CS1102: Data Structures and Algorithms

Part 3

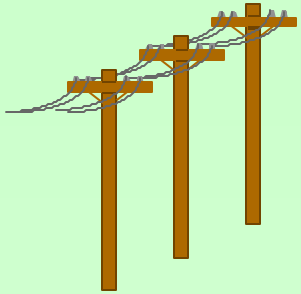
Linked List

Zoltan KATO

S16 06-12

Adopted from Chin Wei Ngan's cs1102 lecture notes

Linked-Lists

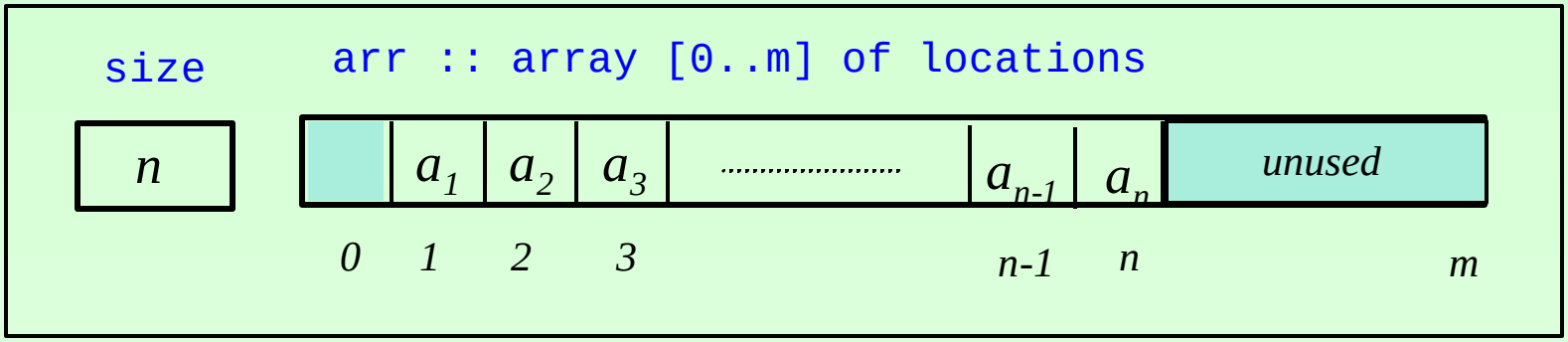


Lists via Arrays

Lists are very pervasive in computing,
e.g. class list, list of chars, list of events

One very simple implementation is to use Java arrays

A sequence of n-elements.



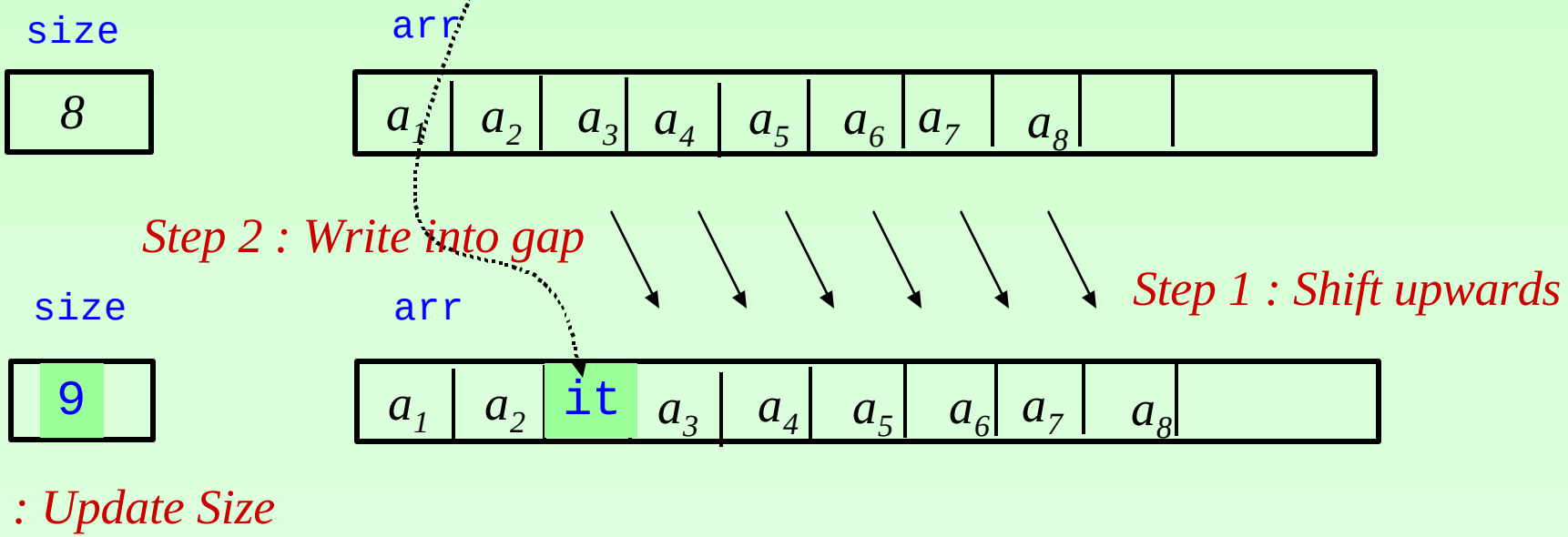
Note: In Java, arrays start from position 0, but here we use it starting from position 1. Position 0 is left unused.

