```
e = head.next
e = e.next;
```

Option B – iterate from the tail if the node is in the second half

```
e = tail.prev
e = e.prev;
```

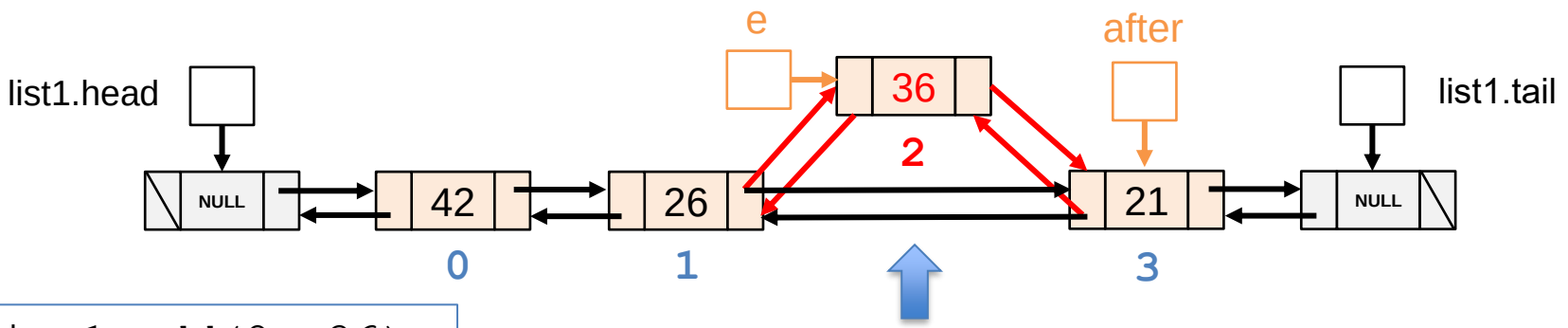
```
public List<E> e(int index) {
        if (index < 0 || index >= size)
                throw new IndexOutOfBoundsException("Index: "
            + index + ", Size:" + size);
        if (index < size / 2) {
                e = head.next;
                // n less than size/2, iterate from start
                while (index-- > 0)
                        e = e.next;
        } else {
                e = tail.prev;
                // n greater than size/2, iterate from end
                while (++index < size)
                        e = e.prev;
        }
        return e.data;
```

- what if we want to return the data?

```
public E get(int n) {
        ListNode<E> e = getNode(n);
        return e.data;
}
```

Size:0

Add Operation



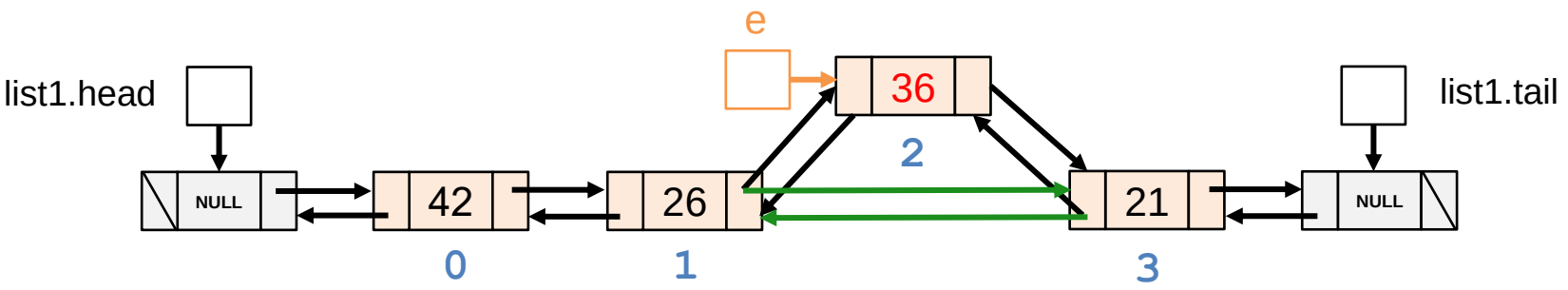

```
list1.add(2, 36);
```

1. Check if index n is legal
2. Create a new node
3. Locate the next node of the new node
   - If we add to the end, it is the tail
   - Otherwise, it is the getNode(index)
3. Insert the new node
4. Update the list size

