Data Structures and Algorithms I

List ADT & Linked Lists

Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Mr. Aaron Tan Tuck Choy, and Dr. Low Kok Lim for kindly sharing these materials.

Policies for students

- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

Recording of modifications

- Course website address is changed to http://sakai.it.tdt.edu.vn
- Slides “Practice Exercises” are eliminated.
- Course codes cs1010, cs1020, cs2010 are placed by 501042, 501043, 502043 respectively.

Objectives

1. Able to define a List ADT
2. Able to implement a List ADT with array
3. Able to implement a List ADT with linked list
4. Able to use Java API LinkedList class

References

Book

- **List ADT:** Chapter 4, pages 227 to 233
- An array-based implementation: Chapter 4, pages 250 to 257
- **Linked Lists:** Chapter 5, pages 265 to 325

IT-TDT Sakai → 501043 website → Lessons

- http://sakai.it.tdt.edu.vn

Programs used in this lecture

- For Array implementation of List:
  - ListInterface.java
  - ListUsingArray.java, TestListUsingArray.java
- For Linked List implementation of List:
  - ListNode.java
  - ListInterface.java (same ListInterface.java as in array implementation)
  - BasicLinkedList.java, TestBasicLinkedList1.java, TestBasicLinkedList2.java
  - EnhancedListInterface.java
  - EnhancedLinkedList.java, TestEnhancedLinkedList.java
  - TailedLinkedList.java, TestTailedLinkedList.java

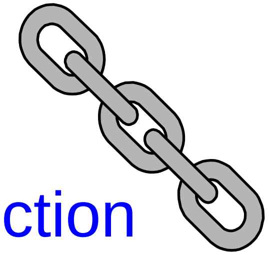
Outline

1. Use of a List (Motivation)
   - List ADT
2. List ADT Implementation via Array
   - Adding and removing elements in an array
   - Time and space efficiency
3. List ADT Implementation via Linked Lists
   - Linked list approach
   - ListNode class: forming a linked list with ListNode
   - BasicLinkedList
4. More Linked Lists
   - EnhancedLinkedList, TailedLinkedList
5. Other Variants
   - CircularLinkedList, DoublyLinkedList
6. Java API: LinkedList class
7. Summary

1 Use of a List

Motivation

Motivation



- **List** is one of the most basic types of data collection
  - For example, list of groceries, list of modules, list of friends, etc.
  - In general, we keep items of the same type (class) in one list
- Typical Operations on a data collection
  - Add data
  - Remove data
  - Query data
  - The details of the operations vary from application to application. The overall theme is the management of data

ADT of a List (1/3)

- A list ADT is a dynamic linear data structure
  - A collection of data items, accessible one after another starting from the beginning (head) of the list

You will learn non-linear data structures such as trees and graphs in 502043.

- Examples of List ADT operations:
  - Create an empty list
  - Determine whether a list is empty
  - Determine number of items in the list
  - Add an item at a given position
  - Remove an item at a position
  - Remove all items
  - Read an item from the list at a position
- The next slide on the basic list interface does not have all the above operations… we will slowly build up these operations in list beyond the basic list.

ADT of a List (2/3)

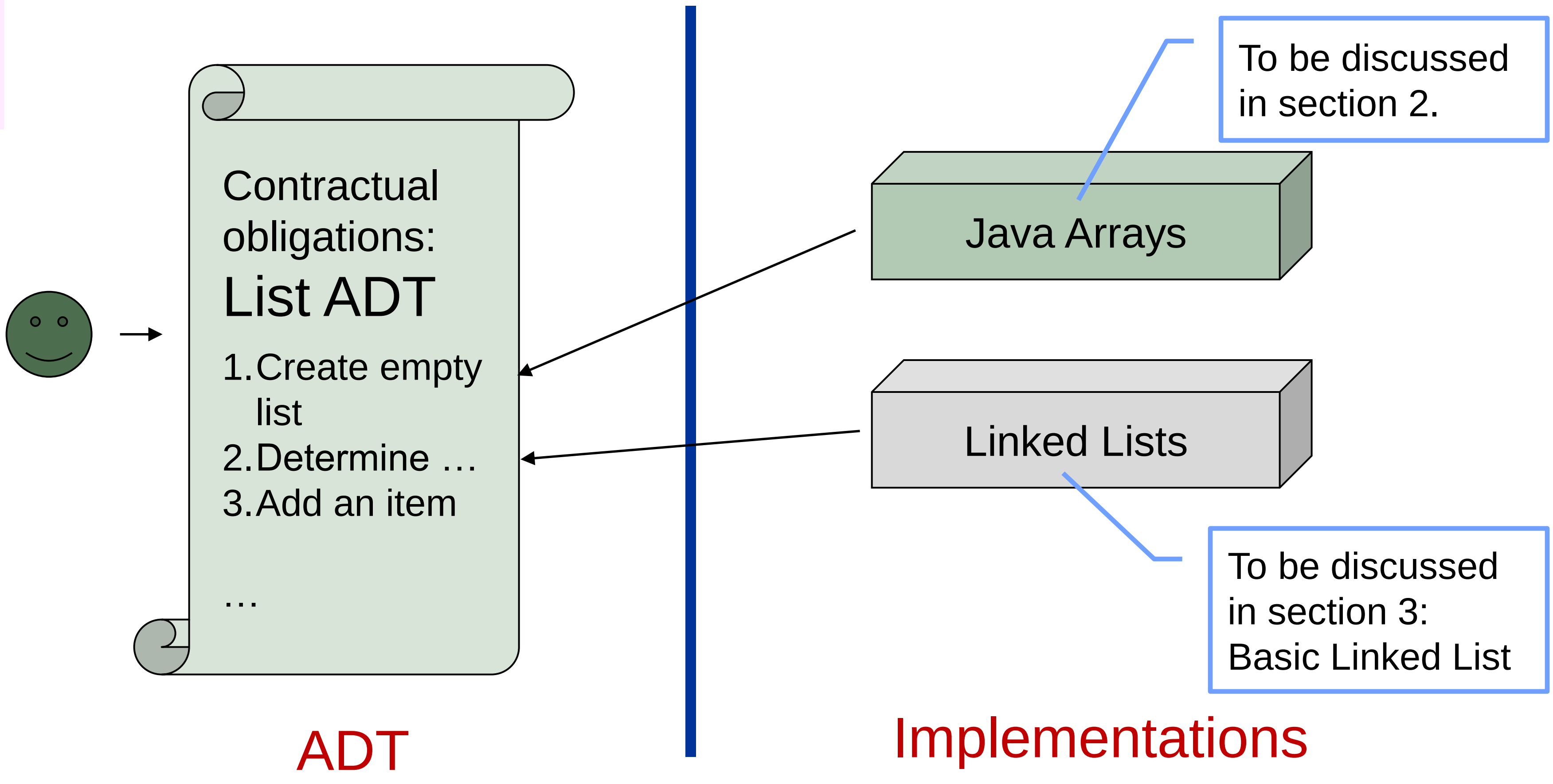
ListInterface.java

```
import java.util.*;

public interface ListInterface <E> {
  public boolean isEmpty();
  public int     size();
  public E       getFirst() throws NoSuchElementException;
  public boolean contains(E item);
  public void    addFirst(E item);
  public E       removeFirst()
                      throws NoSuchElementException;
  public void    print();
}
```

- The **ListInterface** above defines the operations (methods) we would like to have in a List ADT
- The operations shown here are just a small sample. An actual List ADT usually contains more operations.

ADT of a List (3/3)

- We will examine 2 implementations of list ADT, both using the **ListInterface** shown in the previous slide

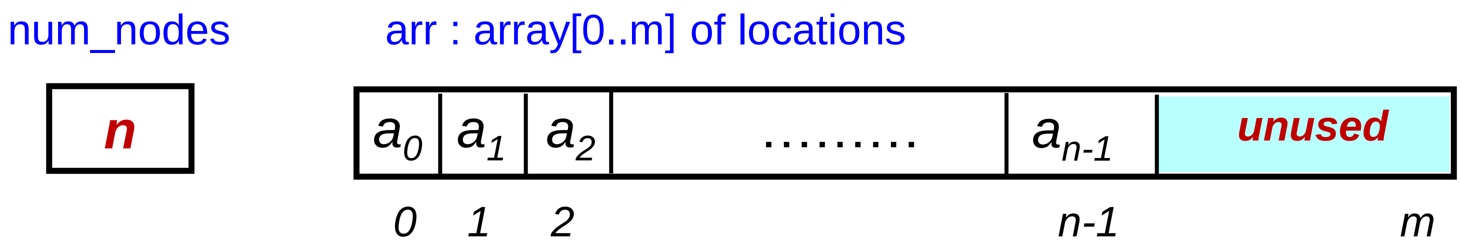


2 List Implementation via Array

Fixed-size list

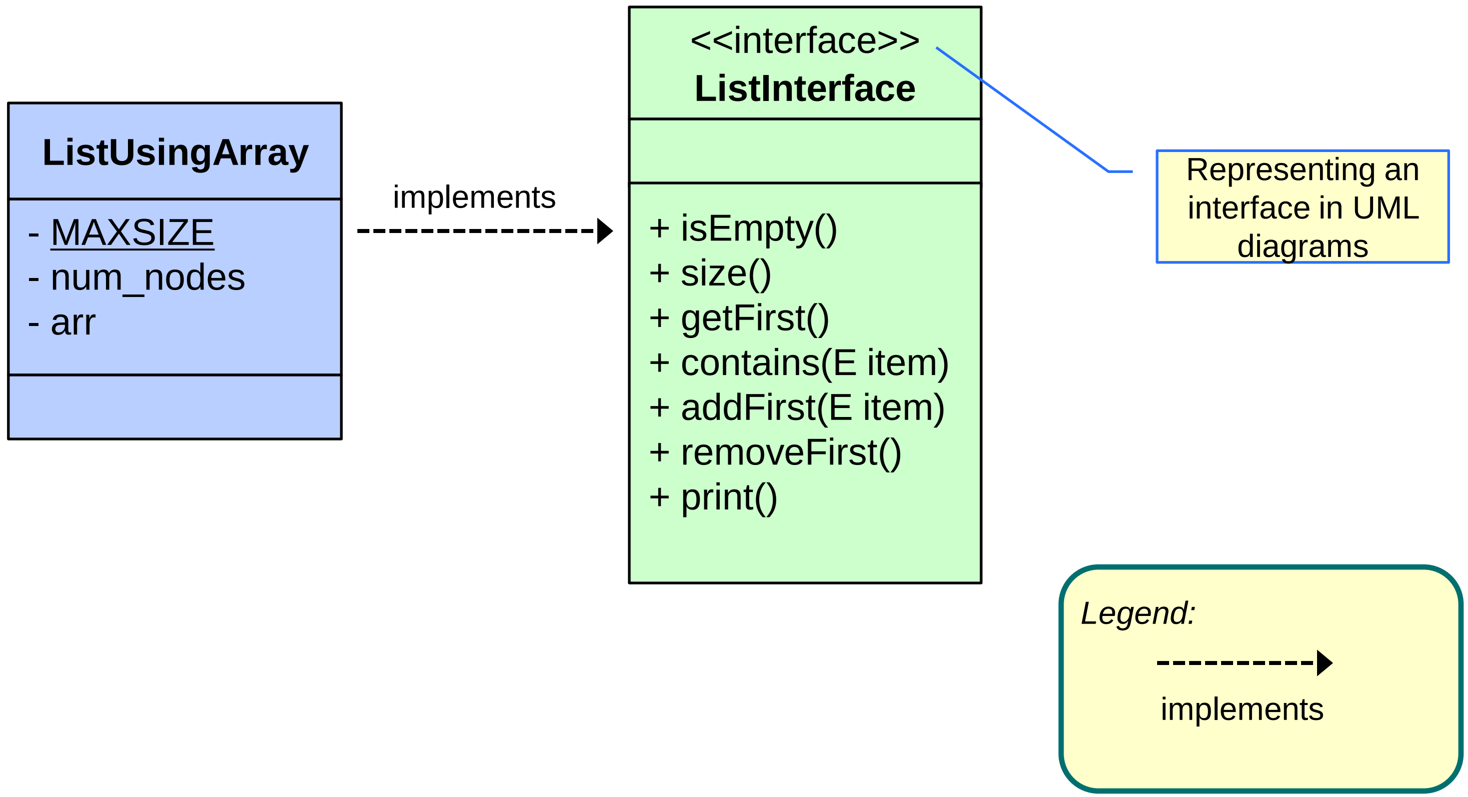
2. List Implementation: Array (1/9)

- This is a straight-forward approach
  - Use Java array of a sequence of *n* elements



2. List Implementation: Array (2/9)

- We now create a class ListUsingArray as an implementation of the interface ListInterface (a user-defined interface, as defined in slide 9)