Third Element : `arr[3]`

Inserting into an Array

While retrieval is very fast, insertion and deletion are slow.
Insert has to shift upwards to create gap

Example : `insert(3,it)`



Step 3 : Update Size

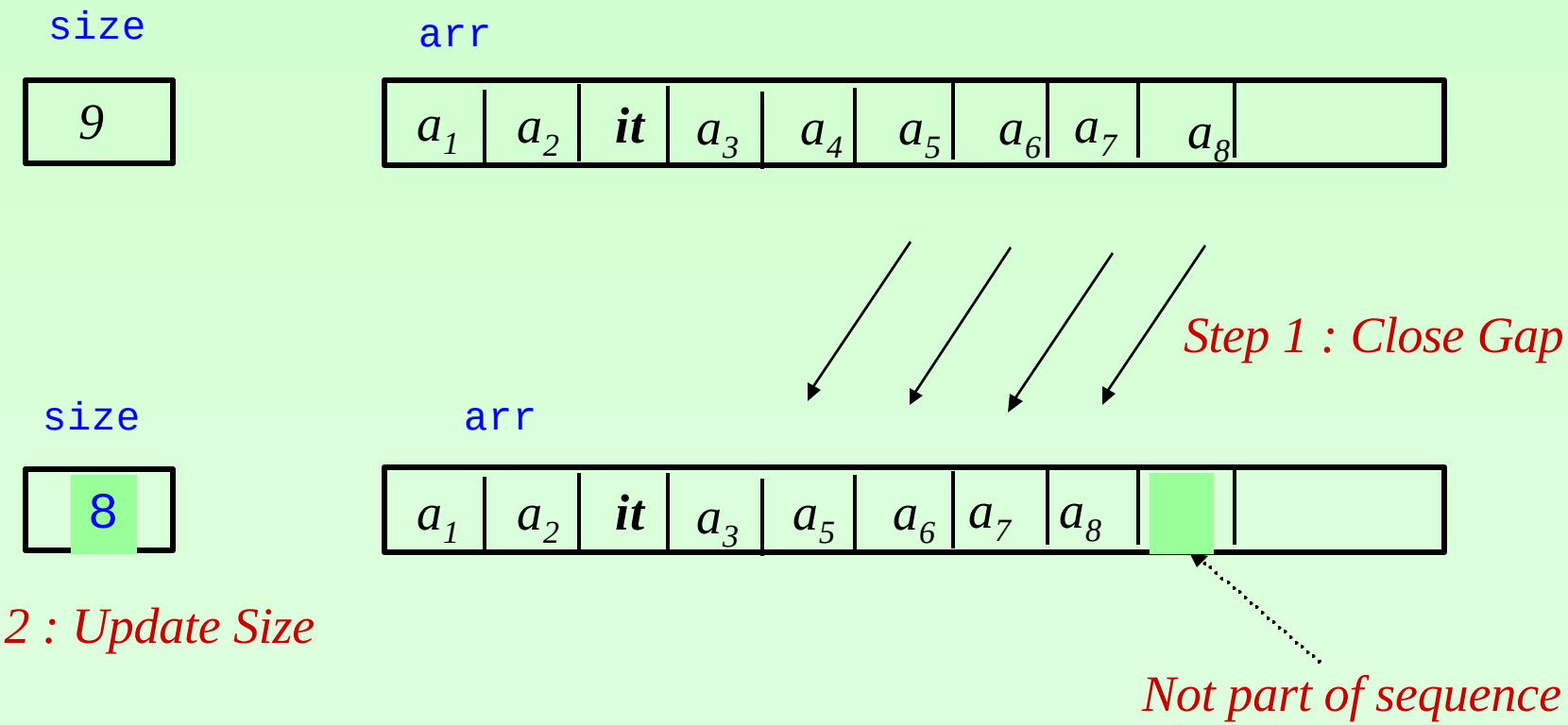
Coding — Inserting into an Array

```
class list {
   private int size
   private Object[] arr;

   public void insert(int j, Object it)
      { // pre : 1<=j<=size+1

        for (i=size; i>=j; i=i-1)
          { arr[i+1]=arr[i]; }; // Step 1 : Create gap

        arr[j]=it;              // Step 2 : Write to gap

        size = size + 1;        // Step 3 : Update size
      }
```