General implementation to cover the special cases: addFirst, addLast

```
public void add(int index, E o) {
        if (index < 0 || index > size)
                throw new IndexOutOfBoundsException("Index: " + index + ",
Size:" + size);
        ListNode<E> e = new ListNode<E>(o);
        ListNode<E> after;
        if (index < size) {
                after = getNode(index);
        } else {
                after = tail;
        }
        e.next = after;
        e.prev = after.prev;
        after.prev.next = e;
        after.prev = e;
        size++;
}
```

Remove Operation




```
list1.remove(2);
```

1. Check if index n is legal
2. Locate the node by getNode(index)
3. Remove the node
4. Update the list size
5. Return the removed data

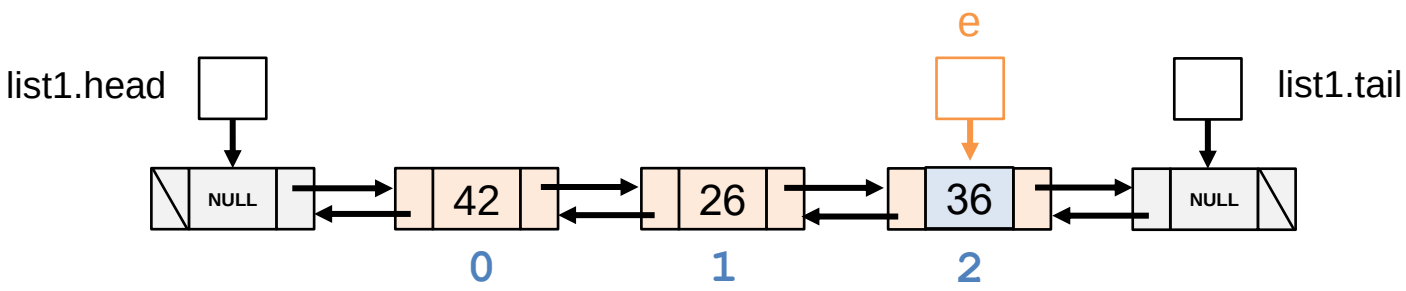
General implementation to cover the special cases: removeFirst, removeLast

```
public E remove(int index) {
        if (index < 0 || index >= size)
                throw new IndexOutOfBoundsException("Index:
                        " + index + ", Size:" + size);
        ListNode<E> e = getNode(index);
        e.next.prev = e.prev;
        e.prev.next = e.next;
        size--;
        return e.data;
}
```

```
list1.remove(0);
```

```
list1.remove(List1.getSize()-1);
```

