Lecture 4

The List ADT and Linked Lists

1. Lists via Arrays
2. Linked Lists
3. Linked List ADT
4. Generic Java Linked Lists
5. Variations of Linked Lists
6. Class LinkedList <E> in generic Java

Readings

- Chapter 4: The List ADT

  Pages 178-183
- Chapter 4: An Array Implementation

  Pages 206-213

- Chapter 5: Linked Lists

  Pages 221-281

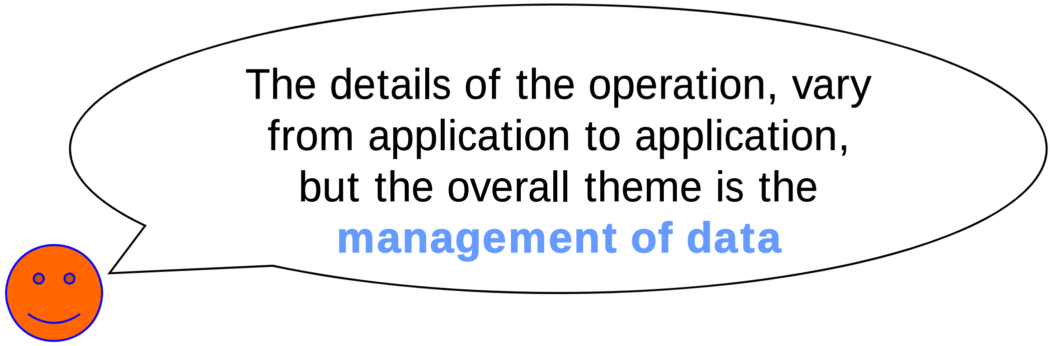
The List ADT

- Lists form one of the most basic type of data collections
  - List of groceries, modules, friends, events
- Contains items of the same type

Q: What are the operations that you would do to a list ADT?

Recap: Typical Operations on Data

- Add data to a data collection
- Remove data from a data collection
- Ask questions about the data in a data collection

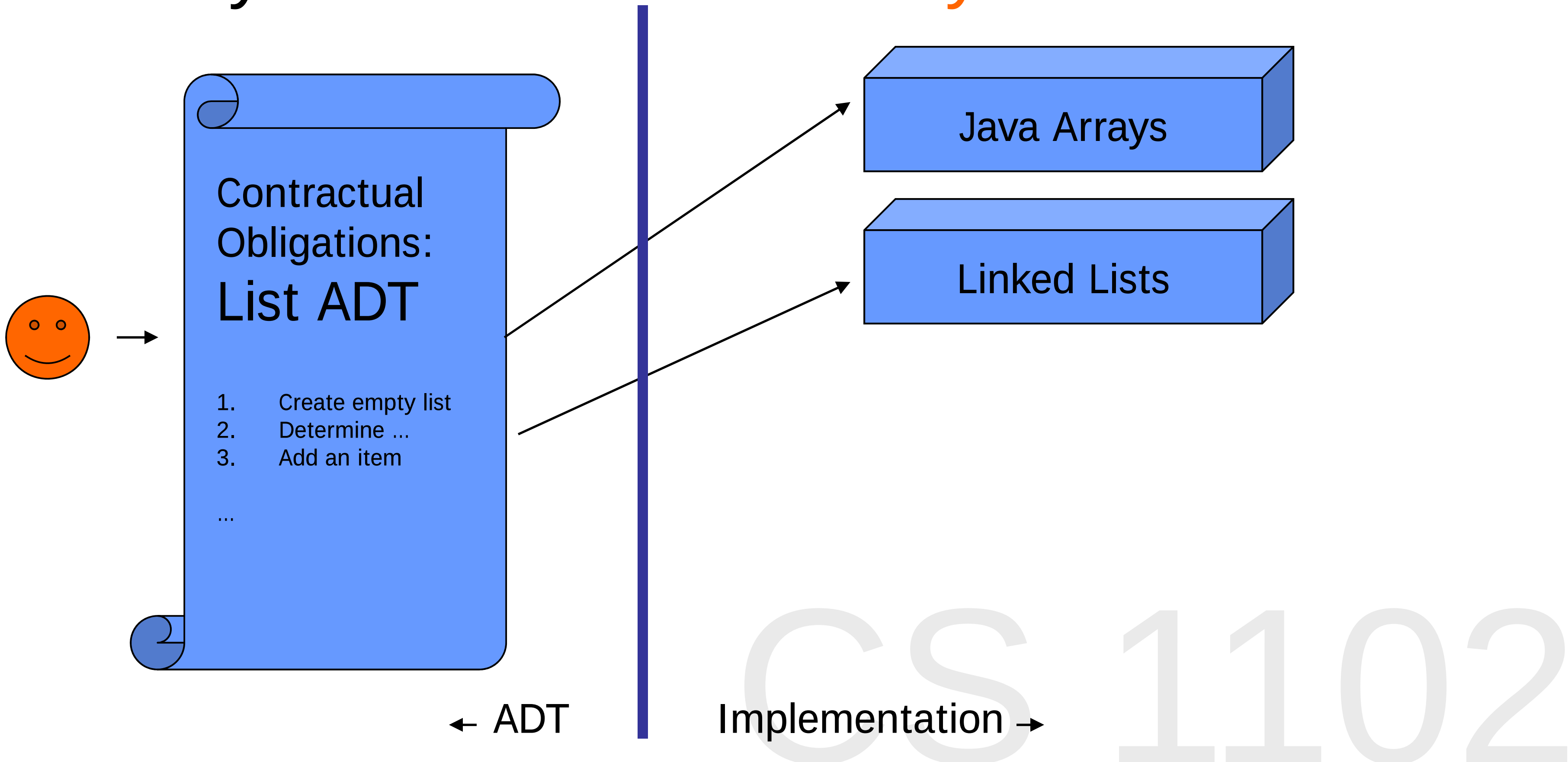


List ADT operations

1. Create an empty list
2. Determine whether a list is empty
3. Determine number of items in the list
4. Add an item at a given position
5. Remove the item at a position
6. Remove all items
7. Read an item from the list at a position

Implementations of the List ADT

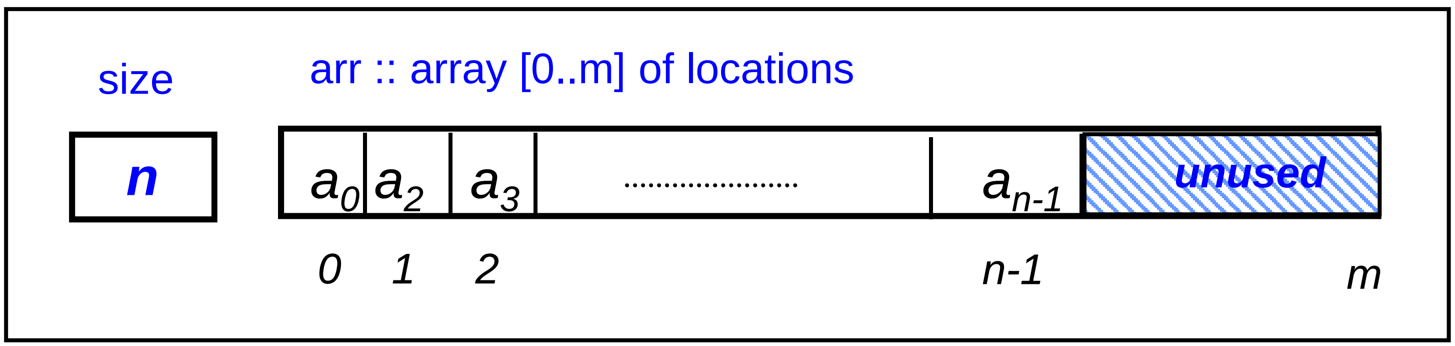
- We’re going through two implementations today of the List ADT: arrays and linked lists.



1. List via Array Implementation

Maintaining a list of data

- Straightforward approach: Use Java arrays
  - A sequence of *n* elements



Implementation issue: Generic arrays of collections in Java

- Arrays can be of many data types:

```
int[] numbers         = new int[100];        // primitive-type
String[] names        = new String[100];     // non-primitive
Integer[] numbers     = new Integer[100];    // wrapped primitive
```

- We want to design data structures/algorithms which are applicable to many data types.

  - Example: A sorting algorithm should work on integers, floats, strings, shapes, etc.
  - Object is a super class of all classes.
  - Generic object type:

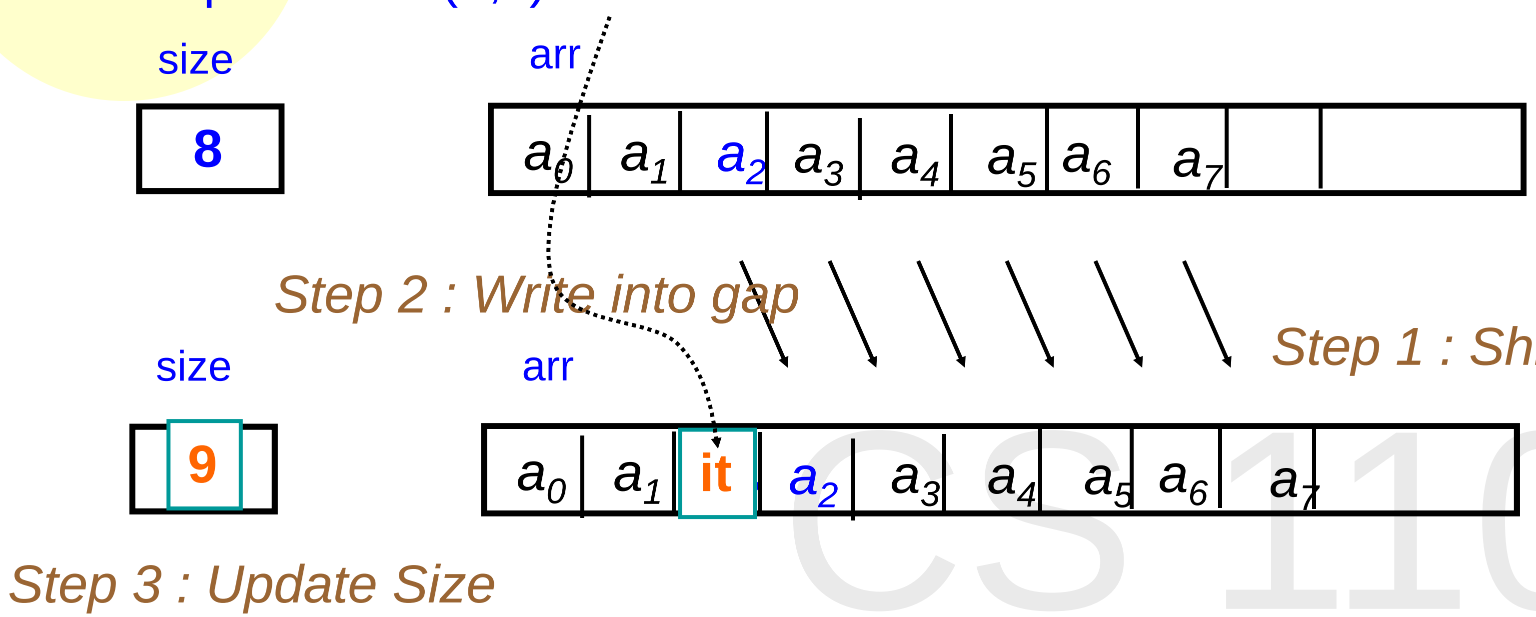
```
Object[] list    = new Object[100];
list[0]          = new String(“abc”);
list[1]          = new Integer(10);
```

Dynamic operations on Arrays – slow!