2. List Implementation: Array (3/9)

ListUsingArray.java

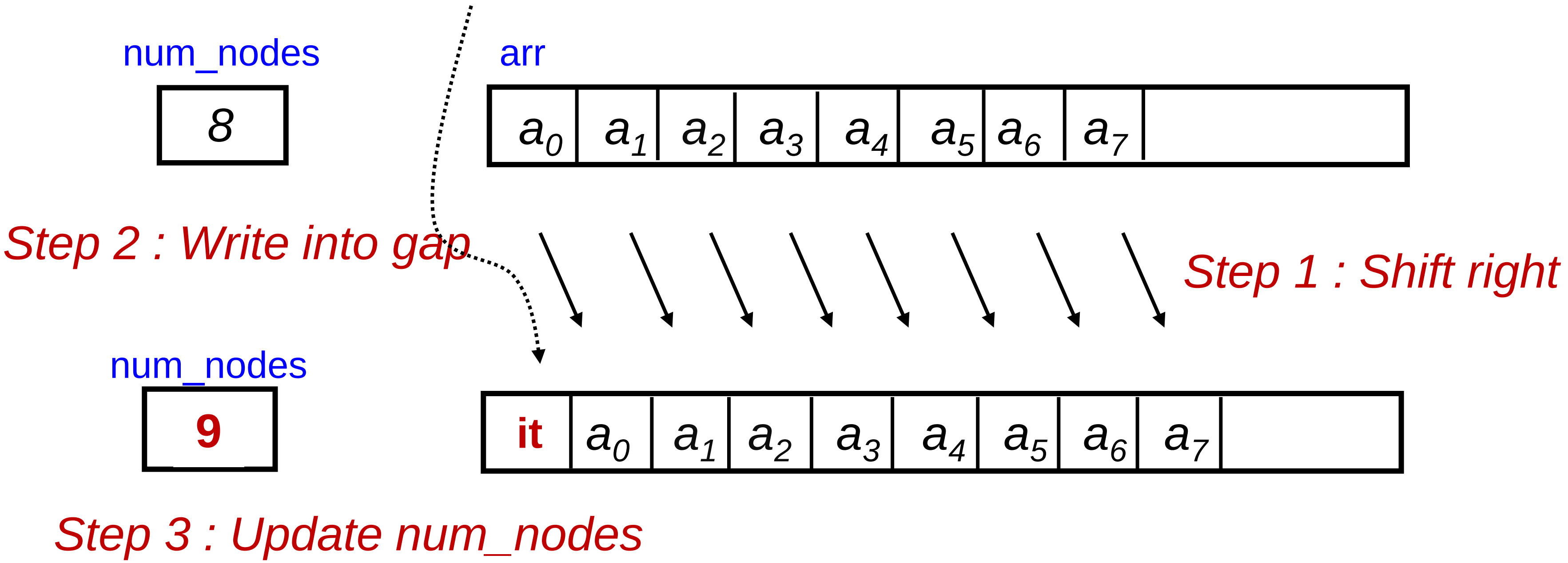
```
import java.util.*;

class ListUsingArray <E> implements ListInterface <E> {
  private static final int MAXSIZE = 1000;
  private int num_nodes = 0;
  private E[] arr = (E[]) new Object[MAXSIZE];

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

  public E getFirst() throws NoSuchElementException {
    if (num_nodes == 0)
      throw new NoSuchElementException("can't get from an empty list");
    else return arr[0];
  }

  public boolean contains(E item) {
    for (int i = 0; i < num_nodes; i++)
      if (arr[i].equals(item)) return true;

    return false;
  }
```

Code continued in slide 17

2. List Implementation: Array (4/9)

- For insertion into first position, need to shift "right" (starting from the last element) to create room

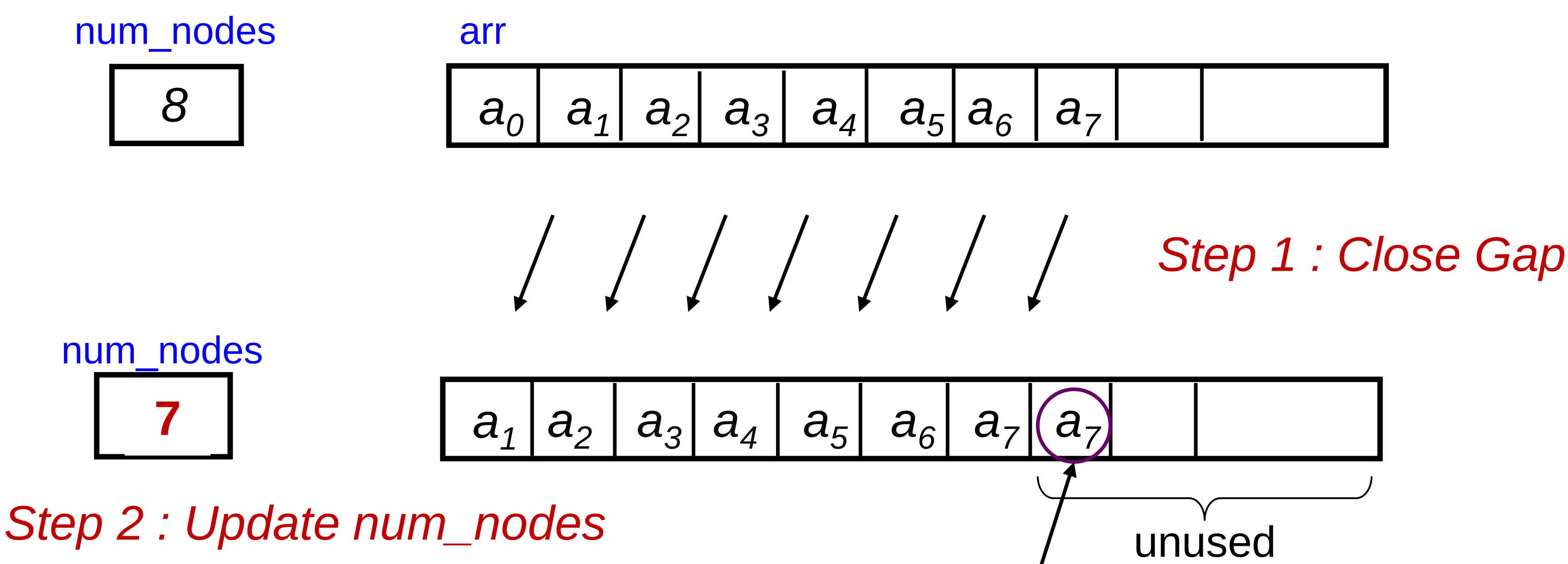
Example: addFirst("it")



2. List Implementation: Array (5/9)

- For deletion of first element, need to shift “left” (starting from the first element) to close gap

Example: removeFirst()



Need to maintain *num_nodes* so that program would not access beyond the valid data.

2. List Implementation: Array (6/9)

```
public void addFirst(E item) throws IndexOutOfBoundsException {
  if (num_nodes == MAXSIZE)
    throw new IndexOutOfBoundsException("insufficient space for add");
  for (int i = num_nodes-1; i >= 0; i--)
    arr[i+1] = arr[i]; // to shift elements to the right

  arr[0] = item;
  num_nodes++; // update num_nodes
}

public E removeFirst() throws NoSuchElementException {
  if (num_nodes == 0)
    throw new NoSuchElementException("can't remove from an empty list");
  else {
    E tmp = arr[0];
    for (int i = 0; i<num_nodes-1; i++)
      arr[i] = arr[i+1]; // to shift elements to the left
    num_nodes--; // update num_nodes
    return tmp;
  }
}
```

`print()` method not shown here. Refer to program.

ListUsingArray.java

2. Testing Array Implementation (7/9)

```
import java.util.*;

public class TestListUsingArray {
  public static void main(String [] args)
                              throws NoSuchElementException {
    ListUsingArray <String> list = new ListUsingArray <String>();
    list.addFirst("aaa");
    list.addFirst("bbb");
    list.addFirst("ccc");
    list.print();

    System.out.println("Testing removal");
    list.removeFirst();
    list.print();

    if (list.contains("aaa"))
      list.addFirst("xxxx");
    list.print();
  }
}
```

TestListUsingArray.java

2. Analysis of Array Impl^{n} of List (8/9)

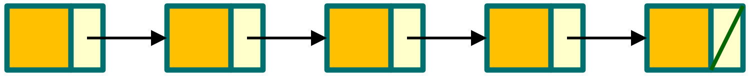
- Question: Time Efficiency?
  - Retrieval: getFirst()
    - Always fast with 1 read operation
  - Insertion: addFirst(E item)
    - Shifting of all *n* items – bad!
  - Insertion: add(int index, E item)
    - Inserting into the specified position (not shown in ListUsingArray.java)
      - Best case: No shifting of items (add to the last place)
      - Worst case: Shifting of all items (add to the first place)
  - Deletion: removeFirst(E item)
    - Shifting of all *n* items – bad!
  - Deletion: remove(int index)
    - Delete the item at the specified position (not shown in ListUsingArray.java)
      - Best case: No shifting of items (delete the last item)
      - Worst case: Shifting of all items (delete the first item)

2. Analysis of Array Impl^{n} of List (9/9)

- Question: What is the Space Efficiency?
  - Size of array collection limited by MAXSIZE
  - Problems
    - We don't always know the maximum size ahead of time
    - If MAXSIZE is too liberal, unused space is wasted
    - If MAXSIZE is too conservative, easy to run out of space
- Idea: make MAXSIZE a variable, and create/copy to a larger array whenever the array runs out of space
  - No more limits on size
  - But copying overhead is still a problem
- When to use such a list?
  - For a fixed-size list, an array is good enough!
  - For a variable-size list, where dynamic operations such as insertion/deletion are common, an array is a poor choice; better alternative – **Linked List**