Set Operation

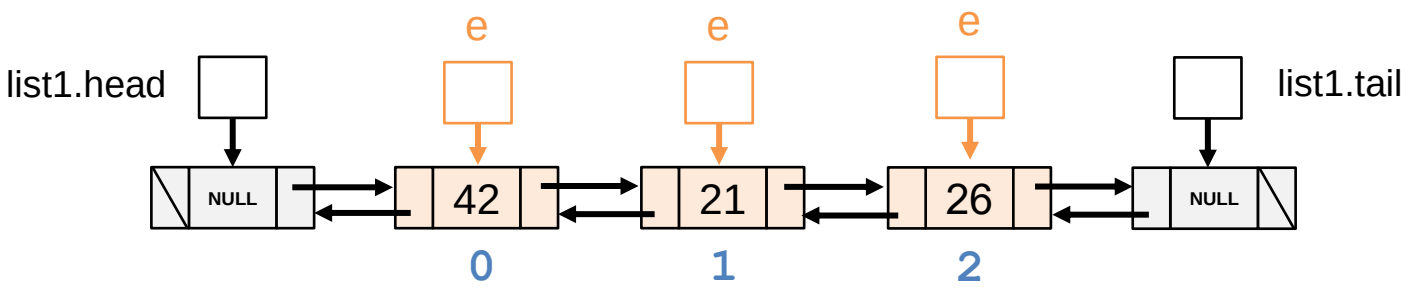



```
list1.set(2, 36);
```

1. Check if index n is legal
2. Locate the node by getNode(index)
3. Save the old data
4. Update the data
5. Return the old data

```
public E set(int index, E o)
{
        if (index < 0 || index >= size)
                throw new IndexOutOfBoundsException("Index:
                        " + index + ", Size:" + size);
        ListNode<E> e = getNode(index);
        E old = e.data;
        e.data = o;
        return old;
}
```

Contain Operation




```
list1.contains(26);
```

true

1. Iterate nodes from start

Compare the data

return true if we found

Otherwise, go to the next node

2. Return false if we don't find

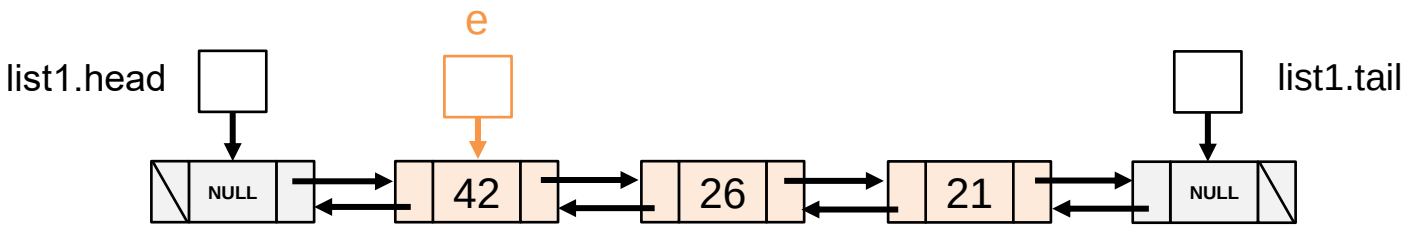
```
list1.contains(5);
```

false

```
public boolean contains(E o)
{
        ListNode<E> e = head.next;
        while (e.next != null)
        {
        if ((e.data).equals(o))
                return true;
        e = e.next;
        }
        return false;
}
```