While retrieval is fast, insertion and deletion are slow:

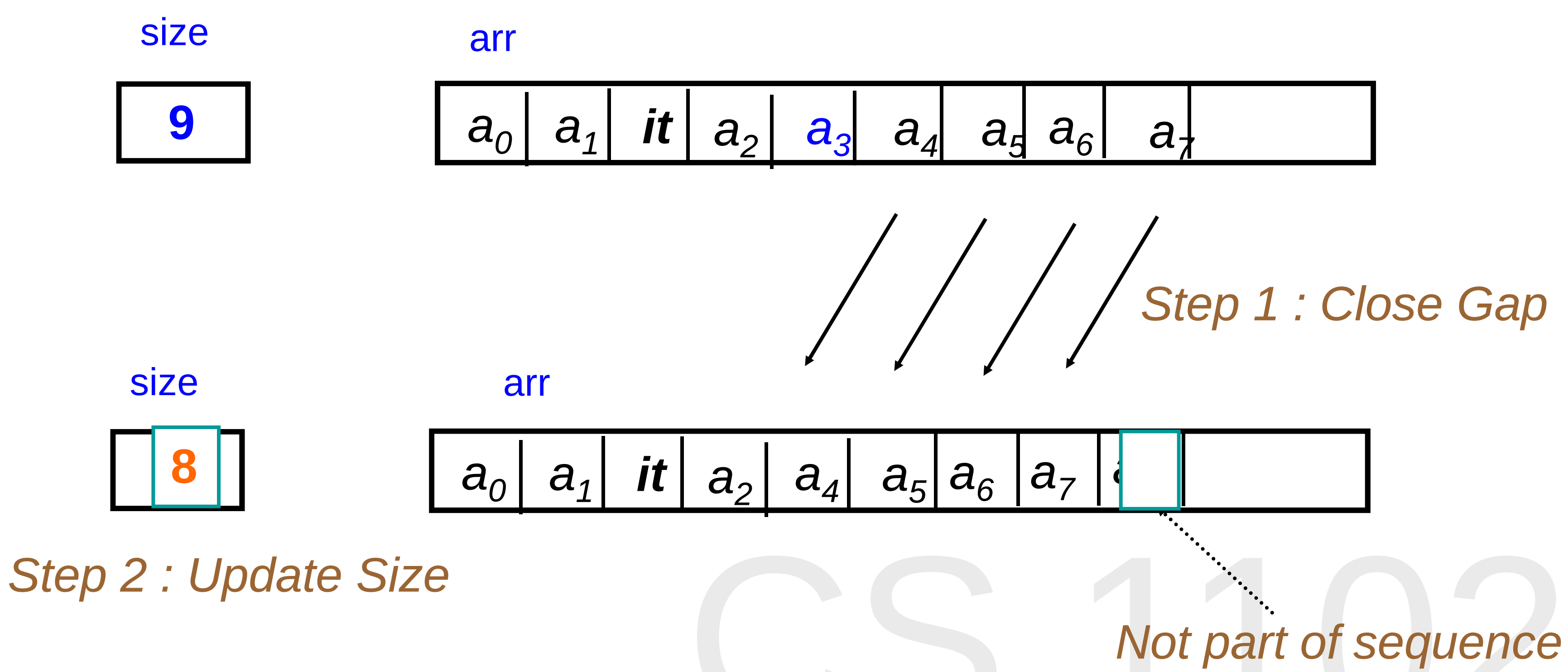
- Insert has to shift "right" to create gap
- Delete has to shift "left" to close gap of deleted item.

Example: insert(2,it)

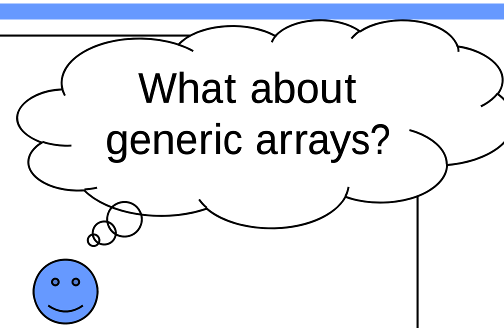


Insertion/deletion can affect all elements

Example: delete(4)



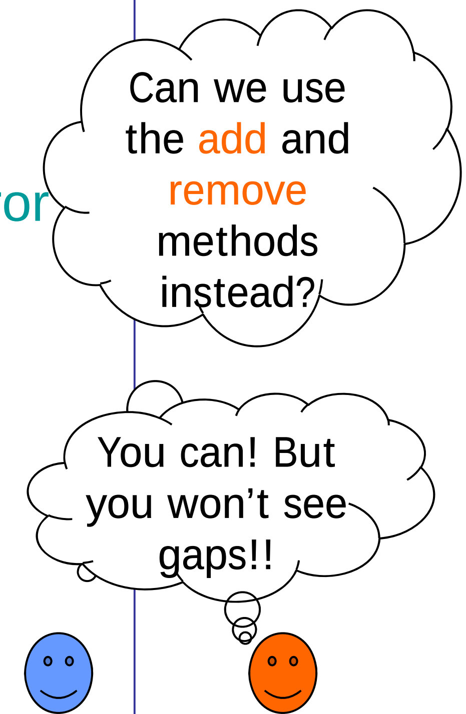
Array Implementation, Take 1




```
public class MyList {
  private static final int MAXSIZE = 1000;
  private int size = 0;
  private Object[] arr = new Object[MAXSIZE];

  public void insert (int j, Object it) {
    if (size==MAXSIZE || j>size)
      throw new ListIndexOutOfBoundsException("Error in insert");
    for (int i = size-1; i >= j; i--)
      arr[i+1] = arr[i];         // Step 1: Create gap
    arr[j] = it;                 // Step 2: Write to gap
    size++;                      // Step 3: Update size
  }

  public void delete (int j) {
    if (j>=size)
      throw new ListIndexOutOfBoundsException("Error in delete");
    for (int i = j+1; i < size; i++)
      arr[i-1] = arr[i];         // Step 1: Close gap
    size--;                      // Step 2: Update size
} }
```

The direction is important!

```
public class MyList <E> {
  private static final int MAXSIZE = 1000;
  private int size = 0;
  private ArrayList <E> arr = new ArrayList <E> (MAXSIZE);

  public void insert (int j, E it)  {
    if (size == MAXSIZE || j>size)
      throw new ListIndexOutOfBoundsException("insert error
    for (int i = size-1; i >= j; i--)
      arr.set (i+1, arr.get (i));          // Step 1: Create gap
    arr.set (j, it);                       // Step 2: Write to gap
    size++;                                // Step 3: Update size
  }

  public void delete (int j)  {
    if (j>=size)
      throw new ListIndexOutOfBoundsException("delete error");
    for (int i = j+1; i < size; i++) {
      arr.set (i-1, arr.get(i));           // Step 1: Close gap
    size--;                                // Step 2: Update size
  } }
```



How time efficient are arrays in representing lists?

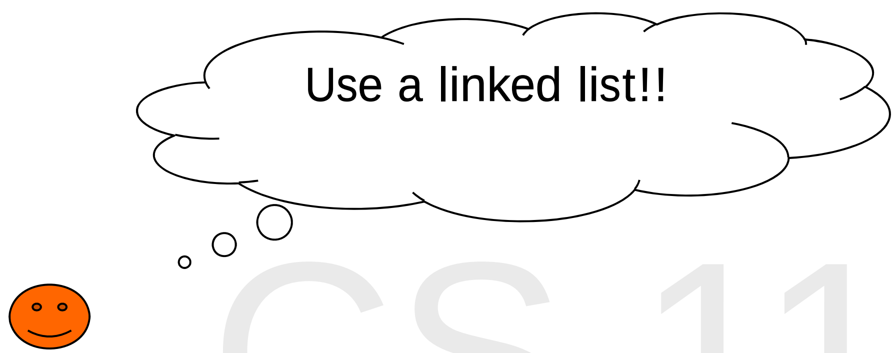
- Retrieval:
  - Always fast: Exactly one read operation.
- Insertion:
  - Best case: No shifting of elements
  - Worst case: Shifting of all n elements.
- Deletion:
  - Best case: No shifting of elements
  - Worst case: Shifting of all n elements
- Overheads in shifting elements may be significant for large collections

How space efficient are arrays in representing collections?

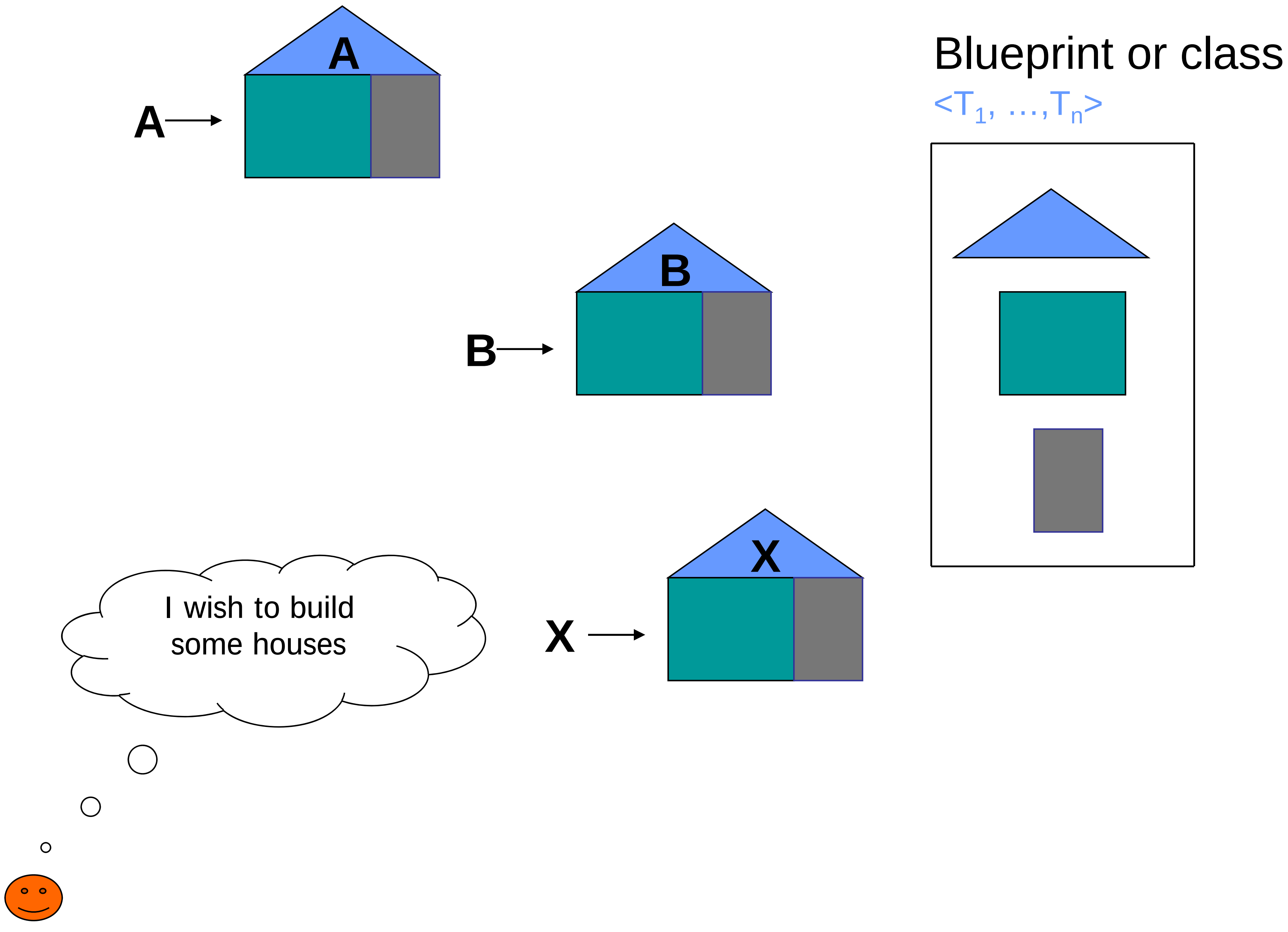
- Size of array collection limited by MAXSIZE.
- Problems:
  - We don't always know maximum size ahead of time.
  - If MAXSIZE is too liberal, unused space is wasted.
  - If MAXSIZE is too conservative, easy to run out of space.
- Idea: We can make MAXSIZE a variable, and we create/copy to a larger array whenever we run out of space.
  - No more limits on size,
  - but space wastage and copying overhead is still a problem.

When to use arrays for lists

- For fixed-size lists, arrays are great.
- For variable-size lists, where dynamic operations such as insert/delete are common, the array is a poor choice of data structure.
  - For such applications, there is a better way.