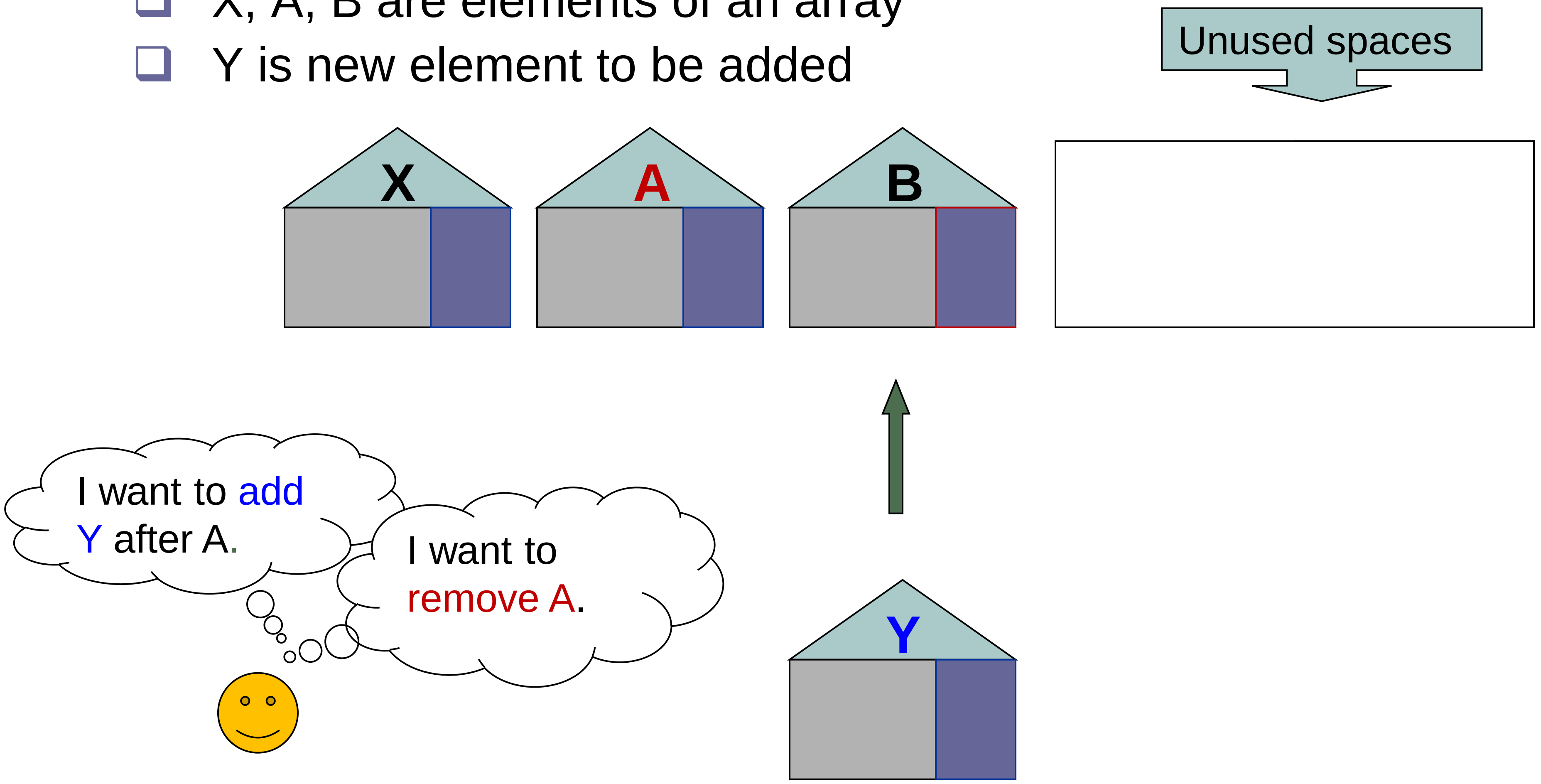
3 List Implementation via Linked List

Variable-size list



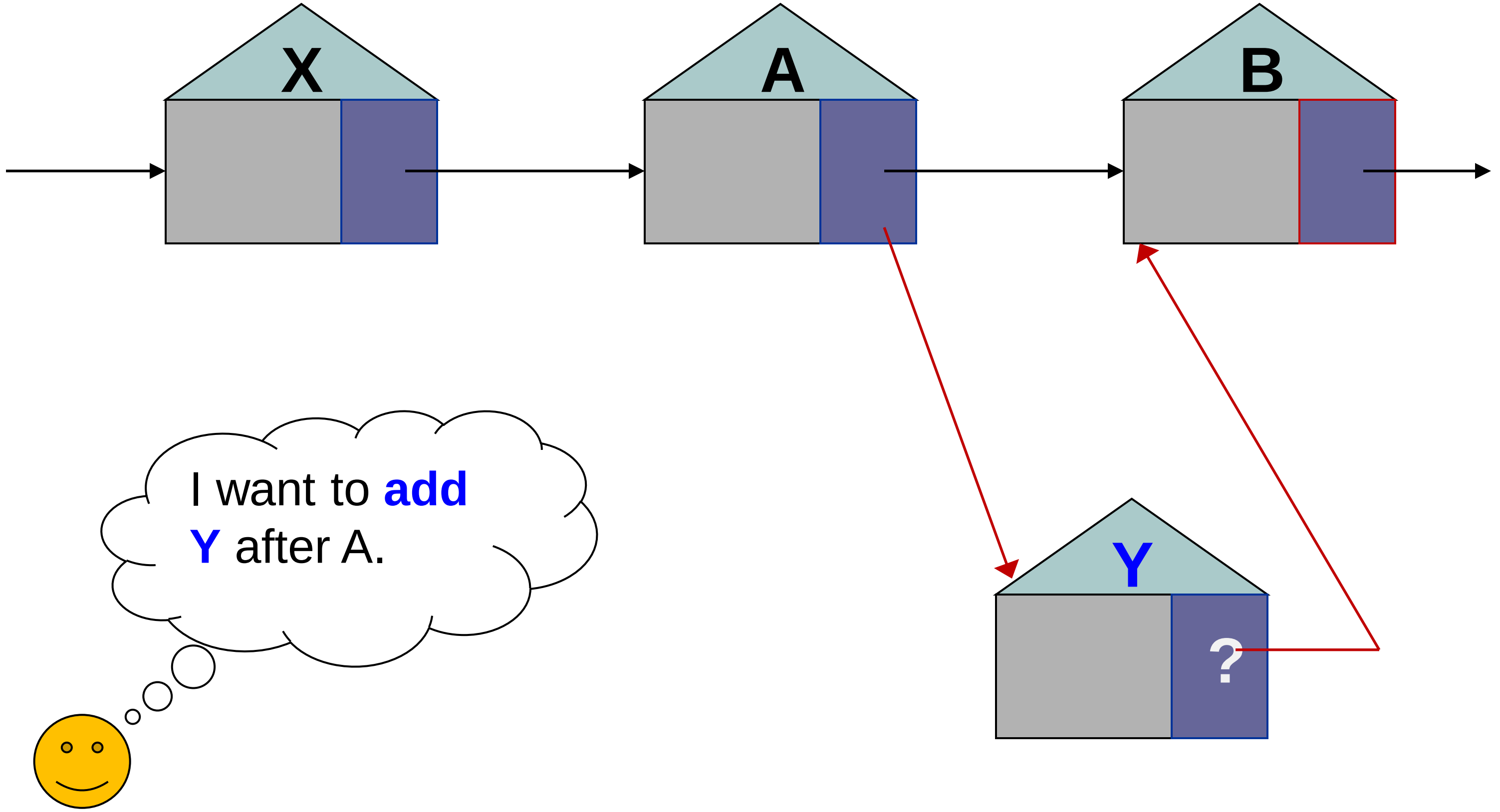
3.1 List Implementation: Linked List (1/3)

- Recap when using an array...
  - X, A, B are elements of an array
  - Y is new element to be added



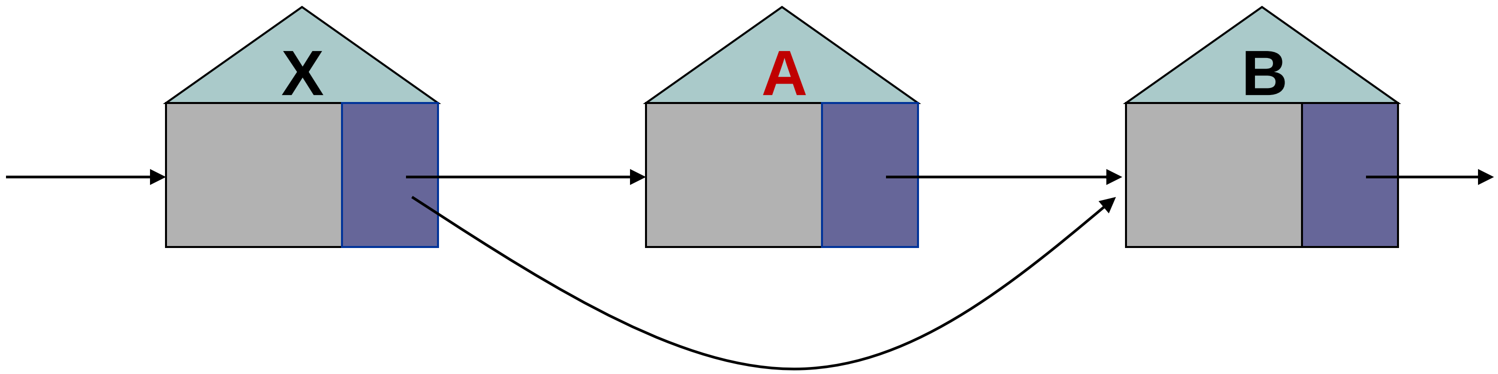
3.1 List Implementation: Linked List (2/3)

- Now, we see the (add) action with linked list…
  - X, A, B are nodes of a linked list
  - Y is new node to be added



3.1 List Implementation: Linked List (3/3)

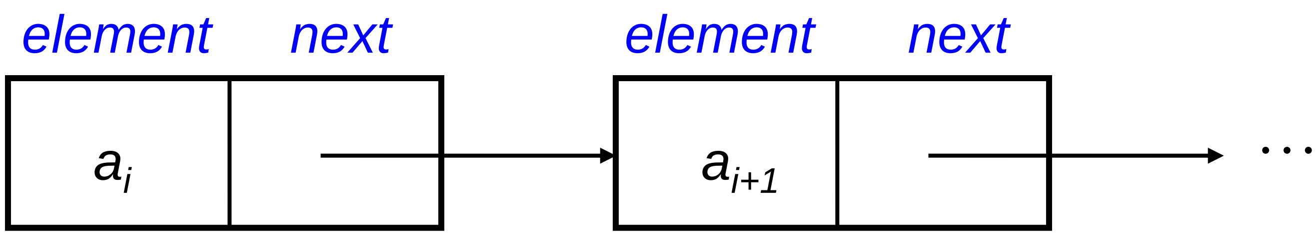
- Now, we see the (remove) action with linked list…



Node A becomes a *garbage*. To be removed during garbage collection.

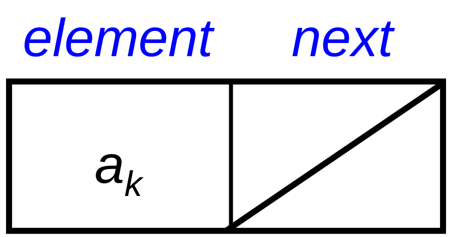
3.2 Linked List Approach (1/4)

- Idea
  - Each element in the list is stored in a *node*, which also contains a next pointer
  - Allow elements in the list to occupy *non-contiguous* memory
  - Order the nodes by associating each with its neighbour(s)



This is one node of the collection…

… and this one comes after it in the collection (most likely not occupying contiguous memory that is next to the previous node).

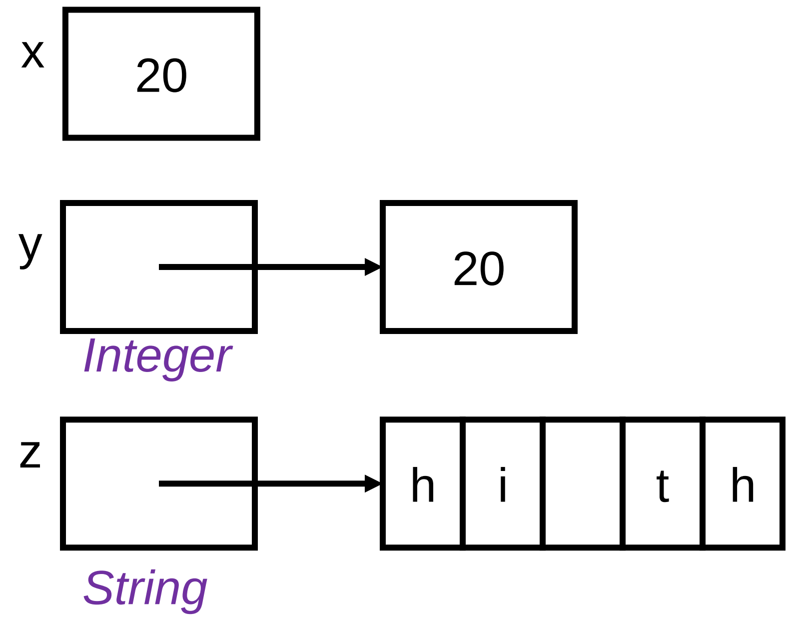


Next pointer of this node is “null”, i.e. it has no next neighbour.

3.2 Linked List Approach (2/4)

- Recap: Object References (1/2)
  - Note the difference between primitive data types and reference data types

```
int x = 20;

Integer y = new Integer(20);

String z = new String("hi th");
```

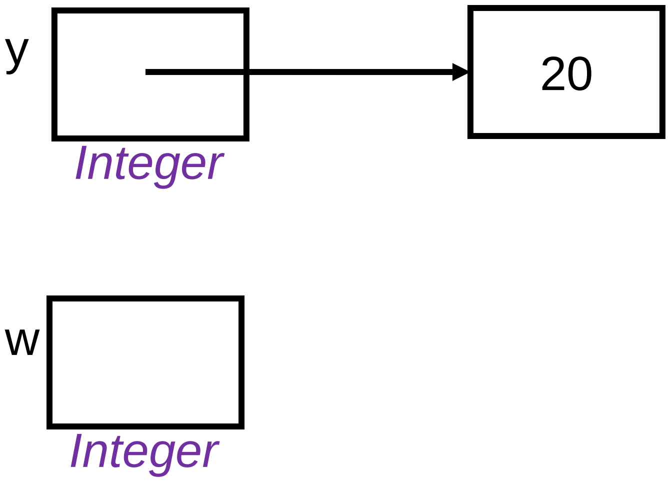


- An instance (object) of a class only comes into existence (constructed) when the new operator is applied
- A reference variable only contains a reference or pointer to an object.

3.2 Linked List Approach (3/4)

- Recap: Object References (2/2)
  - Look at it in more details:

```
Integer y = new Integer(20);
Integer w;
w =  new Integer(20);
if (w == y)
  System.out.println("1. w == y");

w = y;
if (w == y)
     System.out.println("2. w == y");
```



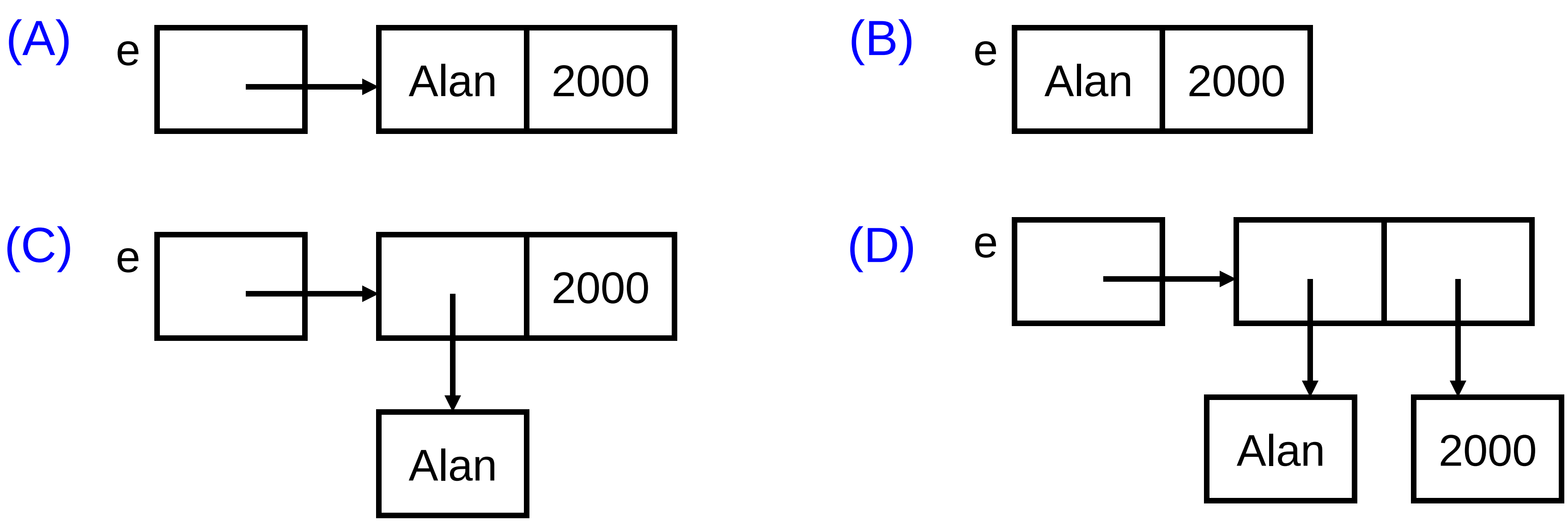
- Output:

3.2 Linked List Approach (4/4)

- Quiz: Which is the right representation of e?