if e.next is equal to null, e is the tail and iteration is finished

toString Operation




```
System.out.println(list1);
```

1. Create an empty string to represent the linked list

2. Iterate nodes from start

   Append the data of each node to the string

   Move to the next node

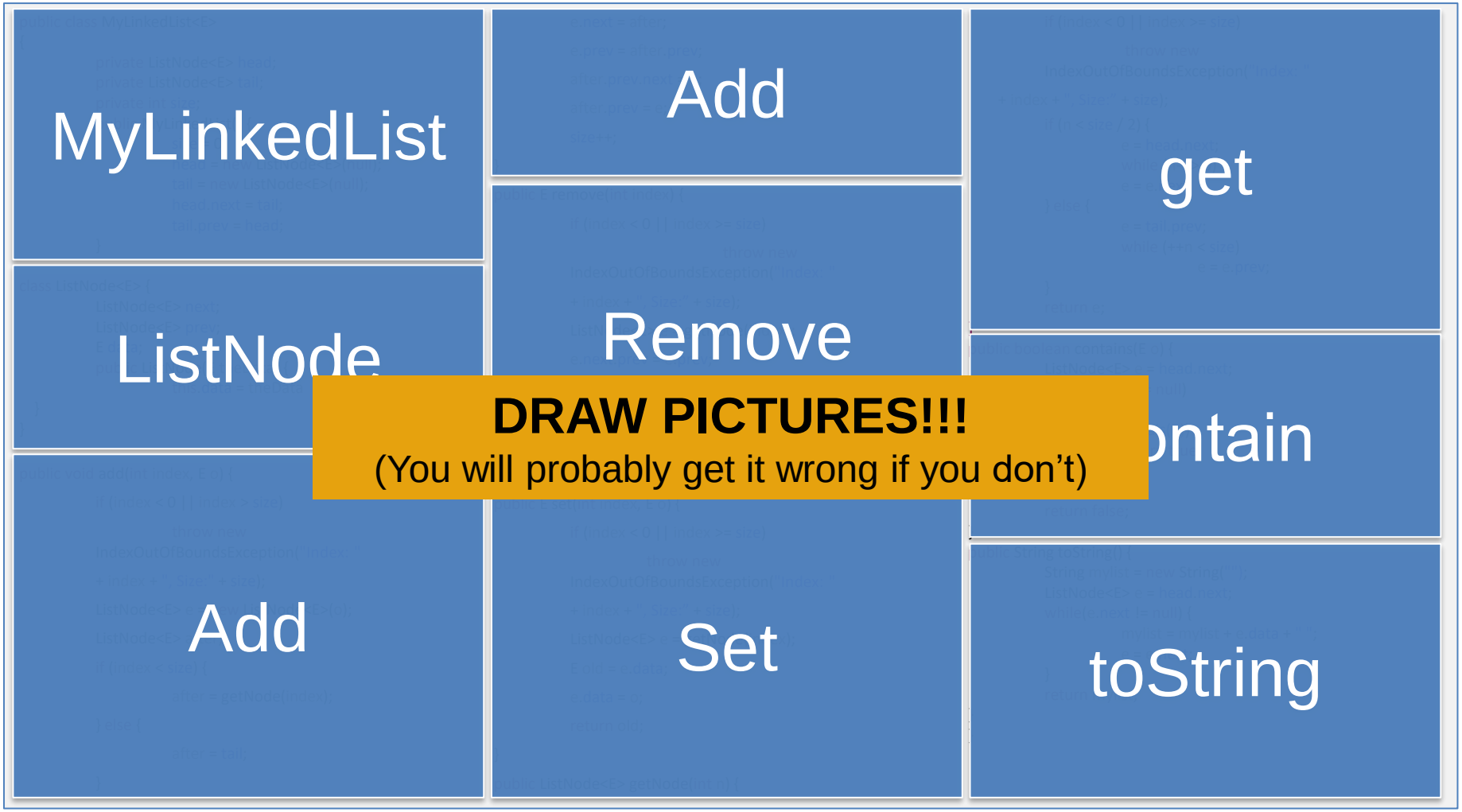
2. Return the string

```
$ 42 26 21
```

```
public String toString() {
        String mylist = new String("");
        ListNode<E> e = head.next;
        while(e.next != null) {
                mylist = mylist + e.data + " ";
                e = e.next;
        }
        return mylist;
}
```

if e.next is equal to null, e is the tail and iteration is finished

Java Code for a Linked List (Contd.)



ListNode<E> contains(E o), remove(E o), indexOf(E o), replace(E old, E new), toArray(), etc.

Testing and Confidence

Gain confidence in Correctness by Testing

How do we reason about confidence?

Different degrees of confidence apply

We need strong confidence about the correctness of the codes that impact people's lives.

Code State	Confidence
Written, hasn't compiled	Extremely low
Compiled, haven't run	Extremely low
Tested against basic input	Low
Tested against corner cases	Medium
Tested against users (beta testing)	Medium-High

Your code would be used by:
User, Hacker, Programmer, Yourself

Wait, can't I just test against all inputs?

An `int` input has more than four billion possible values. An array? A database?

How can we increase confidence?