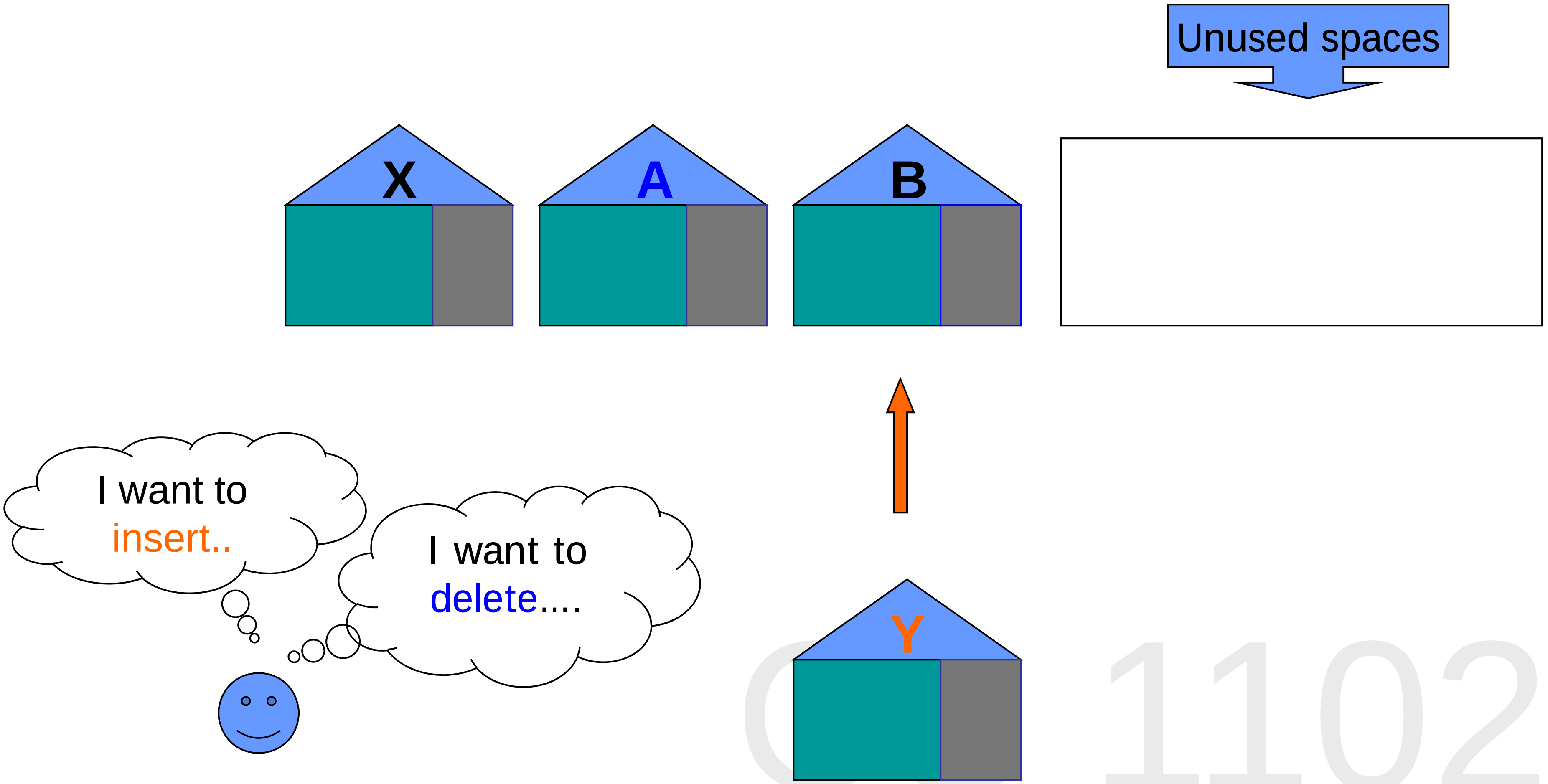


2. List via Linked List Implementation

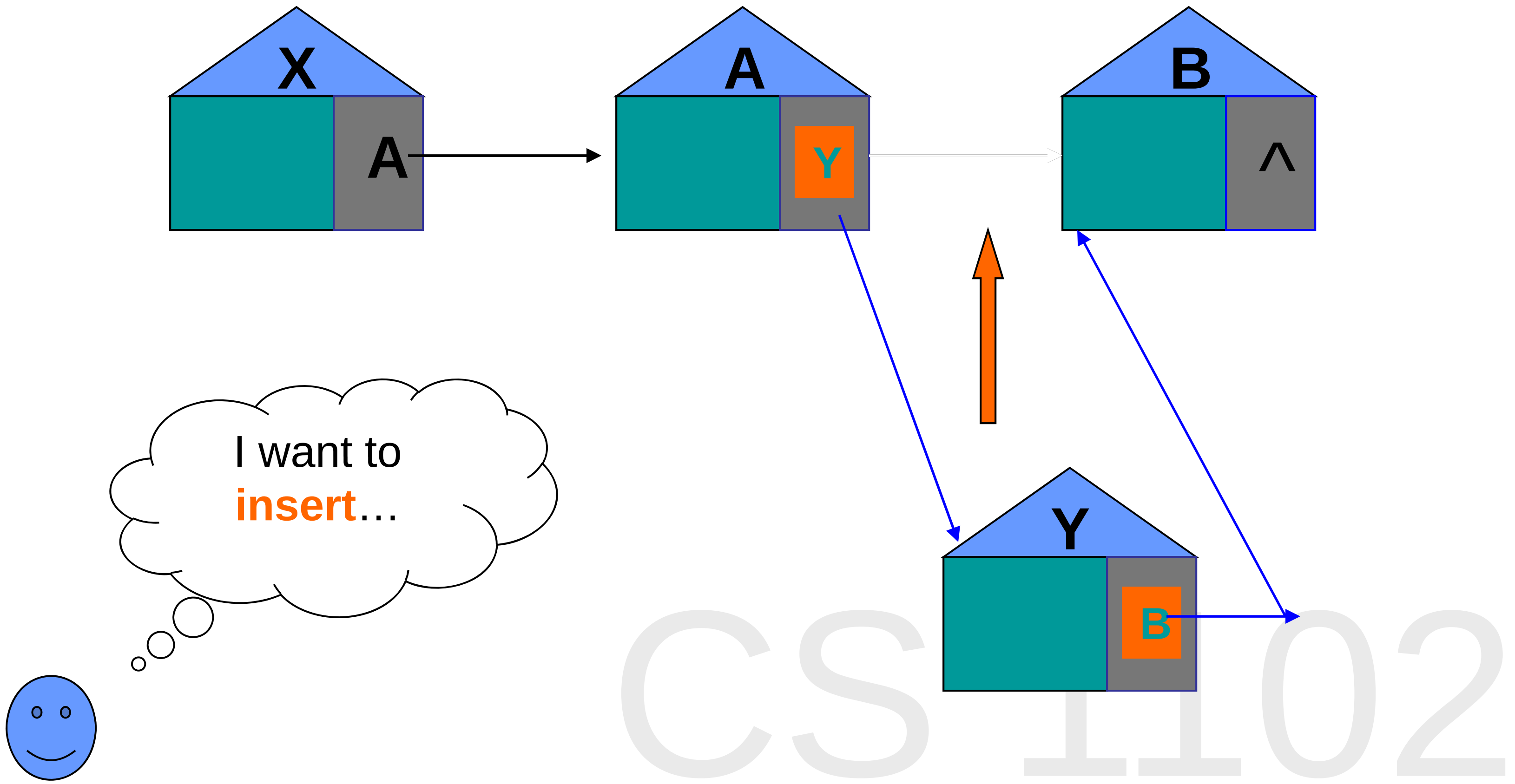


Blueprint or class
$<T_1, \ldots, T_n>$
A
A
B
B
X
X
I wish to build some houses

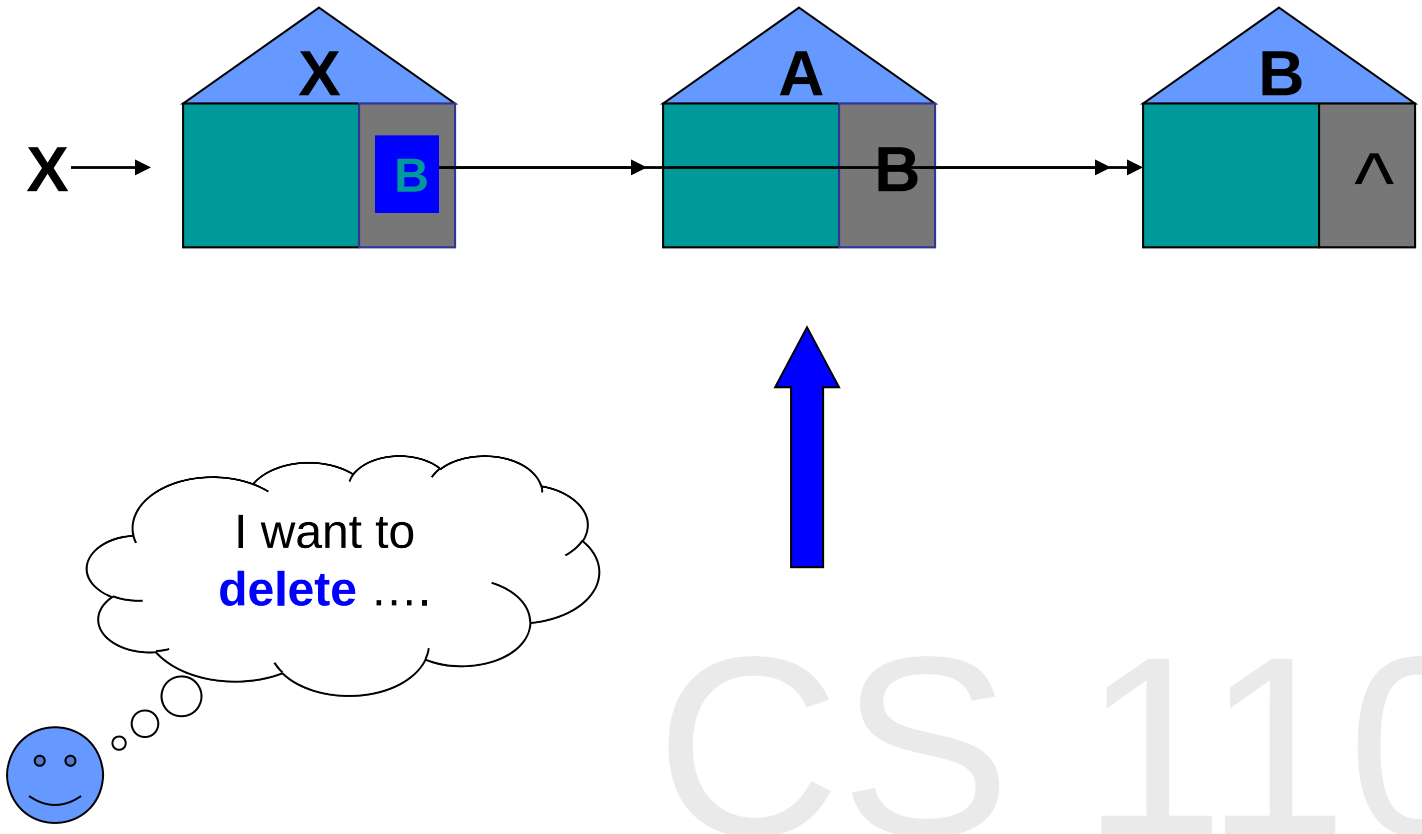
Using an array .....



Using a linked list ....



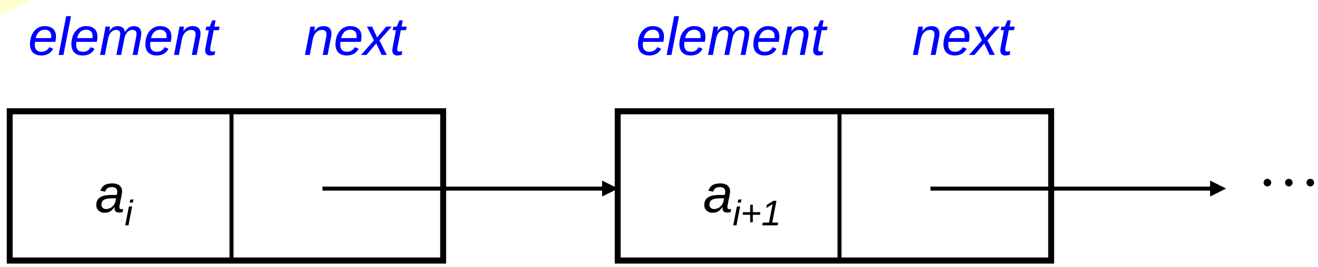
Using a linked list ....



Linked List Approach

- The problem with arrays:
  - Position of contiguous array elements denote ordering of elements.
  - Insertion needs splicing, deletion needs compacting.

- Idea (linked-lists):
  - Allow elements to be non-contiguous in memory.
  - Order the elements by associating each with its neighbour(s).



This is one element of the collection…

… and this one comes after it.