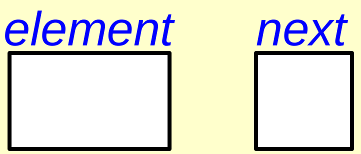
```
class Employee {
    private String name;
    private int salary;
    // etc.
}
```

```
Employee e = new Employee("Alan", 2000);
```



3.3 ListNode (using generic)

ListNode.java

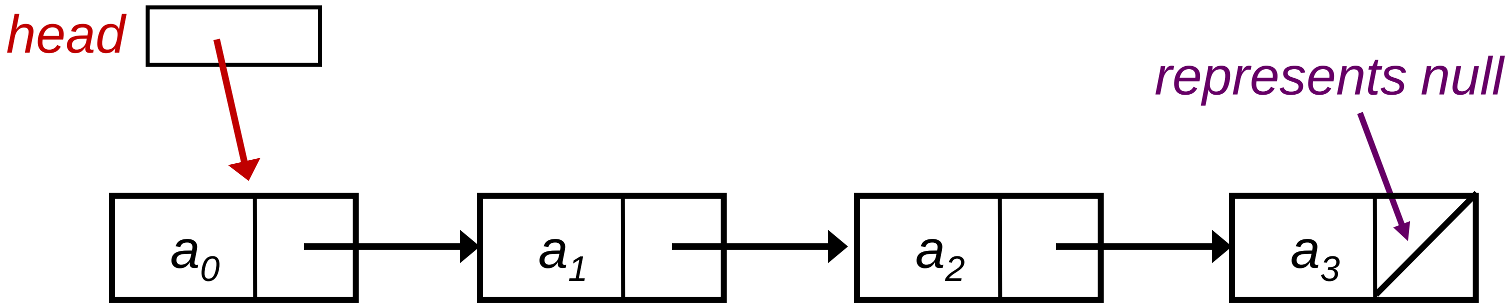



```java
class ListNode <E> {
  /* data attributes */
  private E element;
  private ListNode <E> next;

  /* constructors */
  public ListNode(E item) { this(item, null); }

  public ListNode(E item, ListNode <E> n) {
    element = item;
    next = n;
  }

  /* get the next ListNode */
  public ListNode <E> getNext() { return next; }

  /* get the element of the ListNode */
  public E getElement() { return element; }

  /* set the next reference */
  public void setNext(ListNode <E> n) { next = n };
}
```

Mark this slide – You may need to refer to it later when we study the different variants of linked list.

3.4 Forming a Linked List (1/3)

- For a sequence of 4 items < a_0, a_1, a_2, a_3 >



We need a *head* to indicate where the first node is.
From the *head* we can get to the rest.

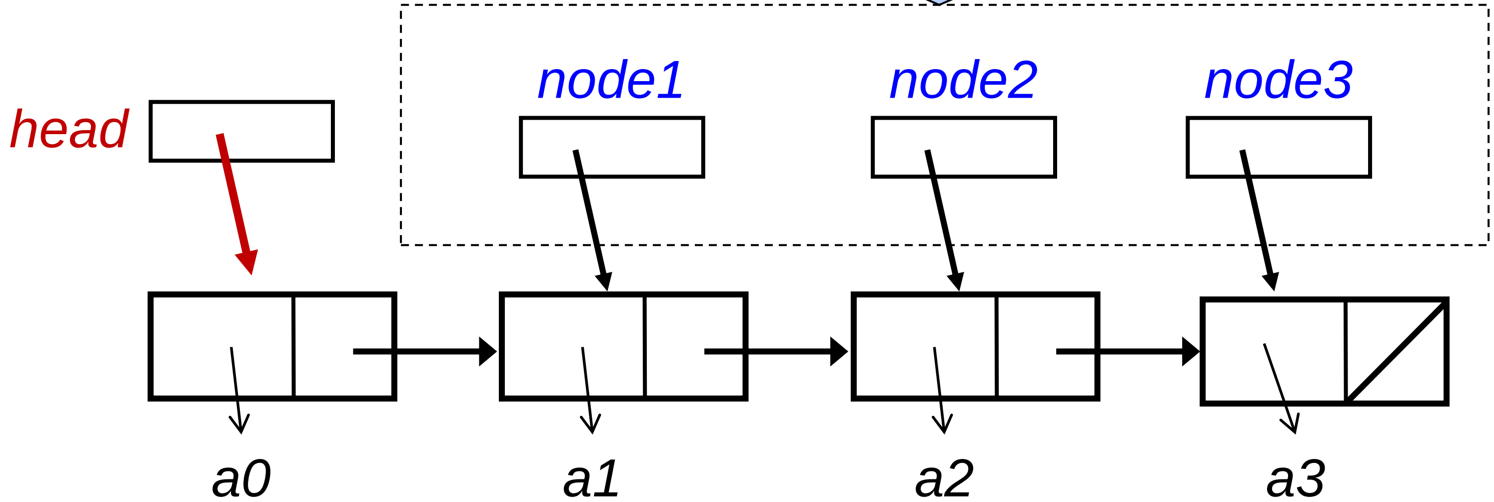
3.4 Forming a Linked List (2/3)

- For a sequence of 4 items $< a_0, a_1, a_2, a_3 >$

```
ListNode <String> node3 = new ListNode <String>("a3", null);
ListNode <String> node2 = new ListNode <String>("a2", node3);
ListNode <String> node1 = new ListNode <String>("a1", node2);
ListNode <String> head  = new ListNode <String>("a0", node1);
```

Can the code be rewritten without using these object references node1, node2, node3?

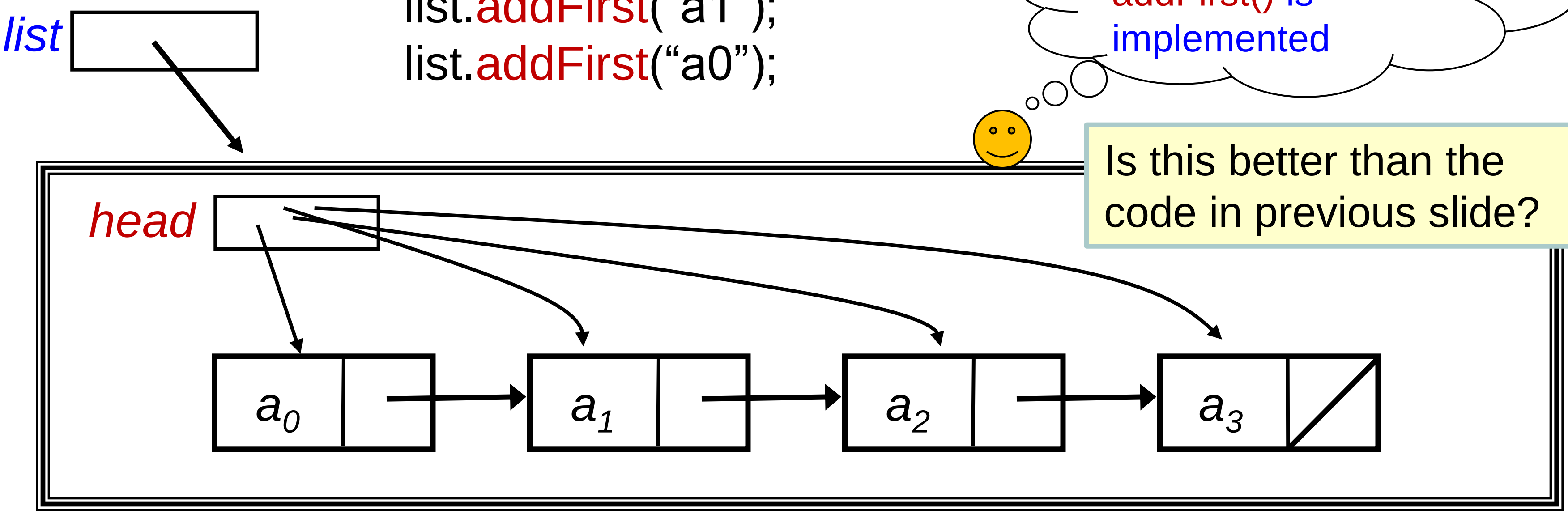
No longer needed after list is built.



3.4 Forming a Linked List (3/3)

- Alternatively we can form the linked list as follows:
  - For a sequence of 4 items < a_0, a_1, a_2, a_3 >, we can build as follows:

```
LinkedList <String> list = new LinkedList <String>();
list.addFirst("a3");
list.addFirst("a2");
list.addFirst("a1");
list.addFirst("a0");
```



I don't care how addFirst() is implemented

Is this better than the code in previous slide?

3.5 Basic Linked List (1/7)

- Using ListNode to define BasicLinkedList

BasicLinkedList.java

```
import java.util.*;
class BasicLinkedList <E> implements ListInterface <E> {
  private ListNode <E> head = null;
  private int num_nodes = 0;

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

  public int size() { return num_nodes; }

  public E getFirst() throws NoSuchElementException {
    if (head == null)
      throw new NoSuchElementException("can't get from an empty list");
    else return head.getElement();
  }

  public boolean contains(E item) {
    for (ListNode <E> n = head; n != null; n = n.getNext())
      if (n.getElement().equals(item)) return true;
    return false;
  }
```

getElement() and getNext() are methods in ListNode class (slide 29)

3.5 Basic Linked List (2/7)

- The adding and removal of first element

BasicLinkedList.java

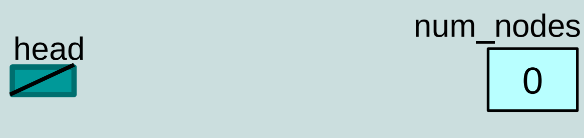
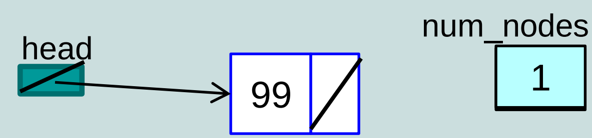
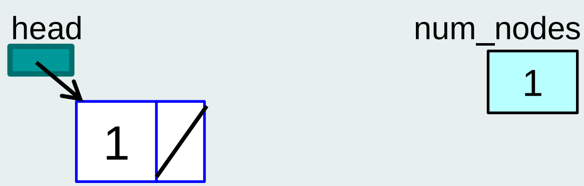
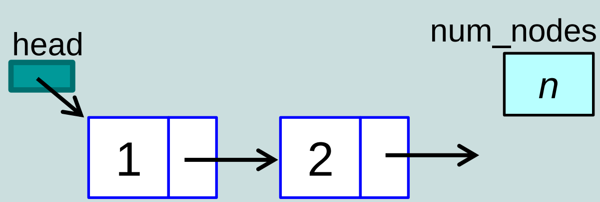
```
public void addFirst(E item) {
  //head = new ListNode <E> (item, head);
  ListNode p = new ListNode (item);
  p.setNext(head);
  head = p;

  num_nodes++;
}

public E removeFirst() throws NoSuchElementException {
  ListNode <E> ln;
  if (head == null)
    throw new NoSuchElementException("can't r
  else {
    ln = head;
    head = head.getNext();
    num_nodes--;
    return ln.getElement();
  }
}
```

getElement() and getNext() are methods in ListNode class (slide 29)

3.5 Basic Linked List (3/7)

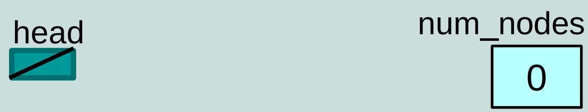
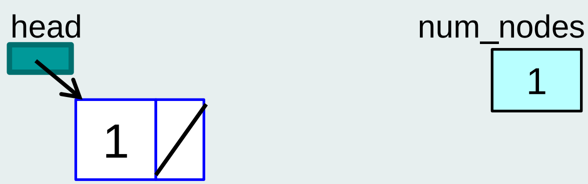
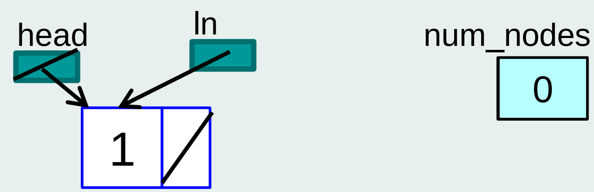
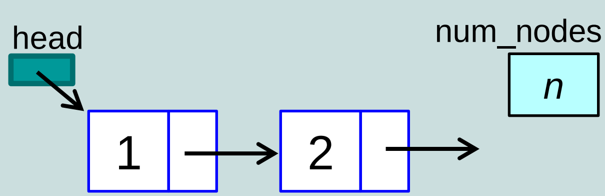
```
public void addFirst(E item) {
   head = new ListNode <E> (item, head);
   num_nodes++;
}
```