Deleting from an Array

Delete has to shift downwards to close gap of deleted item,

Example: `delete(5)`



Coding — Deleting from an Array

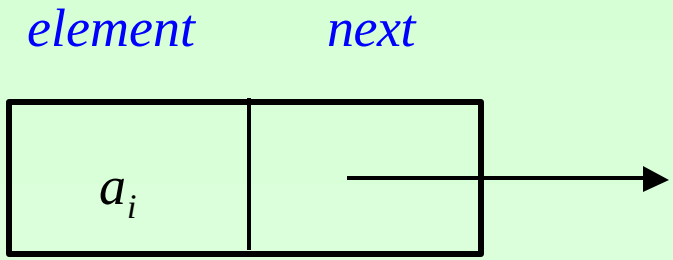
```
public void delete(int j)
{ // pre : 1<=j<=size

  for (i=j+1; i<=size; i=i+1)
    { arr[i-i]=arr[i]; };     // Step 1: Close gap

  size = size - 1;            // Step 2: Update size
}
```

Linked-List Approach

Main problem of array is deletion/insertion slow since it has to shift items in its *contiguous* memory.

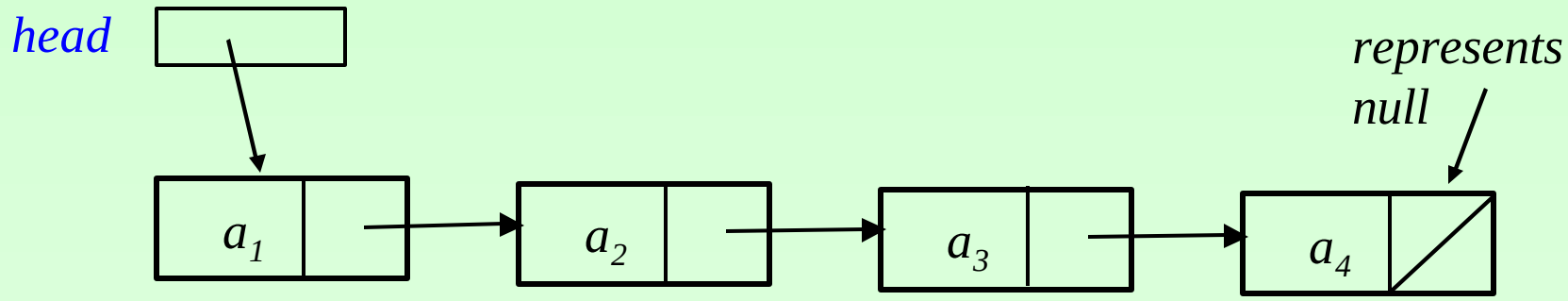
Solution : linked list where items need *not be contiguous* with nodes of the form:

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

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



Sample Linked-List Approach

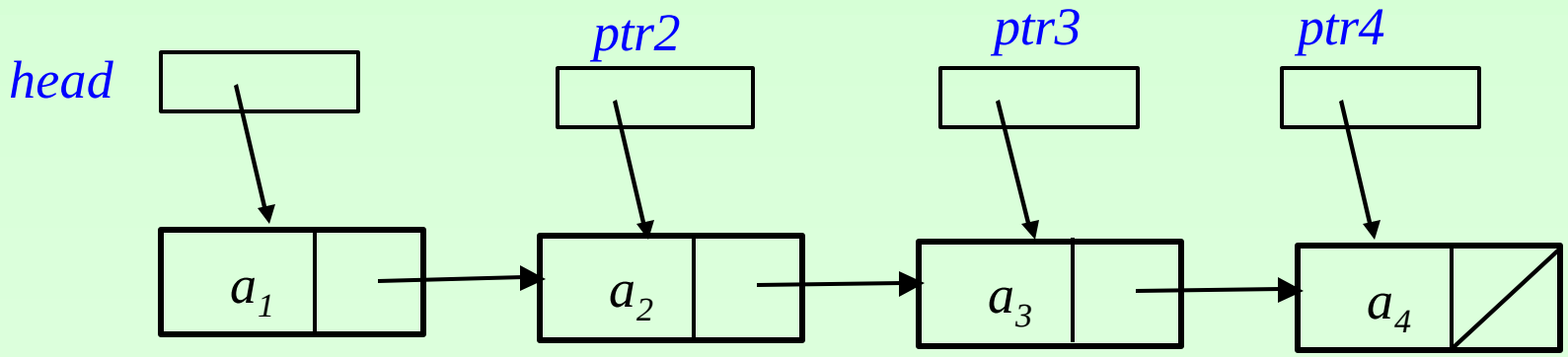
Sequence of four items < a_1,a_2 ,a_3 ,a_4 > can be represented by:



Sample Linked-List Approach

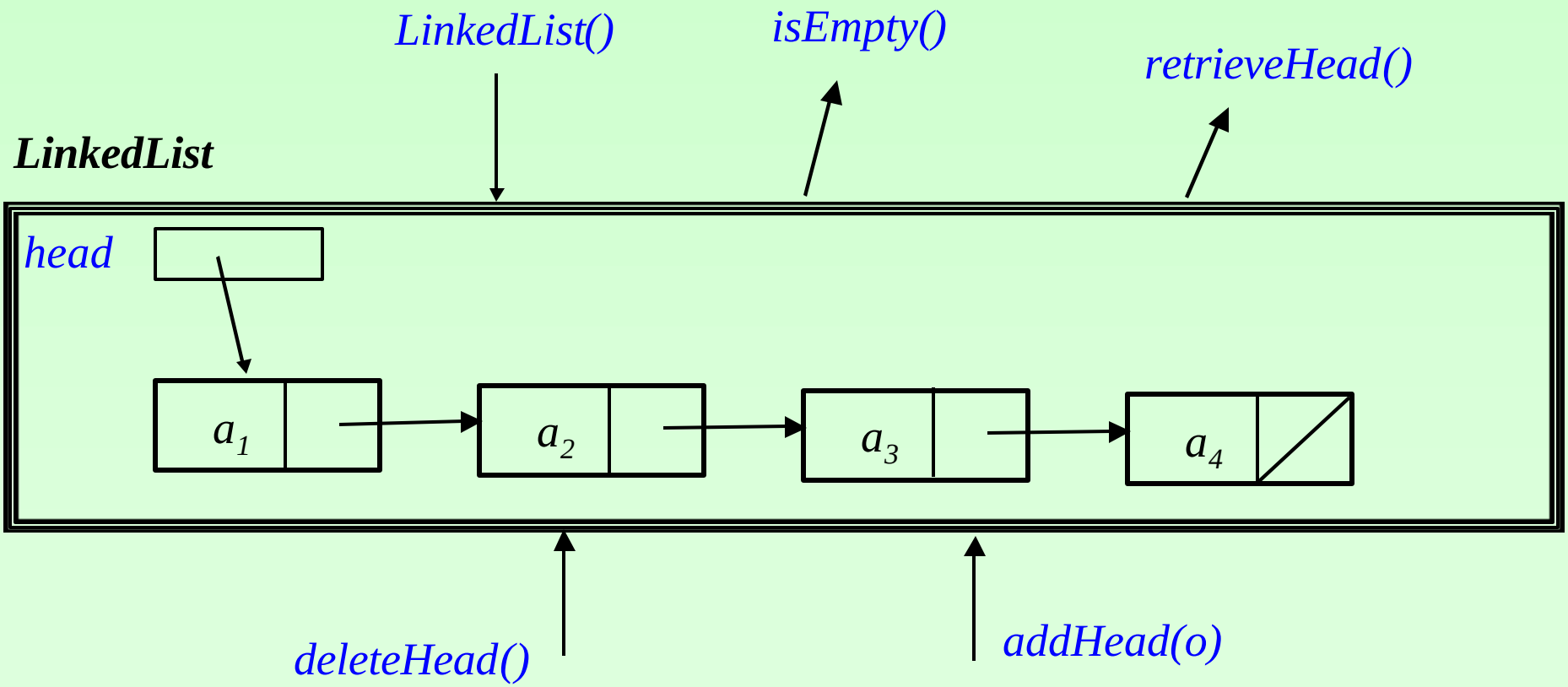
The earlier sequence can be built by:

```
ListNode ptr4 = new ListNode("a4",null);
ListNode ptr3 = new ListNode("a3",ptr4);
ListNode ptr2 = new ListNode("a2",ptr3);
ListNode head = new ListNode("a1",ptr2);
```