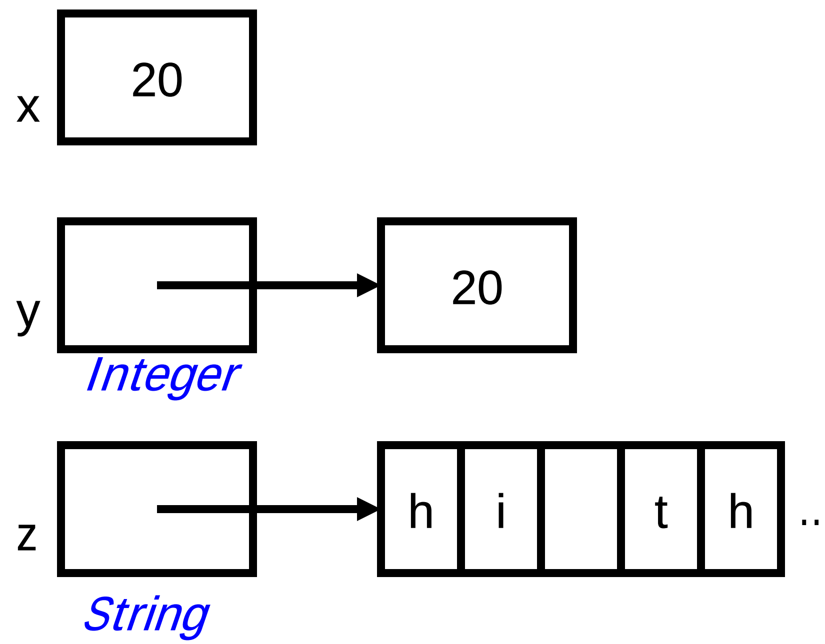
Recap: Object References

- Recall that primitive data types and reference data types are different to Java

int x = 20;

Integer y = new Integer(20);

String z = new String("hi there");

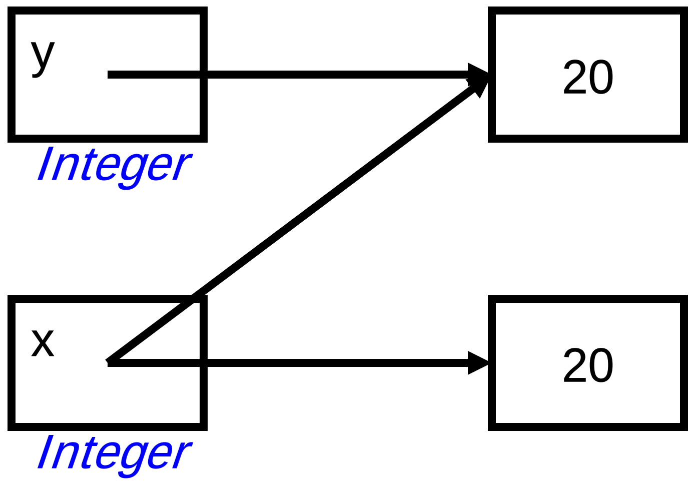


An Object of a class only comes into existence when you apply the new operator. A reference variable only contains a reference or pointer to an object.

Recap: Object References 2

- Let's look at this in more detail:

```
Integer y = new Integer(20);
Integer x;
x = new Integer(20);
if (x == y) { S.o.p("1. x == y") }
x = y;
if (x == y) { S.o.p("2. x == y") }
```



2. x == y

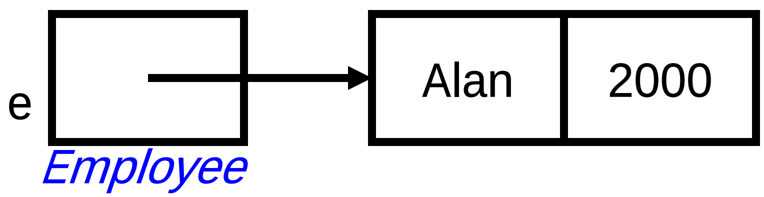
// S.o.p = System.out.println

Quiz Time: References

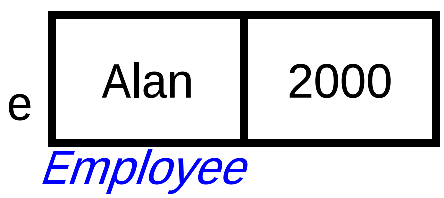
- Q: What is the representation of e?

Employee e = new Employee(“Alan”, 2000);

A.

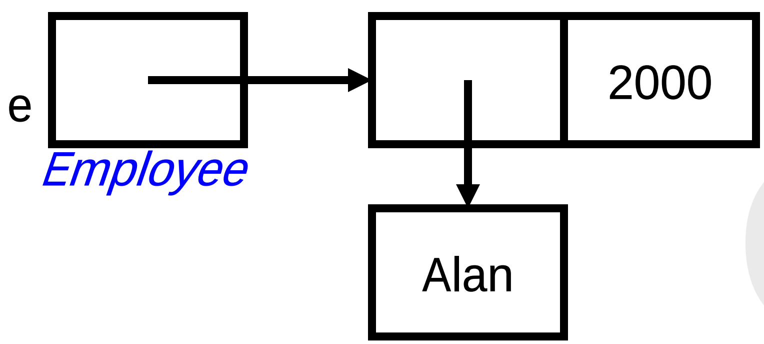


B.

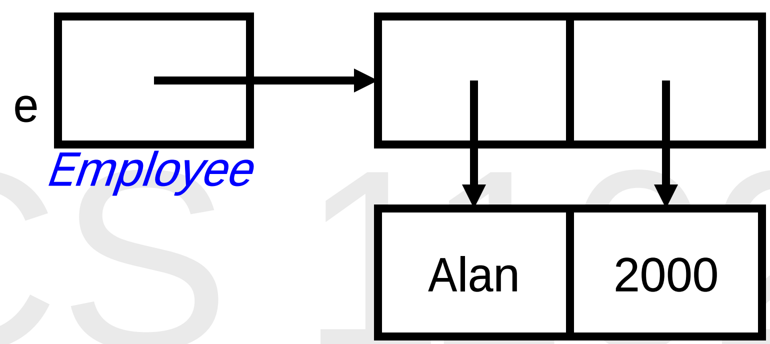



```
class Employee {
   static final int MAX_NUMBER = 50;
   private String name;
   private int salary;
   //etc
}
```

C.



D.



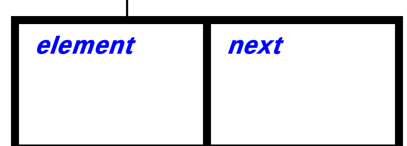
Designing a linked-list node

- We need to ‘wrap’ each data element within a ‘linked-list node’.

```
class ListNode {
    private Object element;
    private ListNode next;

    public ListNode (Object item)
    { element = item; next = null; }

    public ListNode (Object item, ListNode n)
    { element = item; next = n; }
}
```

A data element in the collection…

… and what comes after it.

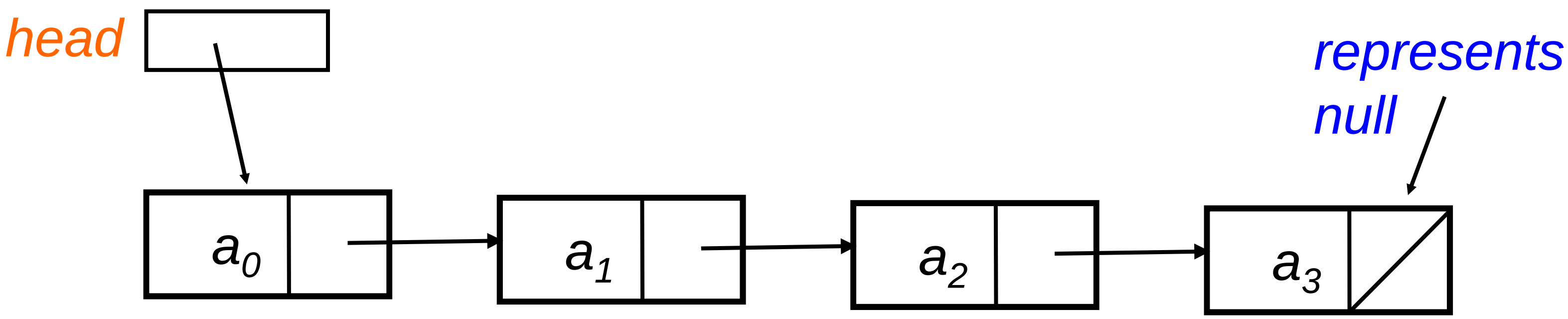


A Linked List node using generic Java

```
class ListNode <E> {
  private E element;
  private ListNode <E> next;

  public ListNode (E item)
    { element = item; next = null; }

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

Example: A linear Linked List

- Sequence of 4 items $< a_0, a_1, a_2, a_3 >$ can be represented by:

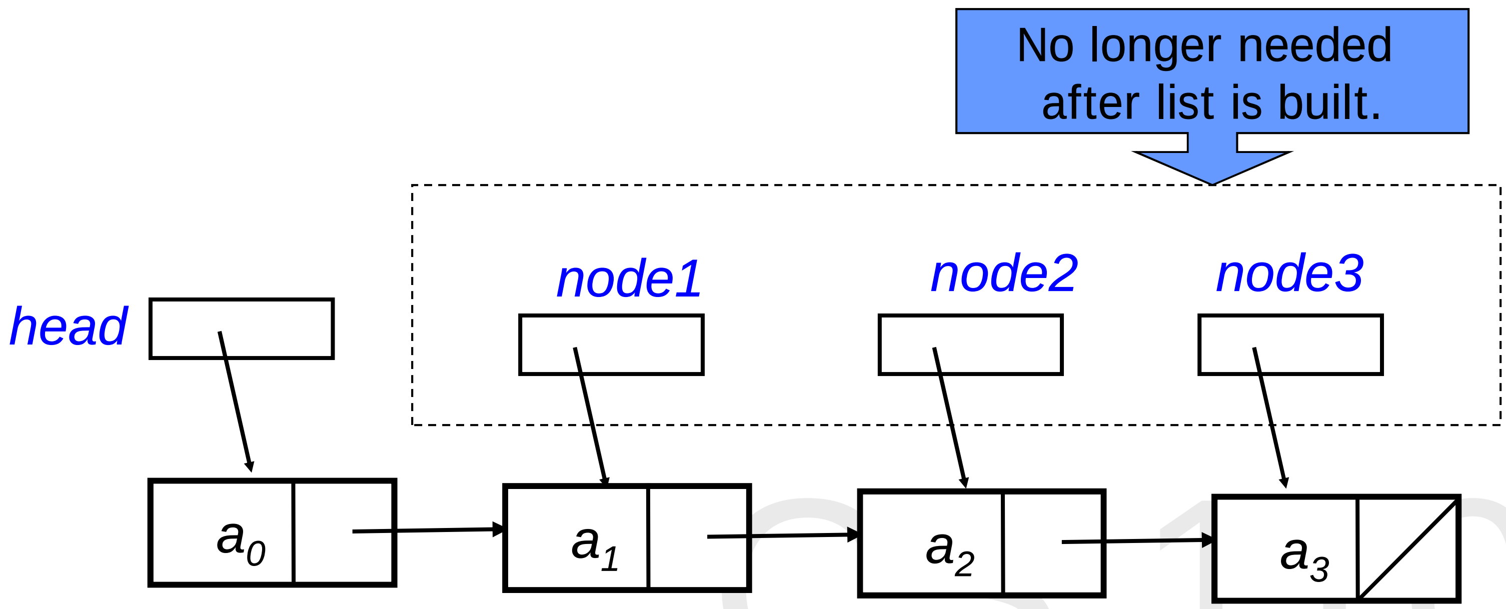


We need a head to indicate where the first node is. From the head we can get to the rest.

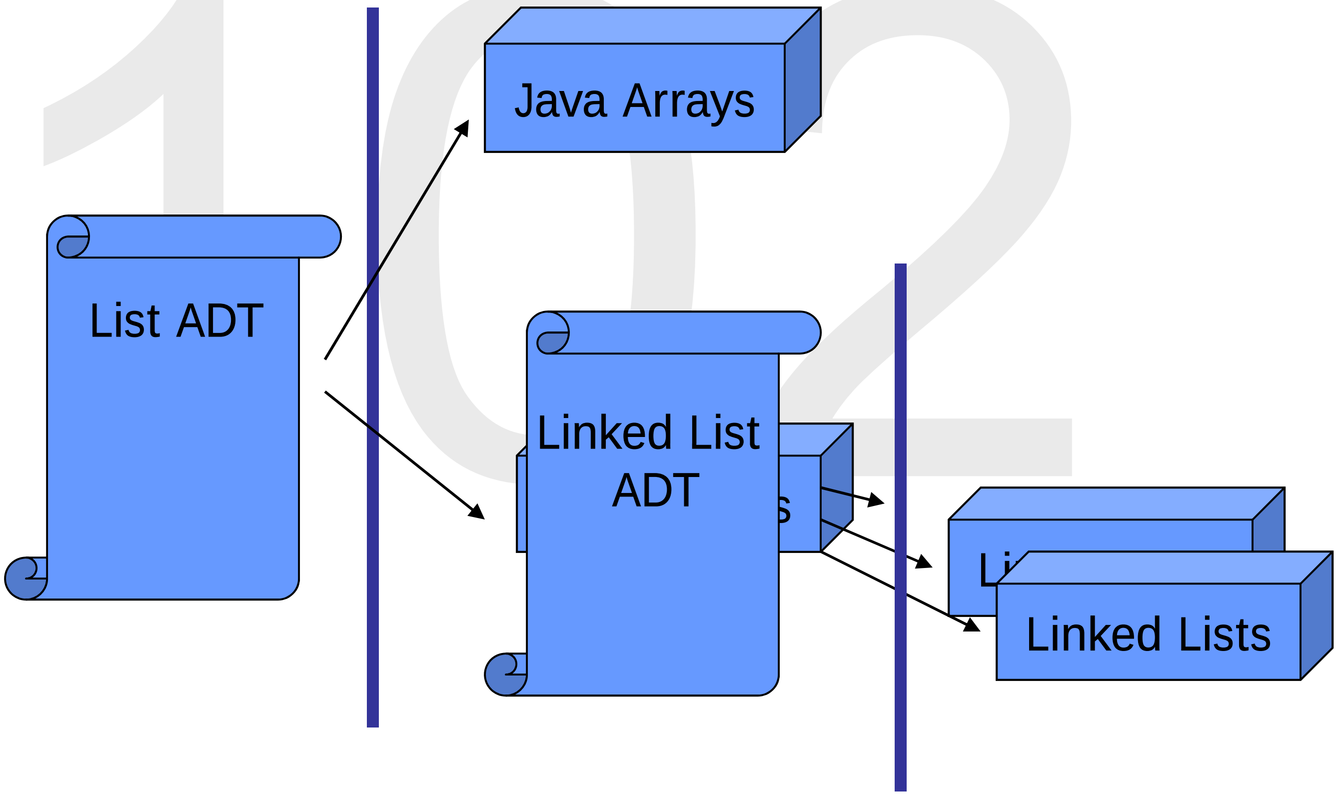
Building a list in reverse order

The earlier sequence can be built by:

```
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);
```

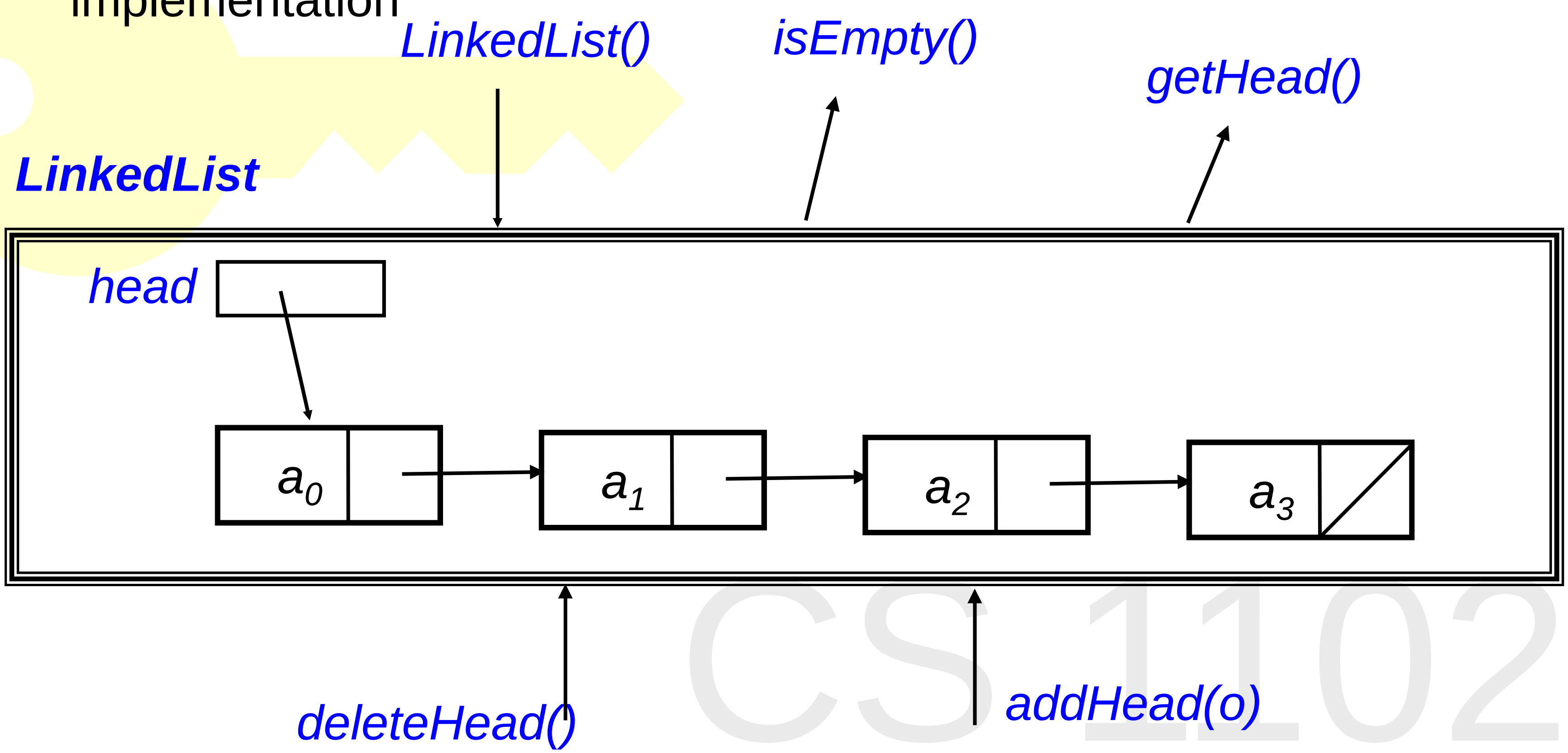


3. Linked List ADT



3. Linked List ADT

- We can provide an ADT for linked lists
  - This can help hide unnecessary internal details
  - With the ADT, we can use a linked list without worrying about its implementation

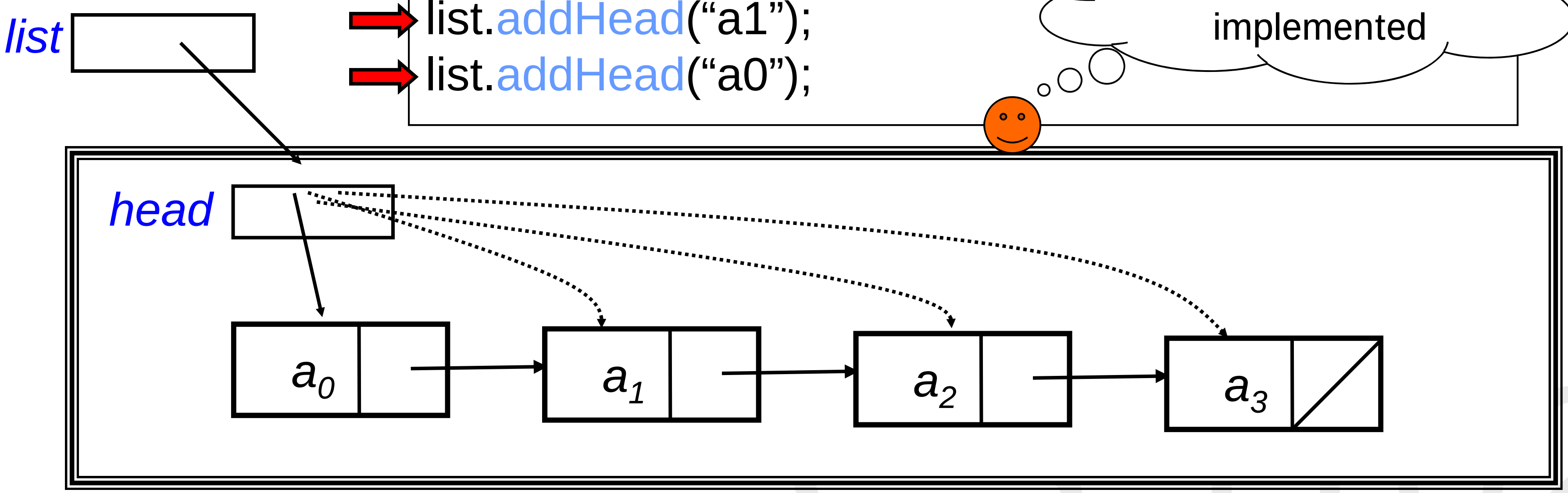


Example: using the Linked List ADT

- Sequence of four items <a0,a1,a2,a3> can be built, as follows:

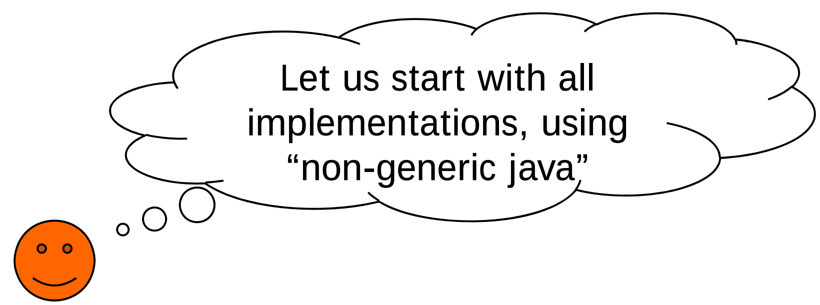
```
LinkedList <String> list = new <String> LinkedList();
list.addHead("a3");
list.addHead("a2");
list.addHead("a1");
list.addHead("a0");
```

I don't care how addHead is implemented

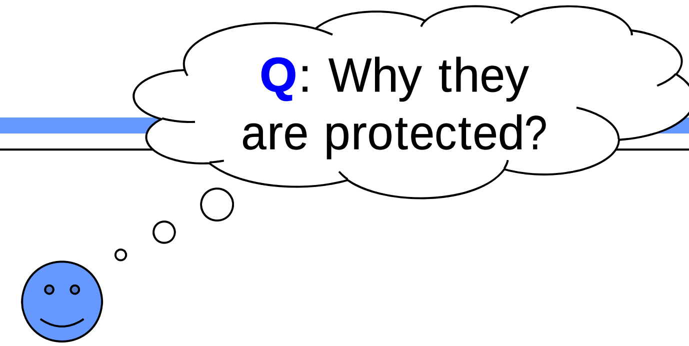


Extending the Exception Class

```
public class ItemNotFoundException extends Exception {
  public ItemNotFoundException (String msg) {
    super (msg);
  }
}
```



The class ListNode




```
class ListNode {
  protected Object element;
  protected ListNode next;

  public ListNode (Object item)                    // add to rear
    { element = item; next = null; }
  public ListNode (Object item, ListNode n) // add to front
    { element = item; next = n; }
  public ListNode getNext (ListNode current) throws
     ItemNotFoundException {
    if (current == null) throw new ItemNotFoundException ("No next node")
    else return current.next; }
  public Object getElement (ListNode current) throws
     ItemNotFoundException {
    if (current == null) throw new ItemNotFoundException ("No such node");
    else return current.element;
  } }
```

Linked List ADT - BasicLinkedListInterface

```
public interface BasicLinkedListInterface {
  public boolean isEmpty ();
  public int size ();

  public Object getHeadElement () throws ItemNotFoundException;
  public ListNode getHeadPtr ();

  public void addHead (Object item);
  public void deleteHead () throws ItemNotFoundException;
}
```

Why use an interface instead of a class?

- A user should be able to make use of a linked list data structure with only these operations.
- Implementation details (such as the ListNode class) should be hidden from the user.

BasicLinkedListInterface Implementation

```
class BasicLinkedList implements BasicLinkedListInterface {
  protected ListNode head = null;
  protected int num_nodes = 0;

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

  public Object getHeadElement() throws ItemNotFoundException {
    if (head==null) throw new
      ItemNotFoundException(“Cannot get from an empty list!”);
    else return head.element;
  }

  public ListNode getHeadPtr () { return head; }

  public void addHead(Object item) {
    head = new ListNode(item, head);
    num_nodes++;
  }

  public void deleteHead () throws ItemNotFoundException {
    if (head ==null) throw new
      ItemNotFoundException (“Cannot delete from an empty list!”);
    else { head = head.next; num_nodes--; }
  }
}
```

Example: Creating a BasicLinkedList