- The addFirst() method

Case	Before: list	After: list.addFirst(99)
0 item		
1 item		
2 or more items		

3.5 Basic Linked List (4/7)

- The removeFirst() method

Case	Before: list	After: list.removeFirst()
0 item		can't remove
1 item		
2 or more items		

```java
public E removeFirst() throws NoSuchElementException {
    ListNode <E> ln;
    if (head == null)
        throw new NoSuchElementException("can't remove");
    else {
        ln = head; head = head.getNext(); num_nodes--;
        return ln.getElement();
    }
}
```

3.5 Basic Linked List (5/7)

- Printing of the linked list

BasicLinkedList.java

```
public void print() throws NoSuchElementException {
   if (head == null)
      throw new NoSuchElementException("Nothing to print...");

   ListNode <E> ln = head;
   System.out.print("List is: " + ln.getElement());
   for (int i=1; i < num_nodes; i++) {
      ln = ln.getNext();
      System.out.print(", " + ln.getElement());
   }
   System.out.println(".");
}
```

3.5 Test Basic Linked List #1 (6/7)

- Example use #1

TestBasicLinkedList1.java

```
import java.util.*;

public class TestBasicLinkedList1 {
  public static void main(String [] args)
                              throws NoSuchElementException {
    BasicLinkedList <String> list = new BasicLinkedList <String>();
    list.addFirst("aaa");
    list.addFirst("bbb");
    list.addFirst("ccc");
    list.print();

    System.out.println("Testing removal");
    list.removeFirst();
    list.print();

    if (list.contains("aaa"))
      list.addFirst("xxxx");
    list.print();
  }
}
```

3.5 Test Basic Linked List #2 (7/7)

- Example use #2

TestBasicLinkedList2.java

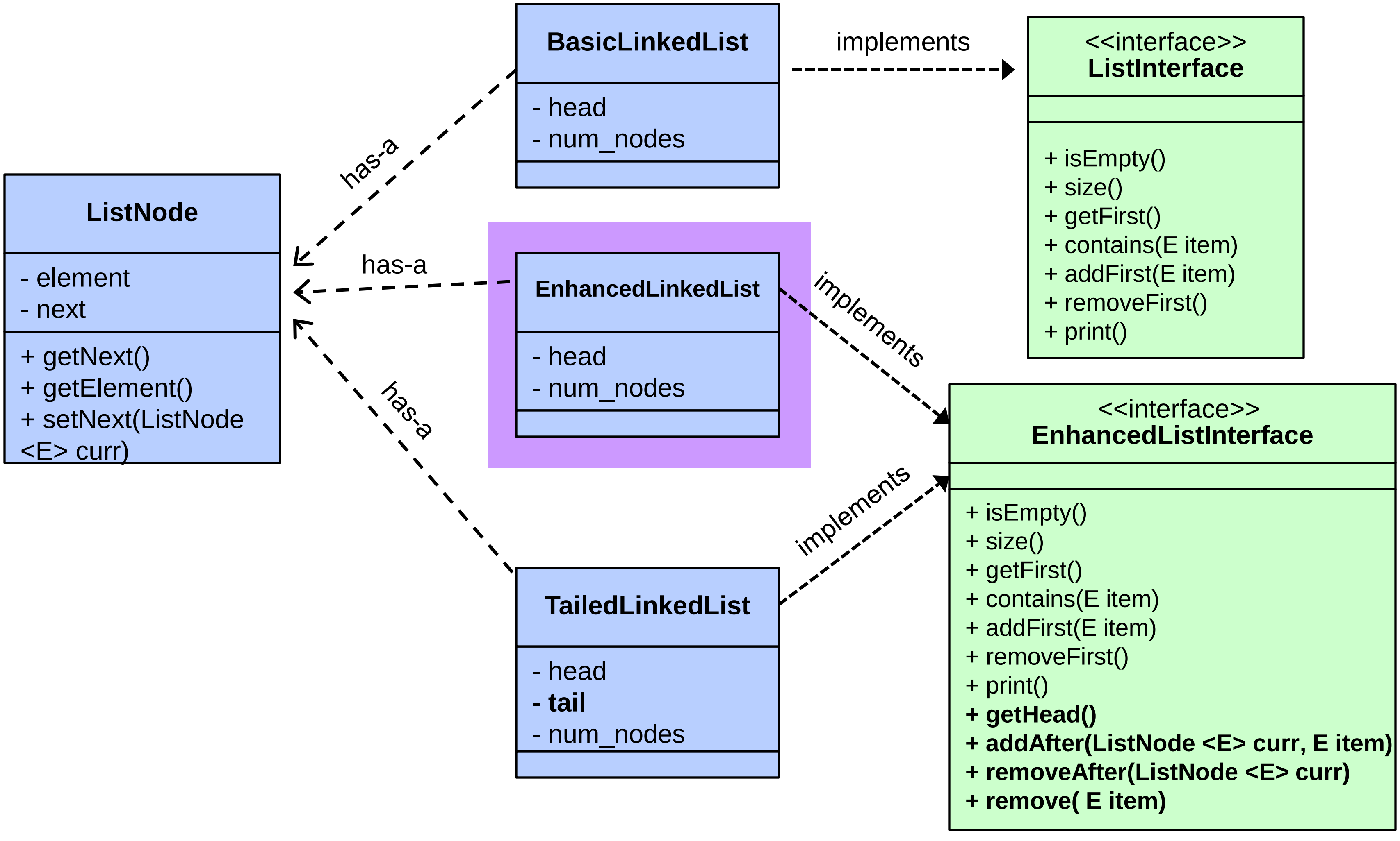
```
import java.util.*;

public class TestBasicLinkedList2 {
  public static void main(String [] args)
                          throws NoSuchElementException {
    BasicLinkedList <Integer> list = new BasicLinkedList <Integer>();

    list.addFirst(34);
    list.addFirst(12);
    list.addFirst(9);
    list.print();

    System.out.println("Testing removal");
    list.removeFirst();
    list.print();
  }
}
```

4 More Linked Lists

Exploring variants of linked list

4. Linked Lists: Variants

OVERVIEW!



4.1 Enhanced Linked List (1/11)

- We explore different implementations of Linked List
  - Basic Linked List, Tailed Linked List, Circular Linked List, Doubly Linked List, etc.
- When nodes are to be inserted to the middle of the linked list, BasicLinkedList (BLL) is not good enough.
- For example, BLL offers only insertion at the front of the list. If the items in the list must always be sorted according to some key values, then we must be able to insert at the right place.
- We will enhance BLL to include some additional methods. We shall call this Enhanced Linked List (ELL).
  - (Note: We could have made ELL a subclass of BLL, but here we will create ELL from scratch instead.)

4.1 Enhanced Linked List (2/11)

- We use a new interface file:

EnhancedListInterface.java

```
import java.util.*;

public interface EnhancedListInterface <E> {

  public boolean isEmpty();
  public int     size();
  public E       getFirst() throws NoSuchElementException;
  public boolean contains(E item);
  public void    addFirst(E item);
  public E       removeFirst() throws NoSuchElementException;
  public void    print();

  public ListNode <E>   getHead();
  public void    addAfter(ListNode <E> current, E item);
  public E       removeAfter(ListNode <E> current)
                    throws NoSuchElementException;
  public E       remove(E item) throws NoSuchElementException;
}
```

New

4.1 Enhanced Linked List (3/11)

EnhancedLinkedList.java

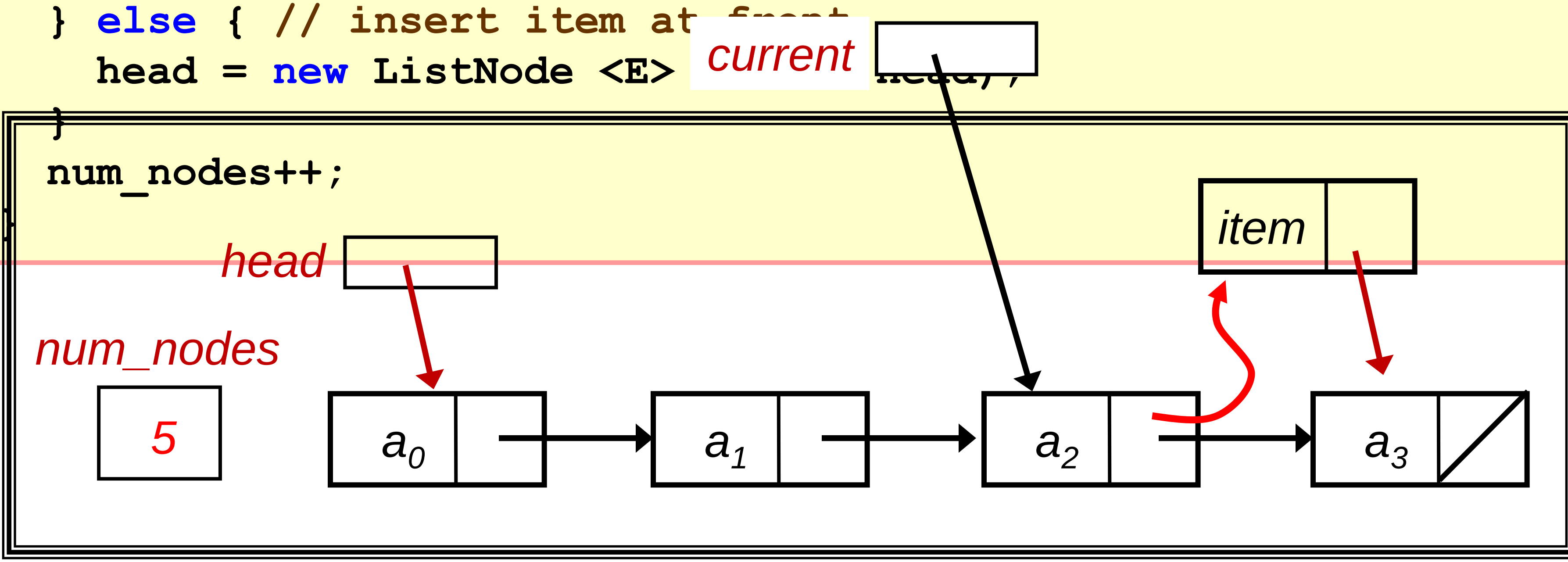
```
import java.util.*;
class EnhancedLinkedList <E>
                implements EnhancedListInterface <E> {
  private ListNode <E> head = null;
  private int num_nodes = 0;

  public boolean isEmpty() { return (num_nodes == 0); }
  public int size() { return num_nodes; }
  public E getFirst() { ... }
  public boolean contains(E item) { ... }
  public void addFirst(E item) { ... }
  public E removeFirst() throws NoSuchElementException { ... };
  public void print() throws NoSuchElementException { ... };

  public ListNode <E> getHead() { return head; }

  public void addAfter(ListNode <E> current, E item) {
    if (current != null)
      current.setNext(new ListNode <E> (item, current.getNext()));
    else // insert item at front
      head = new ListNode <E> (item, head);
    num_nodes++;
  }
```

Same as in BasicLinkedList.java

To continue on next slide

4.1 Enhanced Linked List (4/11)

```
public void addAfter(ListNode <E> current, E item) {
  if (current != null) {
//  current.setNext(new ListNode <E>(item,current.getNext()));
    ListNode<E> p = new ListNode<E>(item);
    p.setNext(current.getNext());
    current.setNext(p);

  } else { // insert item at front
    head = new ListNode <E>               head);
  }
  num_nodes++;
}
```



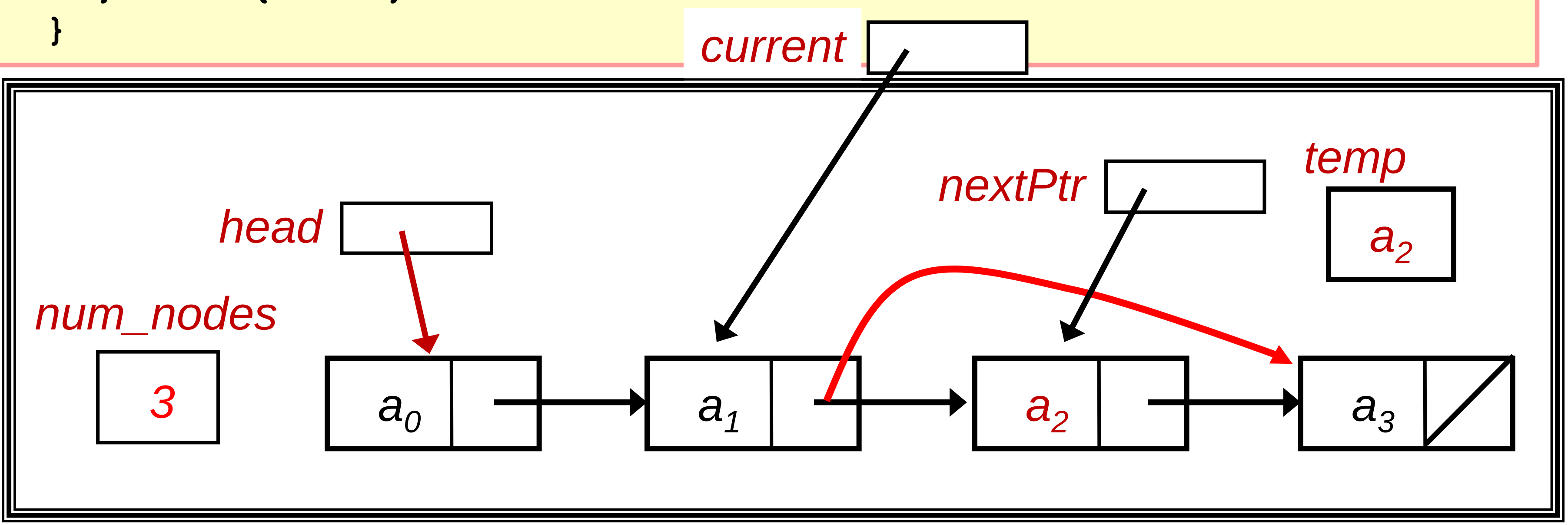
4.1 Enhanced Linked List (5/11)