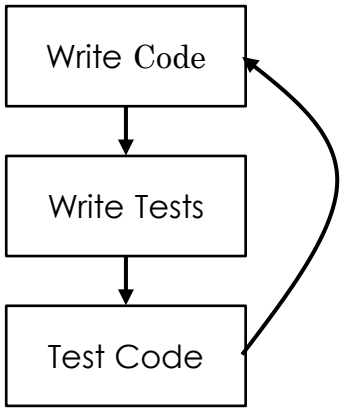
- Be critical of our algorithms/code
- Consider/test corner cases
- Attempt to formally reason about correctness
- Create automated test cases

test the code in a whole bunch of different situations

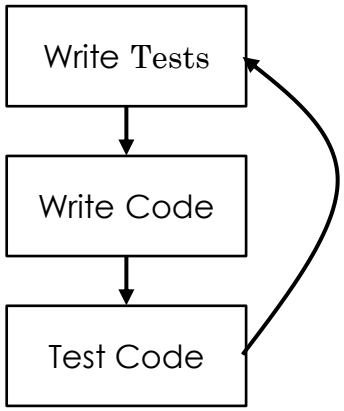
Let's do this next

Unit Testing

Standard Cycle



Test-Driven Development



How can you write code if you don't know how it will be tested?

Nearly everyone agrees, don't wait till the end to test!

When to test and the "best" way to develop code is contentious!

Black Box Testing

Only tests through the interface

Test Interface: Black Box

Abstraction Barrier sets the rules of interaction

Clear Box Testing

Tests which know about the implementation

Test Implementation: Clear Box

Okay, so what do we test?

Unit Testing!

JUnit

Allows us to write and run unit tests.

Should I test every statement, like this?

```
int a = 5;
if( a != 5 ) { System.exit(0); }
```

Way too fine-grain...

Okay, so should I wait for user alpha/beta testing?

✗ Way too late!!!

Usually – methods.

Which of the following are advantages for black box testing?

A. Is often more representative of user use of code

B. Is easier to write by someone unfamiliar with the implementation

C. Is more knowledgeable of potential corner cases which might cause incorrect behavior

JUnit Basic

JUnit is a lightweight Unit Testing Platform

```
assertEquals("Check first", "A", shortList.get(0));
```

Main components:

1. code to setup tests
2. code to perform tests
3. code to cleanup tests

Here, `assertEquals` enforces that `shortlist.get(0)` is ”A”. Otherwise, throws an error.

`emptyList.get(0)` should throw an exception, if it doesn’t, we call the `fail` mehtod.

```
try {
        emptyList.get(0);
        fail("Check out of bounds");
}
catch (IndexOutOfBoundsException e) {
}
```

@Before
`setup`
is run before each test to initialize variables and objects

@Test
`test<feature>`
denote method to test <feature>
Two useful methods:
`1.assertEquals`
`2.fail`

@After
`tearDown<feature>`
can be useful if your test constructed something which needs to be properly torn down (like a database)

Test Get Method of MyLinkedList with JUnit

Which of the following tests should I run? Try to avoid redundant tests.

A. Test get(0) from an empty list ✓ — Tests corner case (empty list)

B. Test get(-1) from a list with 2 element ✓ — Tests corner case (negative index)

C. Test get(0) from a list with 2 element ✓ — Tests standard use

D. Test get(1) from a list with 2 elements ✓ — Ensures we can get more than just the 1st element

E. Test get(2) from a list with 2 elements ✓ — Tests corner case (larger than size)

F. Test get(2) from a list with 3 elements ✗ — Redundant, what is new here?

Summary

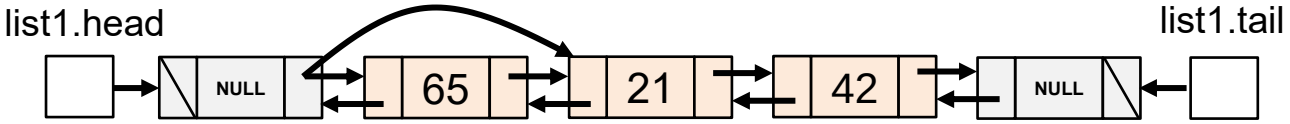
- Consider corner cases when testing
- Test common case use
- Remember testing has costs

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class MyLinkedListTester {

private MyLinkedList<String> shortList;
private MyLinkedList<Integer> emptyList;