```
class TestBasic {
  public static void main (String [ ] args)
    throws ItemNotFoundException {
    BasicLinkedList bl = new BasicLinkedList ();
    bl.addHead ("aaa");
    bl.addHead ("bbb");
    bl.addHead ("ccc");
    printList (bl);
  }

  static void printList (BasicLinkedList bl)
    throws ItemNotFoundException {
    ListNode tempPtr = bl.getHeadPtr ();
    while (tempPtr != null) {
      System.out.print (tempPtr.getElement (tempPtr));
      tempPtr = tempPtr.getNext (tempPtr);
    }
    System.out.println ();
} }
```

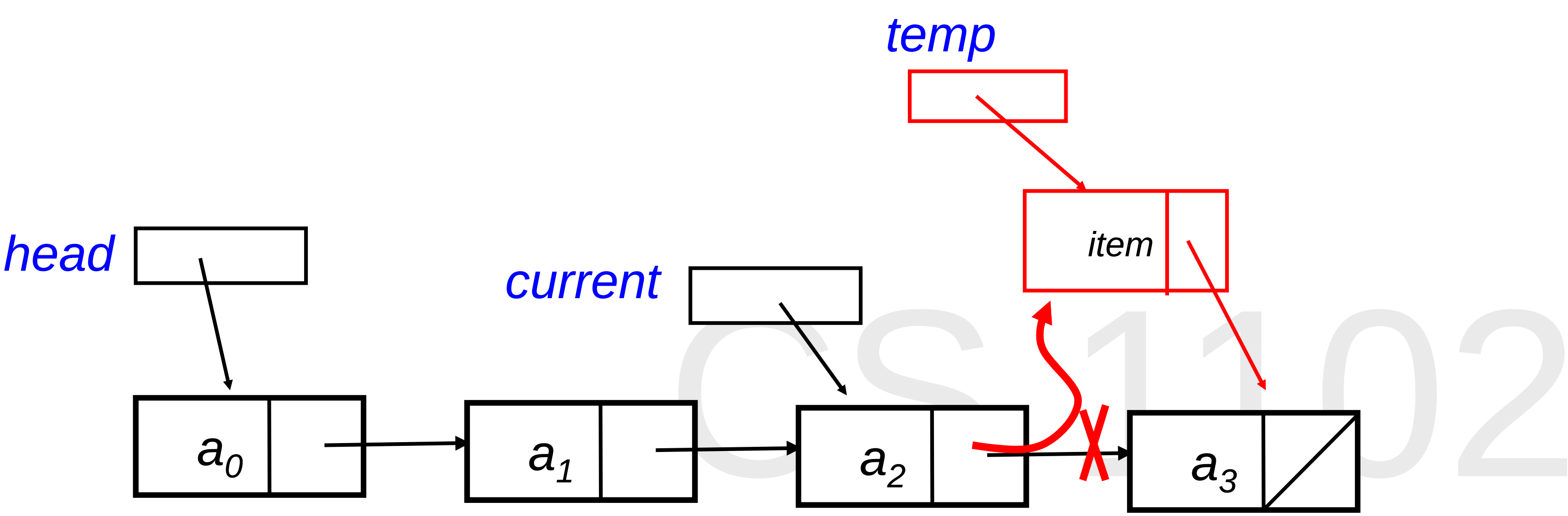
Q: Do we need this throws declaration?
A: Yes B: No C: Maybe??

ExtendedLinkedList – An Enhanced BasicLinkedList

```
class ExtendedLinkedList extends BasicLinkedList {
   public void insertAfter (ListNode current, Object item)
    { … }

   public void deleteAfter (ListNode current)
      throws ItemNotFoundException
   { … }

   public boolean exists (Object item) throws ItemNotFoundException
   { … }

   public void delete (Object item) throws ItemNotFoundException
   { … }
}
```

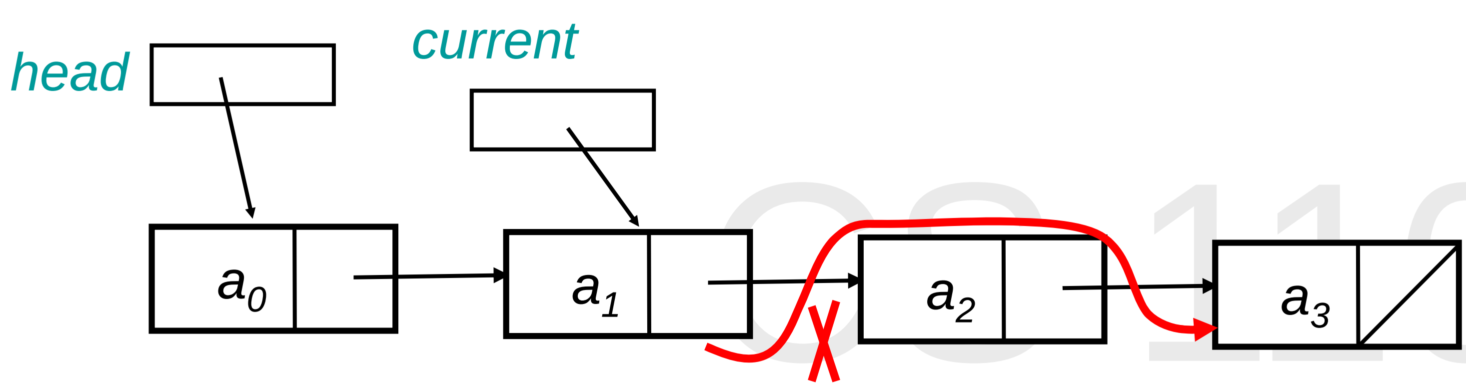
Implementing insertAfter

```
public void insertAfter (ListNode current, Object item) {
  ListNode temp;
  if (current != null) {
    temp = new ListNode (item, current.next);
    current.next = temp;
    num_nodes++;
  } else {
    // if current is null, insert item at beginning.
    head = new ListNode (item, head);
  }
}
```



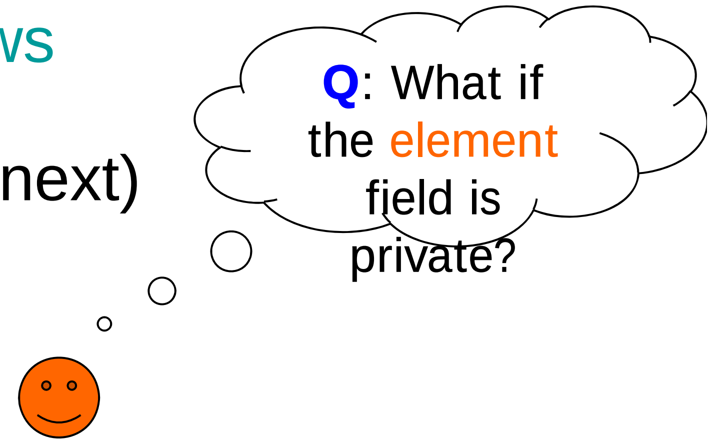
Implementing deleteAfter

```
public void deleteAfter (ListNode current) throws ItemNotFoundException {
  if (current != null) {
    if (current.next != null) {
      current.next = current.next.next; num_nodes--;
    } else
      throw new ItemNotFoundException("No Next Node to Delete");
  } else { // if current is null, assume we want to delete head.
    head = head.next; num_nodes--;
  }
}
```



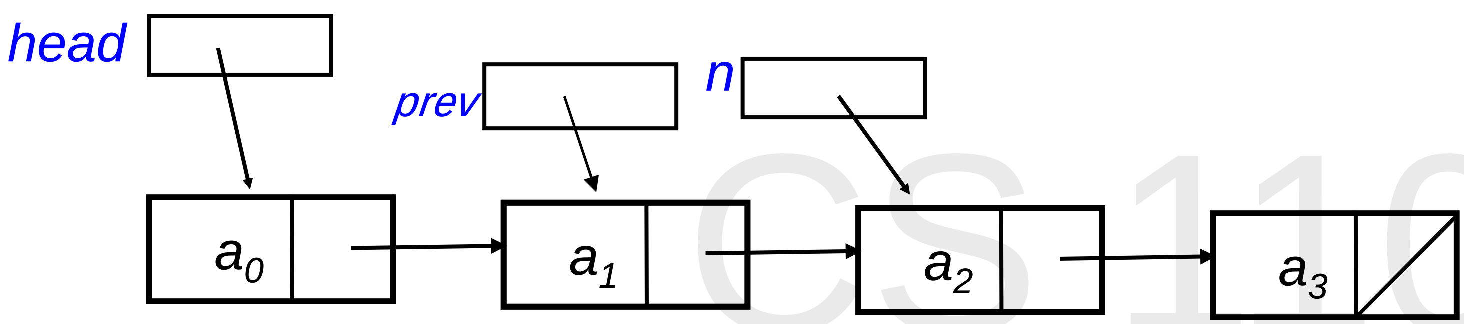
Implementing exists

```
public boolean exists (Object item) throws
    ItemNotFoundException {
  for (ListNode n = head; n != null; n = n.next)
    if (n.element.equals(item))
      return true;
  return false;
}
```



ExtendedLinkedList – delete Method

```
public void delete (Object item) throws ItemNotFoundException {
  for (ListNode n=head, prev=null;    n!=null;    prev=n, n=n.next)
    if (n.element.equals(item) {
      deleteAfter (prev);
      return;
    }
  throw new ItemNotFoundException("Can\'t find item to delete");
}
```



Creating an ExtendedLinkedList

- Assume that we have this class:

```
public class ComplexNo {
   private int realPart, imagePart;
   ComplexNo (int r, int i) {
      realPart = r;
      imagPart = i;
   }
   public String toString () {
     return "Complex (" + realPart
     + ", " + imagePart + ")";
   }
}
```

- And this method:

```
// part of ListDriver, on next slide
static void printList (ListNode front)
   throws ItemNotFoundException {
  while (front != null) {
    System.out.print
     (front.getElement (front)  + "  ");
    front = front.getNext (front);
  }
  System.out.println ();
}
```

List Driver, example (cont)

```
public class ListDriver {
  public static void main (String [] args) throws ItemNotFoundException {
    ExtendedLinkedList bl = new ExtendedLinkedList ();
    bl.addHead ("bbb");
    bl.addHead (new ComplexNo (2, 3));
    bl.addHead ("aaa");
    bl.addHead ("ccc");

    ListNode current = bl.getHeadPtr ();
    bl.insertAfter (current, "xxx");
    bl.insertAfter (bl.head, "yyy");
    bl.insertAfter (bl.head, new ComplexNo (6, 6));
Q: printList (bl.getHeadPtr ());
    System.out.println();
    bl.delete ("aaa");
    ListNode front = bl.getHeadPtr ();
    printList (front);
  }
```

Linked Lists with Generic Java

1. ListNode <T>
2. BasicLinkedListInterface <T>
3. BasicLinkedList <T>
4. ExtendedLinkedList <T>
5. Using ExtendedLinkedList <T>

1. Class ListNode <T>

```
public class ListNode <T> {
  protected T element;
  protected ListNode <T> next;

  public ListNode (T item) {
    element = item;
    next = null;
  }
  public ListNode (T item, ListNode <T> n)  {
    element = item;
    next = n;
  }
  // more on the next slide
```

Class ListNode <T> (cont)

```
public ListNode getNext (ListNode <T> current) throws
    ItemNotFoundException {
  if (current == null) throw new ItemNotFoundException ("No next node");
  return current.next;
}
public T getElement (ListNode <T> current) throws ItemNotFoundException
    {
  if (current == null) throw new ItemNotFoundException ("No next node");
  else return current.element;
}
// end class ListNode <T>
```

2. Class BasicLinkedListInterface <T>

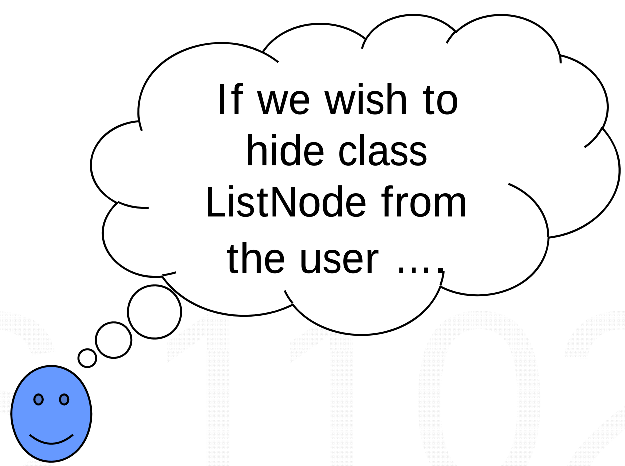
```
public interface BasicLinkedListInterface <T> {

  public boolean isEmpty ();
  public int size ();
  public ListNode <T> getHeadPtr ();
  public T getHeadElement () throws ItemNotFoundException;
  public void addHead (T item);
  public void deleteHead ()  throws ItemNotFoundException;

}
```

3. Class BasicLinkedList <T>

```
public class BasicLinkedList <T>
        implements BasicLinkedListInterface <T> {
  protected ListNode <T> head = null;
  protected int num_nodes = 0;

  public boolean isEmpty () { return (head == null); }
  public int size () { return num_nodes; }
  public ListNode <T> getHeadPtr () { return head; }
  public T getHeadElement () throws ItemNotFoundException {
    if (head == null) throw new ItemNotFoundException("Cannot get from an
     empty list");
    else return head.element;
  }
  public void addHead (T item) {
    head = new ListNode <T> (item, head);
    num_nodes++;
  }
```

Class BasicLinkedList <T> (Cont)