EnhancedLinkedList.java

```
public E removeAfter(ListNode <E> current)
                     throws NoSuchElementException {
  E temp;
  if (current != null) {
    ListNode <E> nextPtr = current.getNext();
    if (nextPtr != null) {
      temp = nextPtr.getElement();
      current.setNext(nextPtr.getNext());
      num_nodes--;
      return temp;
    } else throw new NoSuchElementException("No next node to remove");
  } else { // if current is null, assume we want to remove head
    if (head != null) {
      temp = head.getElement();
      head = head.getNext();
      num_nodes--;
      return temp;
    } else throw new NoSuchElementException("No next node to remove");
  }
}
```

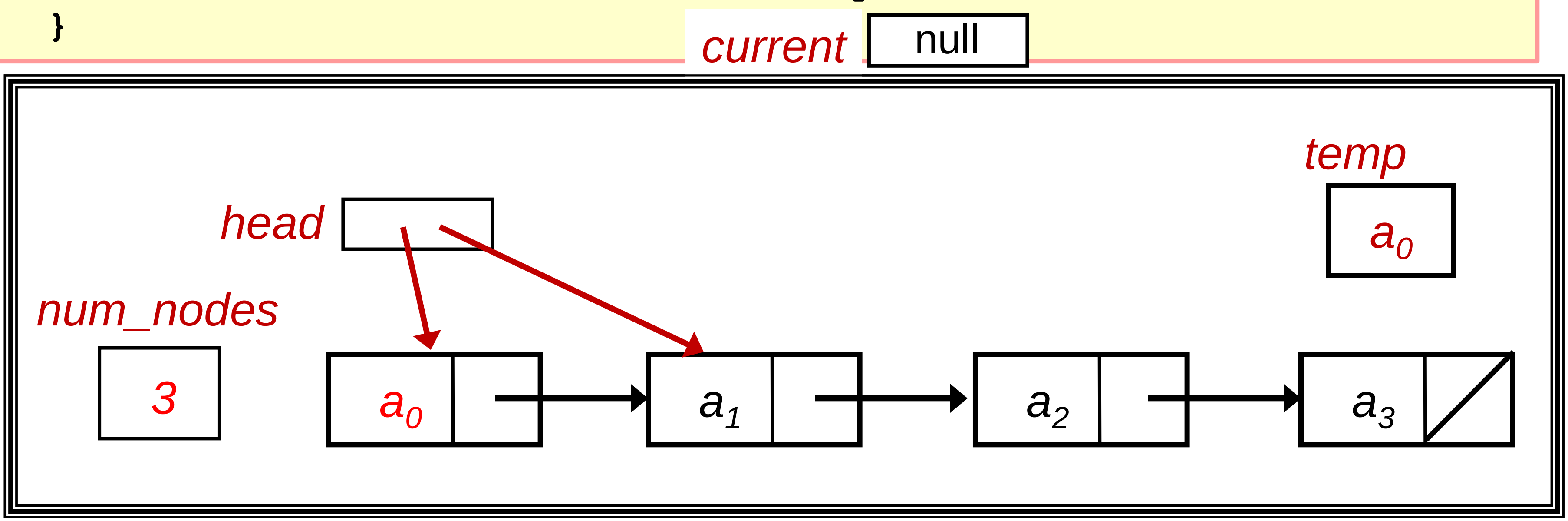
4.1 Enhanced Linked List (6/11)

```
public E removeAfter(ListNode <E> current) throws ... {
   E temp;
   if (current != null) {
      ListNode<E> nextPtr = current.getNext();
      if (nextPtr != null) {
         temp = nextPtr.getElement();
         current.setNext(nextPtr.getNext());
         num_nodes--;
         return temp;
      } else throw new NoSuchElementException("...");
   } else { ... }
}
```



4.1 Enhanced Linked List (7/11)

```
public E removeAfter(ListNode <E> current) throws ... {
  E temp;
  if (current != null) {
    ...
  } else { // if current is null, we want to remove head
    if (head != null) {
      temp = head.getElement();
      head = head.getNext();
      num_nodes--;
      return temp;
  } else throw new NoSuchElementException("...");
}
```



4.1 Enhanced Linked List (8/11)

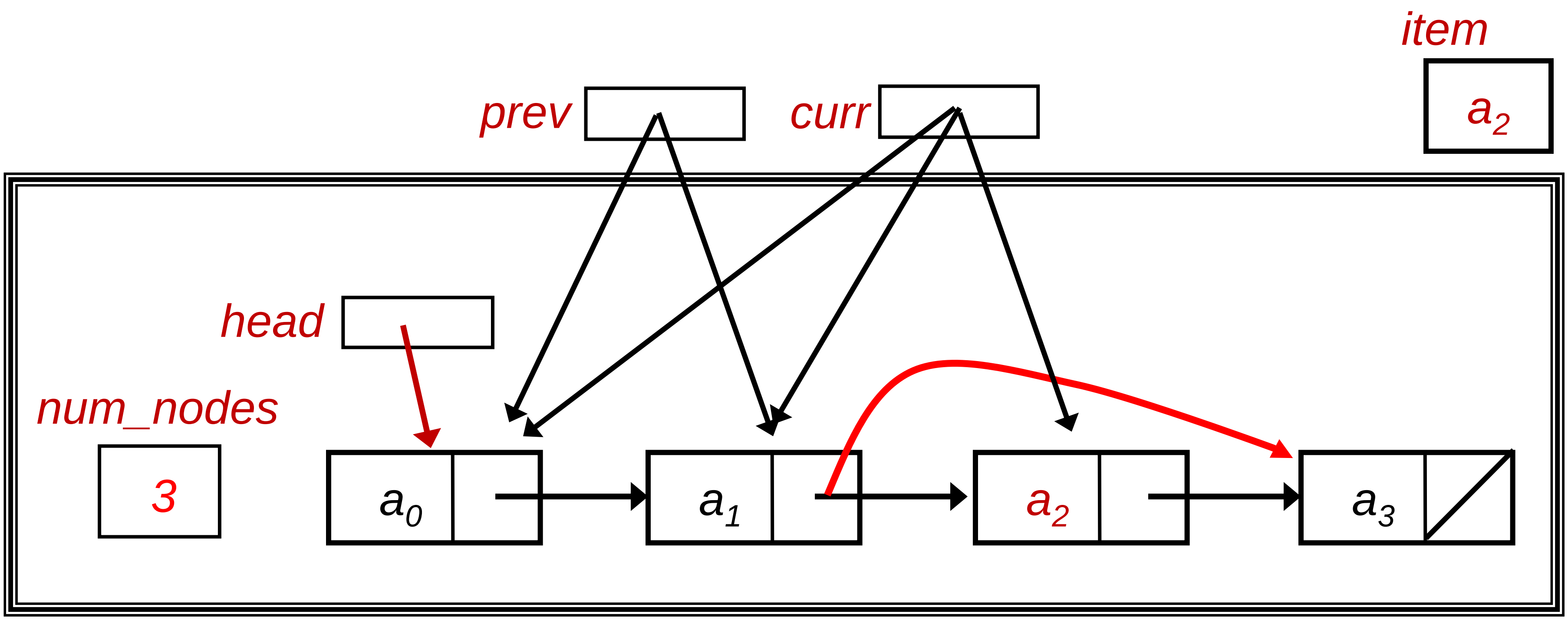
- remove(E item)
  - Search for item in list
  - Re-using removeAfter() method

EnhancedLinkedList.java

```
public E remove(E item)
                  throws NoSuchElementException {
  // Write your code below...
  // Should make use of removeAfter() method.









  }
}
```

4.1 Enhanced Linked List (9/11)

```
public E remove(E item) throws ... {




}
```



4.1 Test Enhanced Linked List (10/11)

TestEnhancedLinkedList.java

```
import java.util.*;

public class TestEnhancedLinkedList {
  public static void main(String [] args) throws NoSuchElementException {

    EnhancedLinkedList <String> list = new EnhancedLinkedList
<String>();
    System.out.println("Part 1");
    list.addFirst("aaa");
    list.addFirst("bbb");
    list.addFirst("ccc");
    list.print();


    System.out.println();
    System.out.println("Part 2");
    ListNode <String> current = list.getHead();
    list.addAfter(current, "xxx");
    list.addAfter(current, "yyy");
    list.print();
```

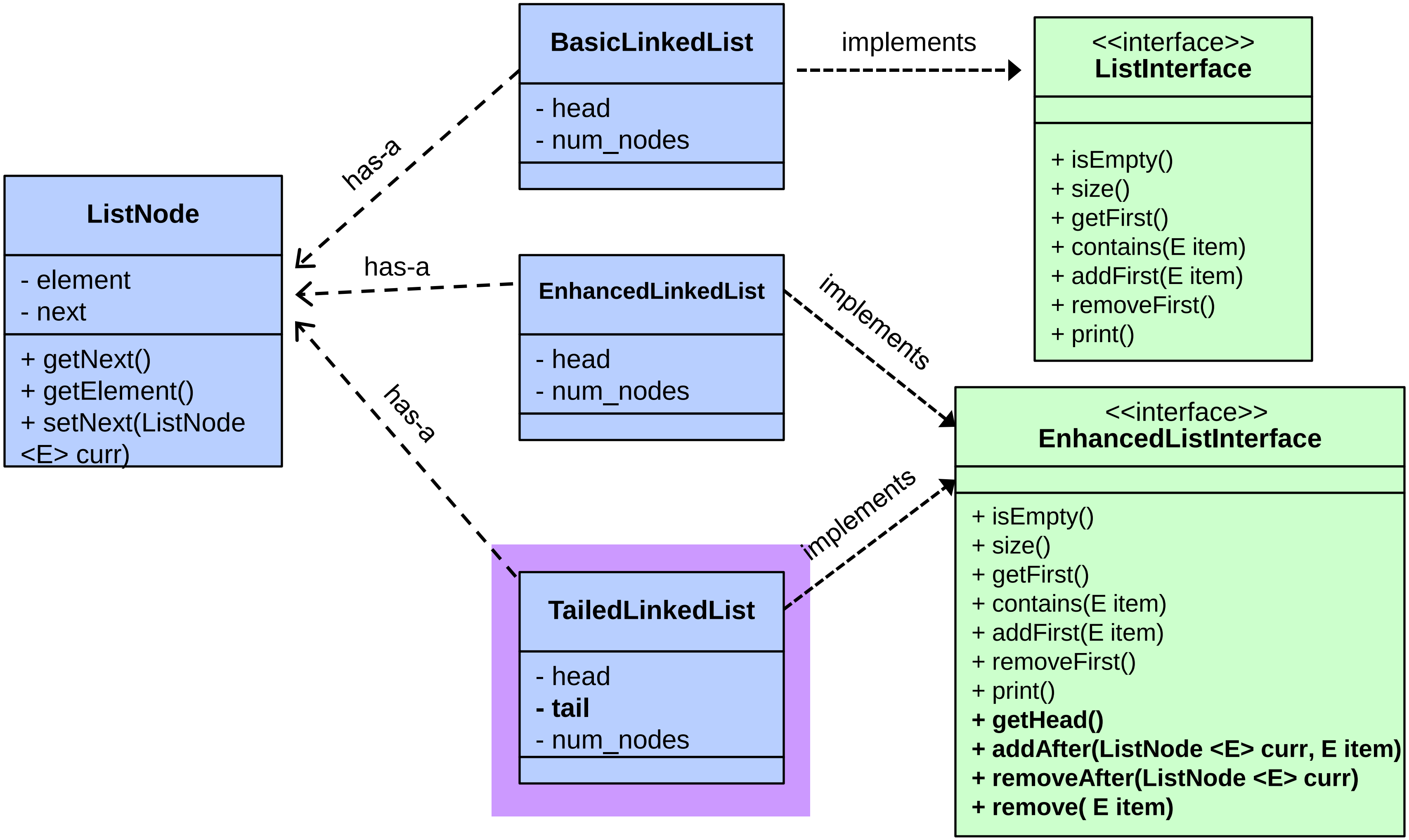
4.1 Test Enhanced Linked List (11/11)

TestEnhancedLinkedList.java

```
// (continue from previous slide)

    System.out.println();
    System.out.println("Part 3");
    current = list.getHead();
    if (current != null) {
      current = current.getNext();
      list.removeAfter(current);
    }
    list.print();

    System.out.println();
    System.out.println("Part 4");
    list.removeAfter(null);
    list.print();
  }
}
```