@Before
public void setUp() throws Exception {
        shortList = new MyLinkedList<String>();
        shortList.add(0, "A");
        shortList.add(0, "B");
        emptyList = new MyLinkedList<Integer>();
        list1 = new MyLinkedList<Integer>();
}
```

```
@Test
public void testGet() {
        try {
                emptyList.get(0);
                fail("Check out of bounds");
        }
        catch (IndexOutOfBoundsException e) {}

        try {
                shortList.get(-1);
                fail("Check out of bounds");
        }
        catch (IndexOutOfBoundsException e) {}

assertEquals("Check first", "B", shortList.get(0));
assertEquals("Check second", "A", shortList.get(1));

        try {
                shortList.get(2);
                fail("Check out of bounds");
        }
        catch (IndexOutOfBoundsException e) {}
}
}
```

Test Remove Method of MyLinkedList with JUnit

assume that the list integrity is ensured by other tests



In `testRemove()` you run: `int a = list1.remove(0);`

A. Verify that a has the value 65 ✓
B. Call list1.get(-1) to check if it throws a NullPointerException ✗
C. Call list1.get(0) and check that index 0 has the value 21 ✓
D. Call list1.get(1) and check that index 1 has the value 42 ✗
E. Call list1.get(2) to check if it throws a NullPointerException ✗
F. Call list.size() to check if size is 2 ✓

What verification code should you include to make sure this operation worked correctly?

- Return correct value.
- Remove this value from the list
- Update size of the list

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class MyLinkedListTester {
private MyLinkedList<Integer> list1;
@Before
public void setUp() throws Exception {
        list1 = new MyLinkedList<Integer>();
        list1.add(0, 42);
        list1.add(0, 21);
        list1.add(0, 65);
}
@Test
public void testRemove(){
        int a = list1.remove(0);
        assertEquals("Remove: check a is correct ", 65, a);
        assertEquals("Remove: check element 0 is correct ", (Integer)21, list1.get(0));
        assertEquals("Remove: check size is correct ", 2, list1.size());
}
```

What about this bug?

Check that list1.get(0).prev is equal to list1.head

breaks black box testing, but you can include if you want

Summary

```
import java.util.*;

ArrayList<E> arrL = new ArrayList<E>();

LinkedList<E> linkL = new LinkedList<E>();
```

```
[Item1, Item2]
[First Item, Item1, Item2, Last Item]
First Item
Changed first item
[Item1, Item2]
[Newly added item, Item1]
```

```
import java.util.*;
public class LinkedListExample {
        public static void main(String args[]) {
        LinkedList<String> linkedlist =
                        new LinkedList<String>();
        linkedlist.add("Item1");
        linkedlist.add("Item2");
        System.out.println(linkedlist);
        linkedlist.addFirst("First Item");
        linkedlist.addLast("Last Item");
        System.out.println(linkedlist);
        Object firstvar = linkedlist.get(0);
        System.out.println(firstvar);
        linkedlist.set(0, "Changed first item");
        Object firstvar2 = linkedlist.get(0);
        System.out.println(firstvar2);
        linkedlist.removeFirst();
        linkedlist.removeLast();
        System.out.println(linkedlist);
        linkedlist.add(0, "Newly added item");
        linkedlist.remove(2);
        System.out.println(linkedlist);
```

ArrayList	LinkedList
ArrayList internally uses **dynamic array** to store the elements.	LinkedList internally uses **doubly linked list** to store the elements.
Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.
ArrayList has **less memory overhead**, and each index only holds actual data.	LinkedList has **more memory overhead**, and each node holds both data and references

Additional Resources

- Linked Lists
  - https://docs.oracle.com/javase/10/docs/api/java/util/LinkedList.html -- Oracle's official API document
  - My implementation code with unit tests is uploaded on blackboard
- Writing JUnit tests
  - http://www.tutorialspoint.com/junit/junit_test_framework.htm -- explains fixtures, test suites, test runners, JUnit classes
- Exceptions and Exception Handling
  - https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html - Oracle's tutorial on Exceptions.