```
public void deleteHead() throws ItemNotFoundException {
  if (head == null) throw new ItemNotFoundException ("Cannot delete from
     an empty list");
  else {
    head = head.next;
    num_nodes--;
  } }

public void printList() throws ItemNotFoundException {
  ListNode <T> tempPtr = head;
  while (tempPtr != null) {
    System.out.print (tempPtr.element + " -- ");
    tempPtr = tempPtr.next;
  }
System.out.println ();
}
// end class
```



4. Class ExtendedLinkedList <T>

```
public class ExtendedLinkedList <T> extends BasicLinkedList <T> {
  public void deleteAfter (ListNode <T> current) throws
     ItemNotFoundException {
    if (current != null) {
      if (current.next!=null)
        current.next = current.next.next;
      else
        throw new ItemNotFoundException("No next node to delete");
      --num_nodes;
    } else { // If current is null, assume we want to delete head.
      head = head.next;
      --num_nodes;
    }
  }
  // more on next slide …
```

Class ExtendedLinkedList <T> (cont)

```
public void insertAfter (ListNode <T> current, T item) {
  ListNode <T> temp;
  if (current != null) {
    temp = new ListNode <T> (item, current.next);
    current.next = temp;
    num_nodes++;
  } else { // If current is null, insert item at beginning.
    head = new ListNode <T> (item, head);
    num_nodes++;
  } }
public boolean exists (T item) throws ItemNotFoundException {
  for (ListNode <T> n = head; n != null; n=n.next) {
    if (n.element.equals(item))
      return true;
  }
  return false;
}
```

Class ExtendedLinkedList <T> (cont)

```
public void delete (T item) throws ItemNotFoundException {
  for (ListNode <T> n = head, prev = null;    n != null;   prev = n, n = n.next) {
    if (n.element.equals (item)) {
      if (prev == null) {
        head = head.next;
        num_nodes--;
      } else {
        prev.next = n.next;
        num_nodes--;
      }
      return; // Note: returns after first occurrence
    }
  }
  throw new ItemNotFoundException ("Can\'t find item to delete");
}
// end of ExtendedLinkedList class
```

5. Using ExtendedLinkedList

```
public class ExtendedListDriver {
  public static void main (String [] args) throws ItemNotFoundException {
    ExtendedLinkedList <String> bl = new ExtendedLinkedList <String> ();
    bl.addHead ("bbb");
    bl.addHead ("aaa");
    bl.addHead ("ccc");
    bl.printList ();
    bl.insertAfter (bl.getHeadPtr(), "xxx");
    bl.insertAfter (bl.getHeadPtr(), "yyy");
    bl.printList ();
    bl.delete ("aaa");
    bl.printList ();
  } // main
} // class ELD
```

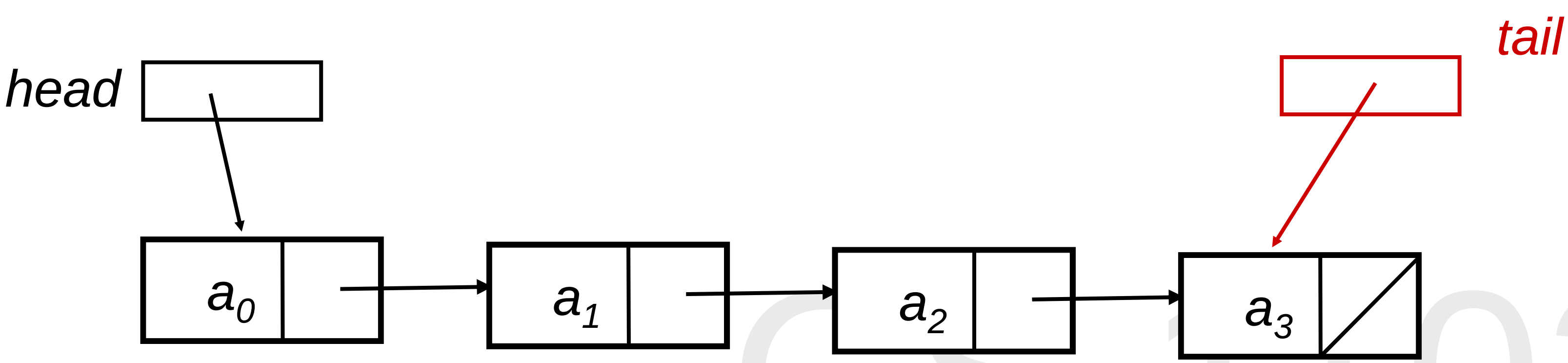
```
1. ccc -- aaa -- bbb --
2. ccc -- yyy -- xxx -- aaa -- bbb --
3. ccc -- yyy -- xxx -- bbb --
4. yyy -- xxx -- ccc -- aaa -- bbb --
5. yyy -- xxx -- ccc -- bbb --
```

Variations of Linked Lists

1. Tailed linked-list
2. Doubly linked-list
3. Circular linked-list
4. Generic Class LinkedList <E>

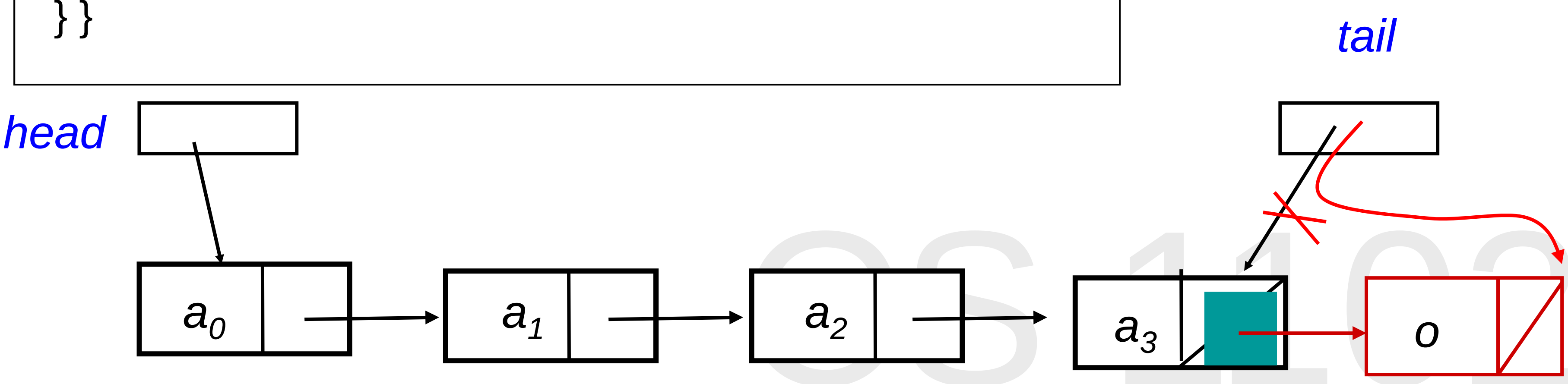
1. Tailed Linked List

- Motivation: Adding to the end of a linked list is slow. We want more efficient access to the end of linked list
- Solution: Add a tail pointer to the linked-list data structure.
  - Useful for queue-like structures
  - Unlike the head, we can't traverse the nodes from the tail.



TailedLinkedList – A subclass of ExtendedLinkedList

```
class TailedLinkedList extends ExtendedLinkedList {
  private ListNode tail = null;
  public void addTail (Object o) {
    if (tail != null) {
      tail.next = new ListNode(o);
      tail = tail.next; num_nodes++;
    } else { // list is empty
       tail = new ListNode (o);
      head = tail; num_nodes++;
    }
  } }
```



TailedLinkedList – changing methods

- Functions that add or delete nodes may affect the tail pointer.
  We must provide code to cater to this possibility

```
private void insertAfter (ListNode current, Object item) {
  ListNode temp;
  if (current != null) {
    temp = new ListNode(item, current.next);
    current.next = temp; num_nodes++;
    if (temp.next == null) tail = temp;
  } else { // if current is null, insert item at beginning.
    head = new ListNode (item, head); num_nodes++;
    if (tail == null) tail = head;
  } }
```

- Note: This method overrides the one in ExtendedLinkedList.

insertAfter(): another implementation

```
private void insertAfter (ListNode current, Object item) {
  super.insertAfter(current,item);
  if (current != null) { // just fix the tail pointer problems
    if (current.next.next == null)
      tail = current.next;
  } else {
    if (tail == null)
      tail = head;
  }
}
```

- Note: This method overrides the one in ExtendedLinkedList but invokes its superclass' insertAfter() method.

TailedLinkedList – Using Generics

```
// Method 1: do all the work yourself

public void insertAfter (ListNode <T>
     current,T item) {
  ListNode <T> temp;

  if (current != null) {
    temp = new ListNode <T> (item,
      current.next);
    current.next = temp;
    if (temp.next == null) tail = temp;
    num_nodes++;
  } else {
    // If current is null, insert item at beginning.
    head = new ListNode <T> (item, head);
    if (tail == null) tail = head;
    num_nodes++;
  }
}
```

```
// Method 2: call superclass

public void insertAfter (ListNode <T> current,
     T item) {
        super.insertAfter (current, item);

        if (current != null) {
            if (current.next.next == null)
                tail = current.next;
        } else {
            if (tail == null)
                tail = head;
        }
    }
}
```

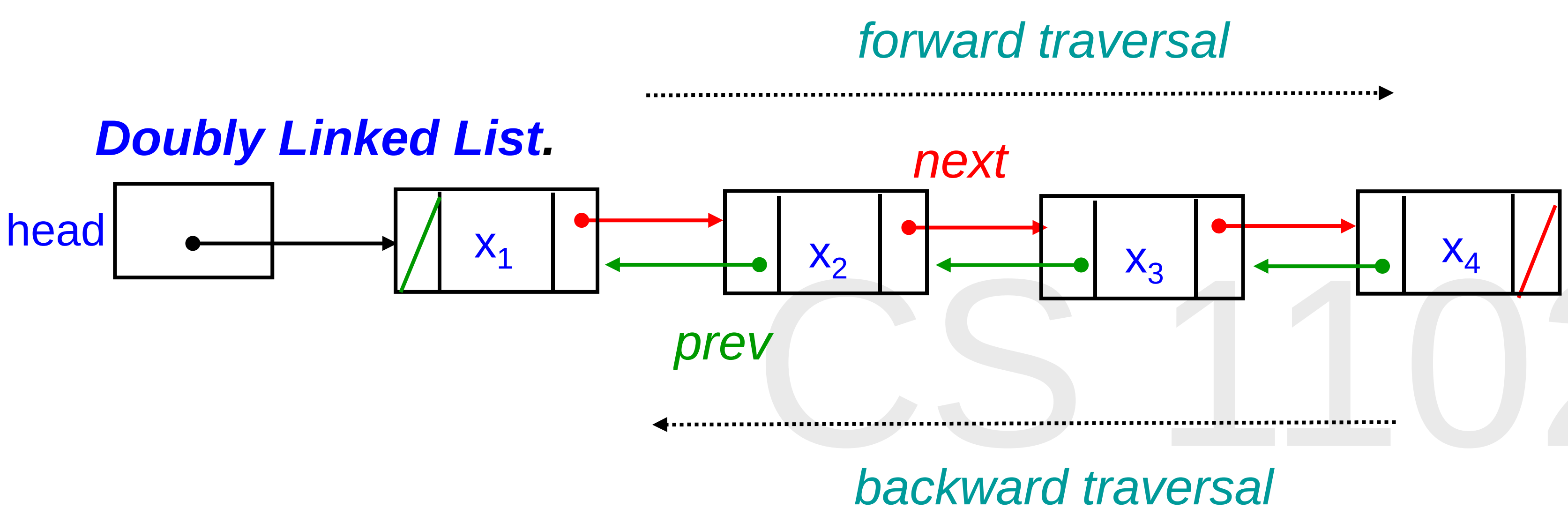
TailedLinkedList – Using Generic Java