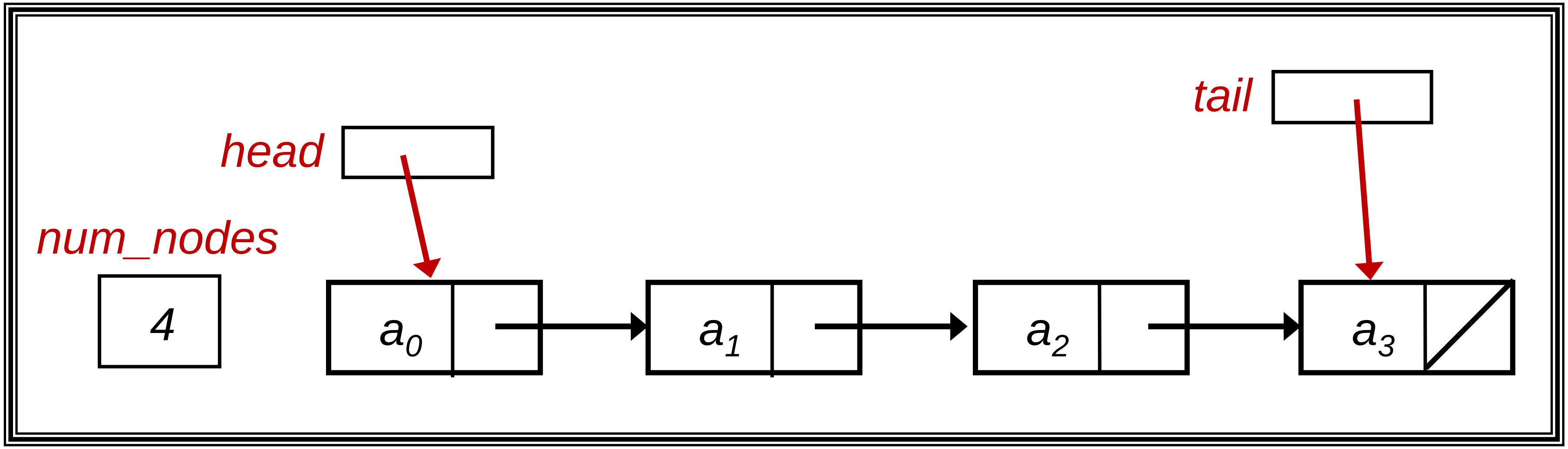
4. Linked Lists: Variants

OVERVIEW!



4.2 Tailed Linked List (1/10)

- We further improve on Enhanced Linked List
  - To address the issue that adding to the end is slow
  - Add an extra data member called tail
  - Extra data member means extra maintenance too – no free lunch!
  - (Note: We could have created this Tailed Linked List as a subclass of Enhanced Linked List, but here we will create it from scratch.)
- Difficulty: Learn to take care of ALL cases of updating...



4.2 Tailed Linked List (2/10)

- A new data member: tail
- Extra maintenance needed, eg: see addFirst()

TailedLinkedList.java

```
import java.util.*;

class TailedLinkedList <E> implements EnhancedListInterface <E> {
  private ListNode <E> head = null;
  private ListNode <E> tail = null;
  private int num_nodes = 0;

  public ListNode <E> getTail() { return tail; }

  public void addFirst(E item) {
    head = new ListNode <E> (item, head);
    num_nodes++;
    if (num_nodes == 1)
      tail = head;
  }
```

New code

4.2 Tailed Linked List (3/10)

- With the new member tail, can add to the end of the list directly by creating a new method addLast()
  - Remember to update tail

TailedLinkedList.java

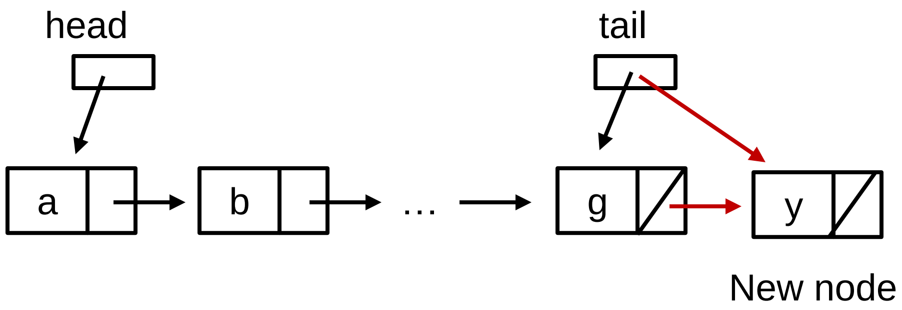
```
public void addLast(E item) {
  if (head != null) {
    tail.setNext(new ListNode <E> (item));
    tail = tail.getNext();
  } else {
    tail = new ListNode <E> (item);
    head = tail;
  }
  num_nodes++;
}
```

4.2 Tailed Linked List (4/10)

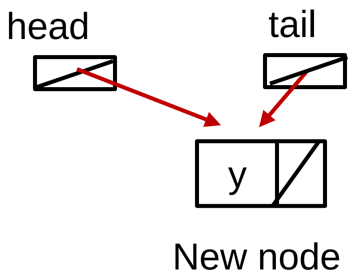
TailedLinkedList.java

```
public void addLast(E item) {
   if (head != null) {
      tail.setNext(new ListNode <E> (item));
      tail = tail.getNext();
   } else {
      tail = new ListNode <E> (item);
      head = tail;
   }
   num_nodes++;
}
```

- Case 1: head != null



- Case 2: head == null



4.2 Tailed Linked List (5/10)

- addAfter() method

TailedLinkedList.java

```
public void addAfter(ListNode <E> current, E item) {
   if (current != null) {
      current.setNext(new ListNode <E> (item, current.getNext()));
      if (current == tail)
         tail = current.getNext();
   } else { // add to the front of the list
      head = new ListNode <E> (item, head);
      if (tail == null)
         tail = head;
   }
   num_nodes++;
}
```

We may replace our earlier addFirst() method (in slide 55) with a simpler one that merely calls addAfter(). How?

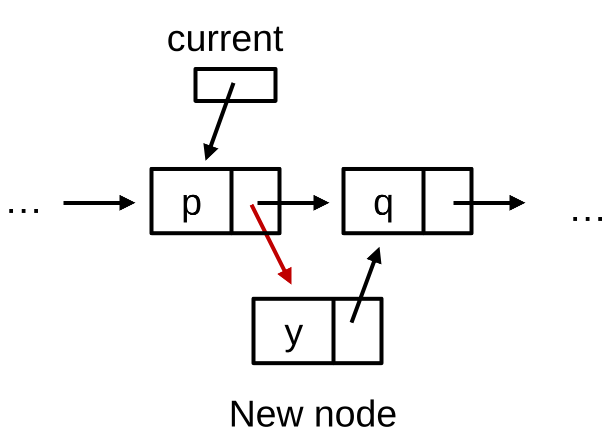
Hint: Study the removeFirst() method (slide 62).

4.2 Tailed Linked List (6/10)

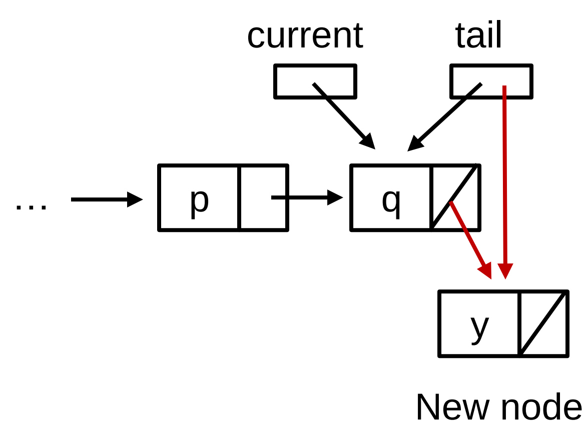
TailedLinkedList.java

```
public void addAfter(ListNode <E> current, E item) {
   if (current != null) {
      current.setNext(new ListNode <E> (item, current.getNext()));
      if (current == tail)
         tail = current.getNext();
   } else {
      . . .
   }
   num_nodes++;
}
```

- Case 1A
  - current != null; current != tail



- Case 1B
  - current != null; current == tail

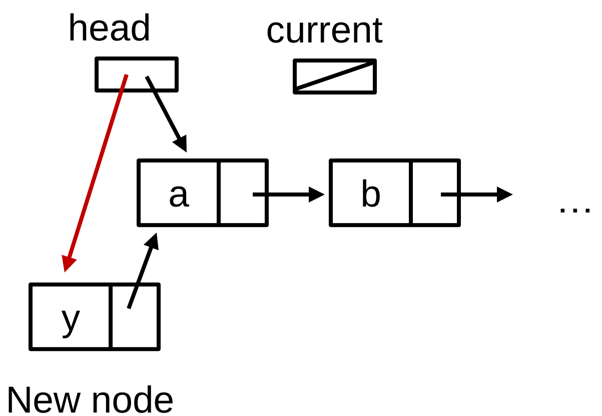


4.2 Tailed Linked List (7/10)

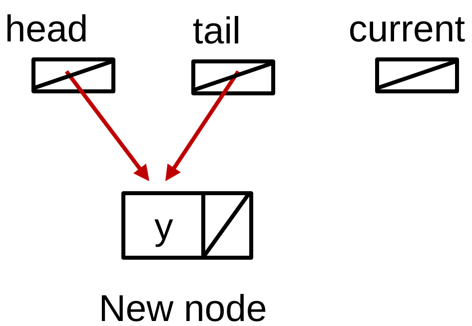
TailedLinkedList.java

```
public void addAfter(ListNode <E> current, E item) {
   if (current != null) {
      . . .
   } else { // add to the front of the list
      head = new ListNode <E> (item, head);
      if (tail == null)
         tail = head;
   }
   num_nodes++;
}
```

- Case 2A
  - current == null; tail != null



- Case 2B
  - current == null; tail == null



4.2 Tailed Linked List (8/10)

- removeAfter() method

TailedLinkedList.java

```
public E removeAfter(ListNode <E> current)
                        throws NoSuchElementException {
   E temp;
   if (current != null) {
      ListNode <E> nextPtr = current.getNext();
      if (nextPtr != null) {
         temp = nextPtr.getElement();
         current.setNext(nextPtr.getNext());
         num_nodes--;
         if (nextPtr.getNext() == null) // last node is removed
            tail = current;
         return temp;
      else throw new NoSuchElementException("...");
   else { // if current == null, we want to remove head
      if (head != null) {
         temp = head.getElement();
         head = head.getNext();
         num_nodes--;
         if (head == null) tail = null;
         return temp;
      } else throw new NoSuchElementException("...");
   }
}
```

4.2 Tailed Linked List (9/10)

- removeFirst() method
  - removeFirst() is a special case in removeAfter()

```
public E removeFirst() throws NoSuchElementException {
   return removeAfter(null);
}
```

TailedLinkedList.java

- Study the full program TailedLinkedList.java on the module website on your own.

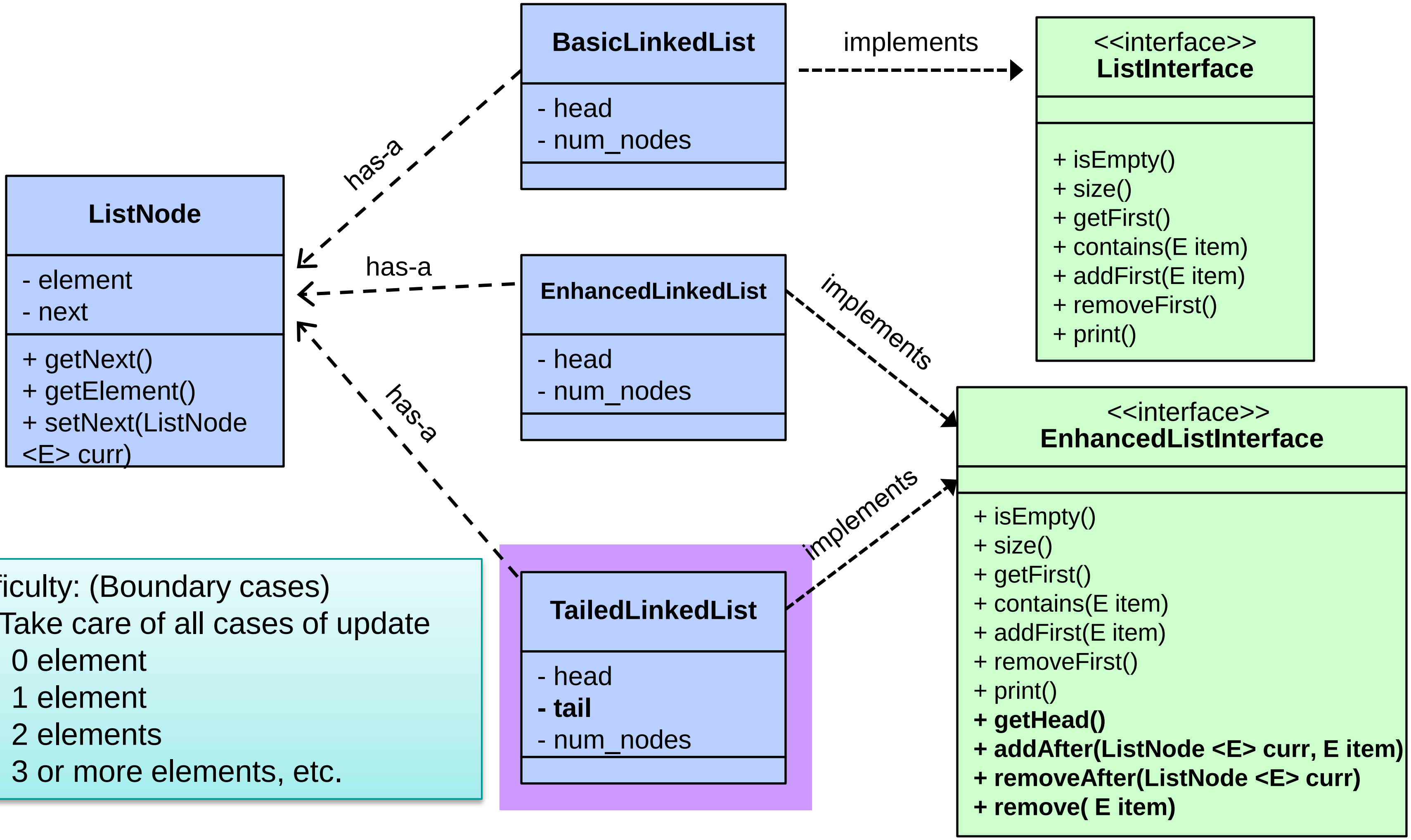
4.2 Test Tailed Linked List (10/10)