```
class TailedLinkedList <T> extends ExtendedLinkedList <T> {
  protected ListNode <T> tail = null;
  public void addTail (T o) {
    if (tail != null) {
      tail.next = new ListNode <T> (o);
      tail = tail.next;
    } else {
      tail = new ListNode <T> (o);
      head = tail;
    }
  }
  public void addHead (T o) {
    super.addHead (o);
    if (head.next == null)
      tail = head;
  }
```

2. Doubly Linked Lists

- Motivation: Frequently, we need to traverse a sequence in BOTH directions efficiently

- Solution : Use doubly-linked list where each node has two pointers
  - Need to modify the ListNode structure.

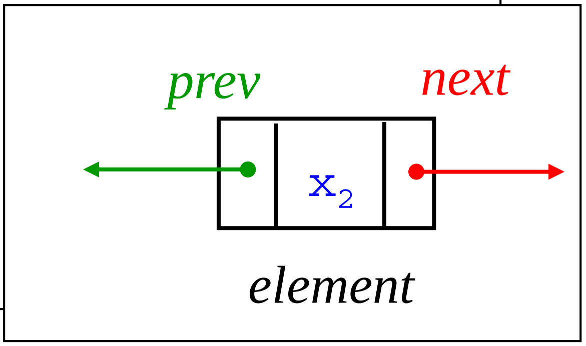


DListNode – A subclass of ListNode

- Each doubly-linked node needs two pointers

- Class declaration for the node: (using generics)

```
class DListNode <E> extends ListNode <E> {
  DListNode <E> prev;

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

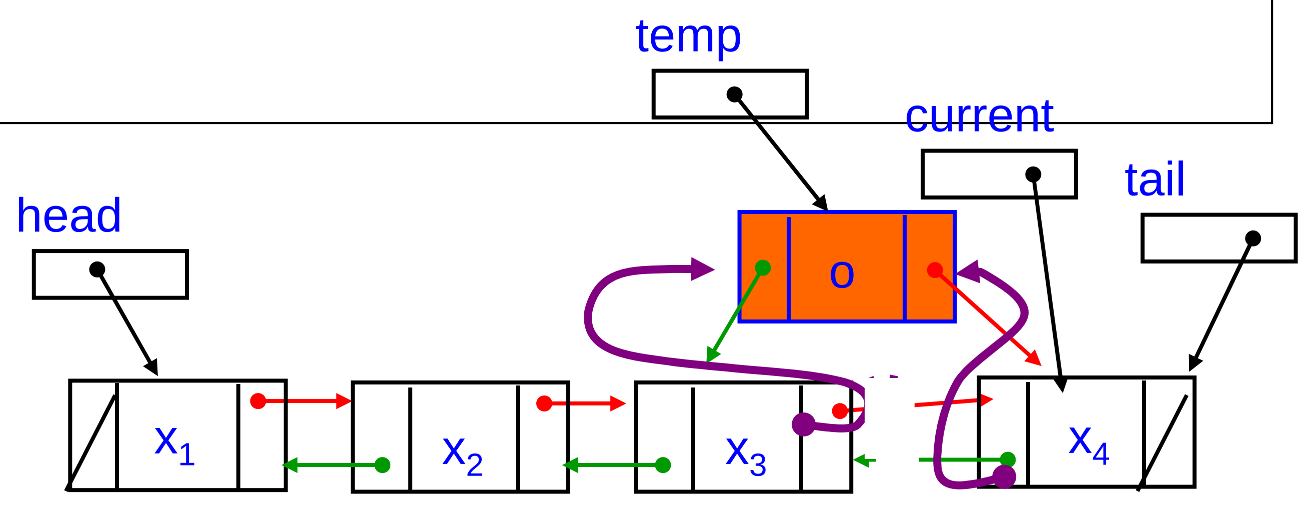


New Methods on Doubly-Linked Lists

```
class DoublyLinkedList <E> extends TailedLinkedList <E> {
    …
    public void insertBefore (DListNode <E> current, E item)
    { … }

    public void deleteCurrent (DListNode <E> current)
    { … }

    // Also need to modify the insertAfter and deleteAfter
    // methods accordingly.
    // DListNode must now be created instead of ListNode.
}
```

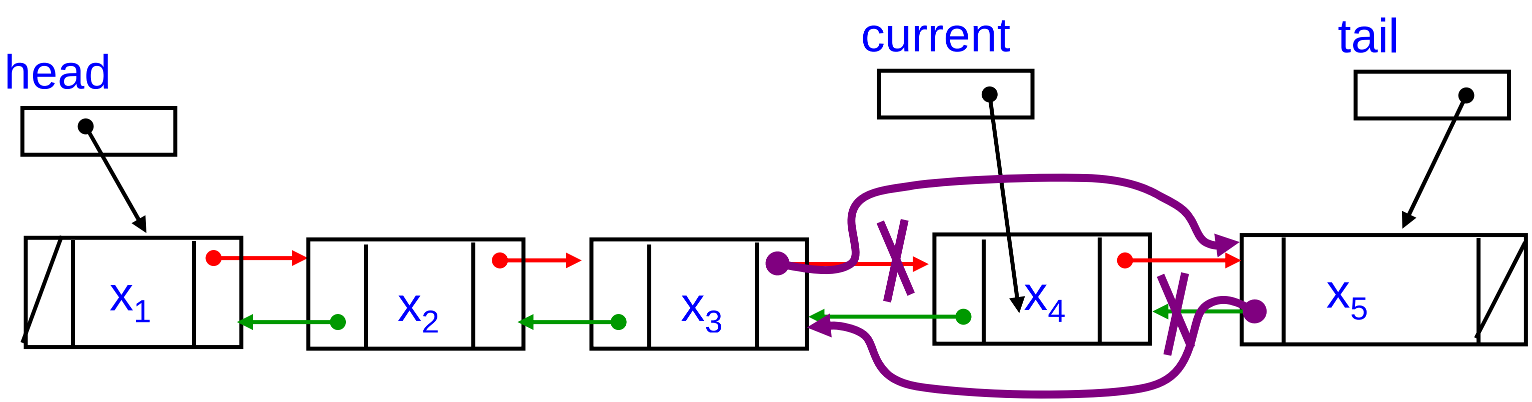
Try to code these functionalities on your own!

```
public void insertBefore (DListNode <E> current, E o)
    throws ItemNotFoundException {

    if (current != null) {
        DListNode <E> temp = new DListNode <E> (o,current,current.prev);
        num_nodes++;

        if (current != head) {
            current.prev.next = temp;
            current.prev = temp;
        } else { // Insert node before head
            current.prev = temp;
            head = temp;
        }
    }
    else { // If current is null, insertion fails.
        throw new ItemNotFoundException("insert fails");
    }
}
```



```
private void deleteCurrent(DListNode <E> current) {
   if (current != tail) {
      DListNode <E> temp = (DListNode <E>) current.next;
      temp.prev = current.prev; num_nodes--;
   } else {                        // was the tail
      tail = current.prev;
      tail.next = null; num_nodes--;
   }

   if (current != head)
      current.prev.next = current.next;
   else {                          // was the head
      head = current.next;
      DListNode temp = (DListNode <E>) temp;
      temp.prev = null;
   }
}
```

Code given doesn't handle one case correctly (Hint: a very small list size) Can you identify it and fix?

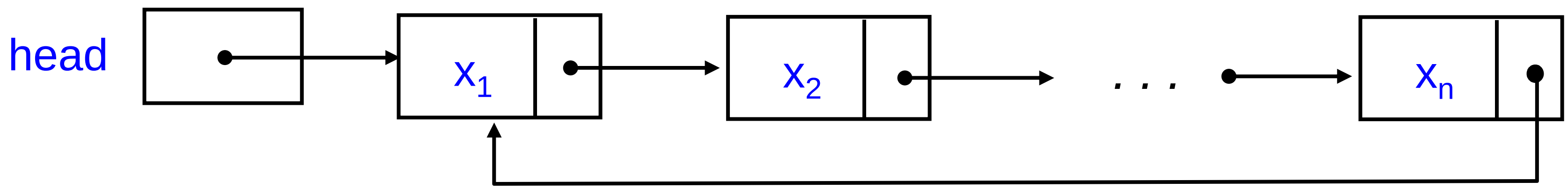


3. Circularly Linked Lists

- Motivation: may need to cycle through a list repeatedly, e.g. round-robin system for allocating a shared resource

- Solution: Have the last node point to the first!

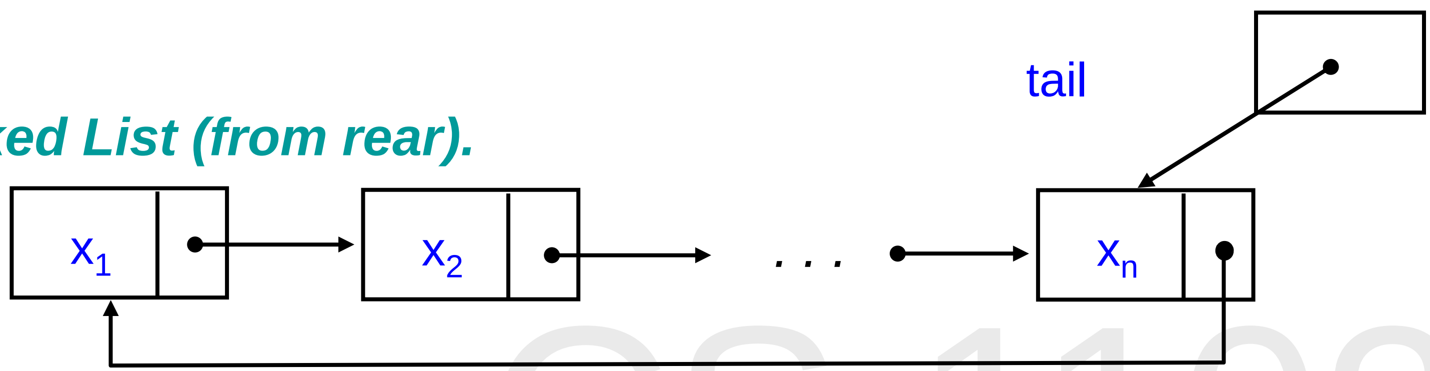
Circular Linked List



Circular Linked List With Tail Pointer

- For singly circular linked-list, it may be better to have a pointer from the rear
- **Q**: Why?

Circular Linked List (from rear).



4. Class LinkedList <E> using Generics

- Linked list implementation of the List interface.
- Implements all optional List operations and permits all elements (including null).
- In addition, the LinkedList class provides uniformly named methods to get, remove and insert an element at the beginning and end of the list – useful for operations in stacks, queues, or deques.

Learn these next week

Method Summary

API for Class LinkedList

boolean	**add**(E o) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(Collection<? extends E> 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<? extends E> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	**addFirst**(E o) Inserts the given element at the beginning of this list.
void	**addLast**(E o) Appends the given element to the end of this list.
void	**clear**() Removes all of the elements from this list.
Object	**clone**() Returns a shallow copy of this `LinkedList`.
boolean	**contains**(Object o) Returns `true` if this list contains the specified element.

E	**element**() Retrieves, but does not remove, the head (first element) of this list.
E	**get**(int index) Returns the element at the specified position in this list.
E	**getFirst**() Returns the first element in this list.
E	**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<E>	**listIterator**(int index) Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
boolean	**offer**(E o) Adds the specified element as the tail (last element) of this list.
E	**peek**() Retrieves, but does not remove, the head (first element) of this list.
E	**poll**() Retrieves and removes the head (first element) of this list.

E	**remove**() Retrieves and removes the head (first element) of this list.
E	**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.
E	**removeFirst**() Removes and returns the first element from this list.
E	**removeLast**() Removes and returns the last element from this list.
E	**set**(int index, E 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.
<T> T[]	**toArray**(T[] 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.

Example: LinkedList <E>

```
import java.util.*;
class LinkedListDriver {
  static void printList (LinkedList <?> alist) {
    System.out.print ("List is: ");
    for (int i = 0; i < alist.size (); i++)
      System.out.print (alist.get (i) + "\t");
    System.out.println ();
  }
  static void printList1 (LinkedList <?> alist) {
    System.out.print ("List is: ");
    while (alist.size () != 0) {
      System.out.print (alist.element () + "\t");
      alist.removeFirst ();
    }
    System.out.println ();
  }
  // … more on next slide …
```

Example: LinkedList <E> (cont.)

```
// continued from first slide

 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);

   LinkedList <Integer> cloneList = (LinkedList <Integer>) alist.clone ();
   // return type is Object for clone method, need cast
   printList1 (cloneList);                      // Q: Will cloneList be empty after the call?
   System.out.println ("First element - " + alist.getFirst());          // A. Yes
   System.out.println ("Last element - "  + alist.getLast ());          // B. No
   alist.addFirst (888);
   alist.addLast (999);
   printList (alist);
   printList (cloneList);
 }
```

```
List is: 1    2    3    4    5
List is: 1    2    3    4    5
First element - 1
Last element - 5
List is: 888    1    2    3    4    5
999
List is:
```

Summary

- This week we discussed the **list ADT**:
  - Its implementation via arrays
  - Its implementation via linked lists
  - When to choose which implementation
- Linked Lists ADT
  - Variations of Linked Lists
  - Class LinkedList <E> in generic Java