TestTailedLinkedList.java

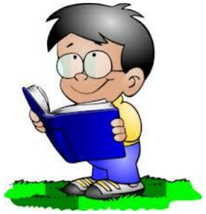
```
import java.util.*;

public class TestTailedLinkedList {
  public static void main(String [] args) throws NoSuchElementException {
    TailedLinkedList <String> list = new TailedLinkedList <String>();

    System.out.println("Part 1");
    list.addFirst("aaa");
    list.addFirst("bbb");
    list.addFirst("ccc");
    list.print();
    System.out.println("Part 2");
    list.addLast("xxx");
    list.print();
    System.out.println("Part 3");
    list.removeAfter(null);
    list.print();
  }
}
```

4. Linked Lists: Variants

OVERVIEW!



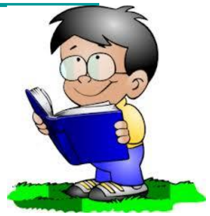
Difficulty: (Boundary cases)
Take care of all cases of update
0 element
1 element
2 elements
3 or more elements, etc.



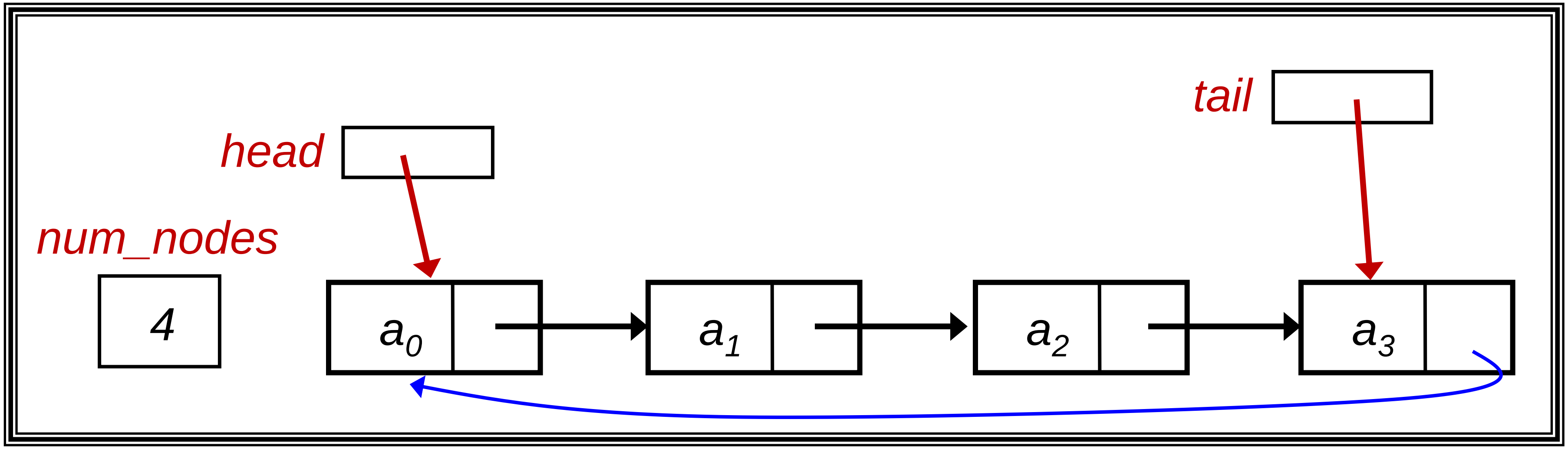
5 Other Variants

Other variants of linked lists

5.1 Circular Linked List

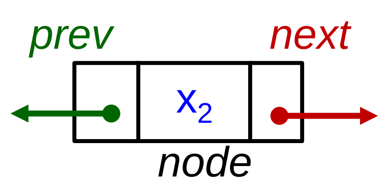


- There are many other possible enhancements of linked list
- Example: Circular Linked List
  - To allow cycling through the list repeatedly, e.g. in a **round robin system** to assign shared resource
  - Add a link from tail node of the TailedLinkedList to point back to head node
  - Different in linking need different maintenance – no free lunch!
- Difficulty: Learn to take care of ALL cases of updating, such as inserting/deleting the first/last node in a Circular Linked List
- Explore this on your own; write a class CircularLinkedList



5.2 Doubly Linked List (1/3)

- In the preceding discussion, we have a "**next**" pointer to move forward
- Often, we need to move backward as well
- Use a "**prev**" pointer to allow backward traversal
- Once again, no free lunch – need to maintain "**prev**" in all updating methods
- Instead of ListNode class, need to create a DListNode class that includes the additional "**prev**" pointer



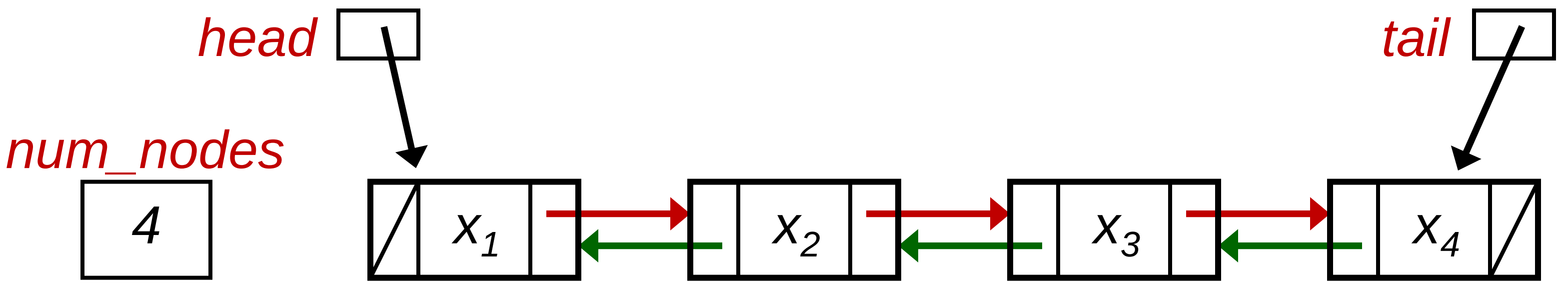
5.2 Doubly Linked List: DListNode (2/3)

DListNode.java

```
class DListNode <E> {
   /* data attributes */
   private E element;
   private DListNode <E> prev;
   private DListNode <E> next;

   /* constructors */
   public DListNode(E item) { this(item, null, null); }

   public DListNode(E item, DListNode <E> p, DListNode <E> n) {
      element = item; prev = p; next = n;
   }

   /* get the prev DListNode */
   public DListNode <E> getPrev() { return this.prev; }
   /* get the next DListNode */
   public DListNode <E> getNext() { return this.next; }

   /* get the element of the ListNode */
   public E getElement() { return this.element; }

   /* set the prev reference */
   public void setPrev(DListNode <E> p) { prev = p };
   /* set the next reference */
   public void setNext(DListNode <E> n) { next = n };
}
```

5.2 Doubly Linked List (3/3)

- An example of a doubly linked list



- Explore this on your own.
- Write a class DoublyLinkedList to implement the various linked list operations for a doubly linked list.

6 Java API: LinkedList class

Using the LinkedList class

6 Java Class: LinkedList <E>

- This is the class provided by Java library
- This is the linked list implementation of the List interface
- It has many more methods than what we have discussed so far of our versions of linked lists. On the other hand, we created some methods not available in the Java library class too.
- Please do not confuse this library class from our class illustrated here. In a way, we open up the Java library to show you the inside working.
- For purposes of sit-in labs or exam, please use whichever one as you are told if stated.

6.1 Class LinkedList: API (1/3)

Constructor Summary
`LinkedList()` Constructs an empty list.
`LinkedList(Collection c)` Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Method Summary	
`void`	`add(int index, Object element)` Inserts the specified element at the specified position in this list.
`boolean`	`add(Object o)` Appends the specified element to the end of this list.
`boolean`	`addAll(Collection c)` 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.
`boolean`	`addAll(int index, Collection c)` Inserts all of the elements in the specified collection into this list, starting at the specified position.
`void`	`addFirst(Object o)` Inserts the given element at the beginning of this list.
`void`	`addLast(Object o)` Appends the given element to the end of this list.
`void`	`clear()` Removes all of the elements from this list.

6.1 Class LinkedList: API (2/3)

boolean	contains(Object o) Returns `true` if this list contains the specified element.
Object	get(int index) Returns the element at the specified position in this list.
Object	getFirst() Returns the first element in this list.
Object	getLast() Returns the last element in this list.
int	indexOf(Object o) Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
int	lastIndexOf(Object o) Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
ListIterator	listIterator(int index) Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
Object	remove(int index) Removes the element at the specified position in this list.
boolean	remove(Object o) Removes the first occurrence of the specified element in this list.
Object	removeFirst() Removes and returns the first element from this list.
Object	removeLast() Removes and returns the last element from this list.

6.1 Class LinkedList: API (3/3)

Object	set(int index, Object element) Replaces the element at the specified position in this list with the specified element.
int	size() Returns the number of elements in this list.
Object[]	toArray() Returns an array containing all of the elements in this list in the correct order.
Object[]	toArray(Object[] a) Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.

Methods inherited from class java.util.AbstractSequentialList
iterator

Methods inherited from class java.util.AbstractList
equals, hashCode, listIterator, removeRange, subList

Methods inherited from class java.util.AbstractCollection
containsAll, isEmpty, removeAll, retainAll, toString

Methods inherited from class java.lang.Object
finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.List
containsAll, equals, hashCode, isEmpty, iterator, listIterator, removeAll, retainAll, subList

6.2 Class LinkedList (1/2)

- An example use (Page 1 of 2)

TestLinkedListAPI.java

```
import java.util.*;

public class TestLinkedListAPI {

  static void printList(LinkedList <Integer> alist) {
    System.out.print("List is: ");
    for (int i = 0; i < alist.size(); i++)
      System.out.print(alist.get(i) + "\t");
    System.out.println();
  }

  // Print elements in the list and also delete them
  static void printListv2(LinkedList <Integer> alist) {
    System.out.print("List is: ");
    while (alist.size() != 0) {
      System.out.print(alist.element() + "\t");
      alist.removeFirst();
    }
    System.out.println();
  }
```

6.2 Class LinkedList (2/2)

- An example use (Page 2 of 2)

TestLinkedListAPI.java

```
  public static void main(String [] args) {
    LinkedList <Integer> alist = new LinkedList <Integer> ();
    for (int i = 1; i <= 5; i++)
      alist.add(new Integer(i));

    printList(alist);

    System.out.println("First element: " + alist.getFirst());
    System.out.println("Last element: "  + alist.getLast());

    alist.addFirst(888);
    alist.addLast(999);
    printListv2(alist);
    printList(alist);
  }
}
```

Why “reinvent the wheel”?

- In a data structures course, students are often asked to implement well-known data structures.
- A question we sometimes hear from students: “Since there is the API, why do we need to learn to write our own code to implement a data structure like linked list?”
- Writing the code allows you to gain an indepth understanding of the data structures and their operations
- The understanding will allow you to appreciate their complexity analysis (to be covered later) and use the API effectively

7 Summary (1/2)

- We learn to create our own data structure
  - In creating our own data structure, we face 3 difficulties:
    1. Re-use of codes (inheritance confusion)
    2. Manipulation of pointers/references (The sequence of statements is important! With the wrong sequence, the result will be wrong.)
    3. Careful with all the boundary cases
  - Drawings are very helpful in understanding the cases (point 3), which then can help in knowing what can be used/manipulated (points 1 and 2)

7 Summary (2/2)

- Once we can get through this lecture, the rest should be smooth sailing as all the rest are similar in nature
  - You should try to add more methods to our versions of LinkedList, or to extend ListNode to other type of node
- Please do not forget that the Java Library class is much more comprehensive than our own – for sit-in labs and exam, please use whichever one as you are told if stated.

9 Visualising Data Structures

- See http://visualgo.net
  - Click on “Linked List, Stack, Queue”
  - (Non-linear data structures such as trees and graphs will be covered in 502043.)
- See http://www.cs.usfca.edu/~galles/visualization/Algorithms.html

End of file