Linked-List ADT

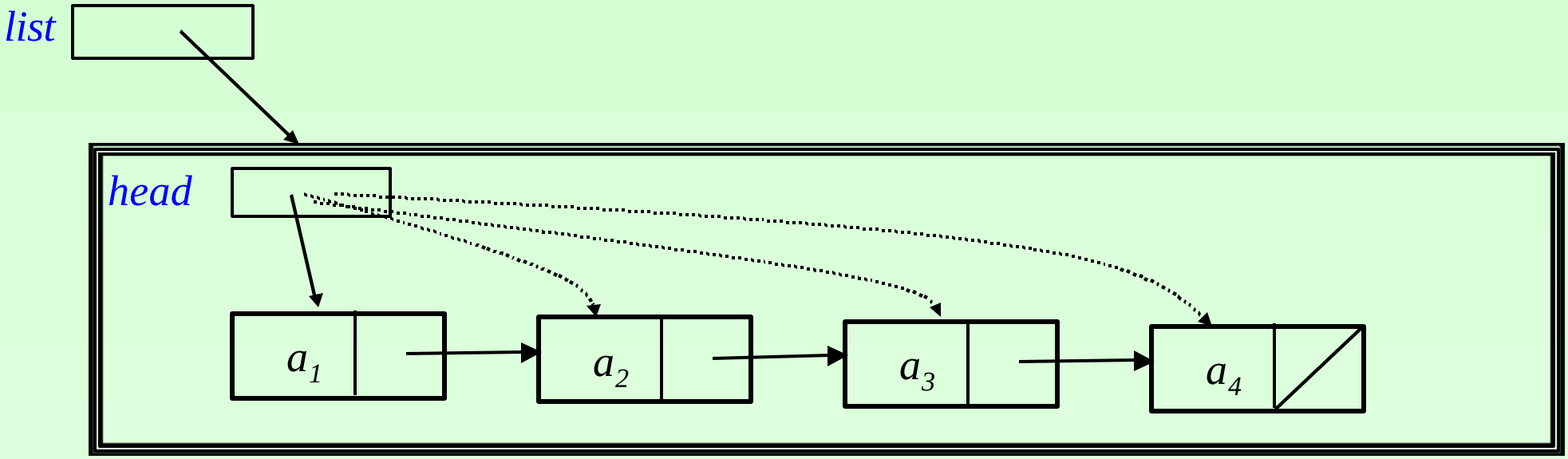
We can provide an ADT for linked-list.
This can help hide unnecessary internal details.



Sample Linked-List ADT

Sequence of four items < a_1,a_2 ,a_3 ,a_4 > can be built, as follows:

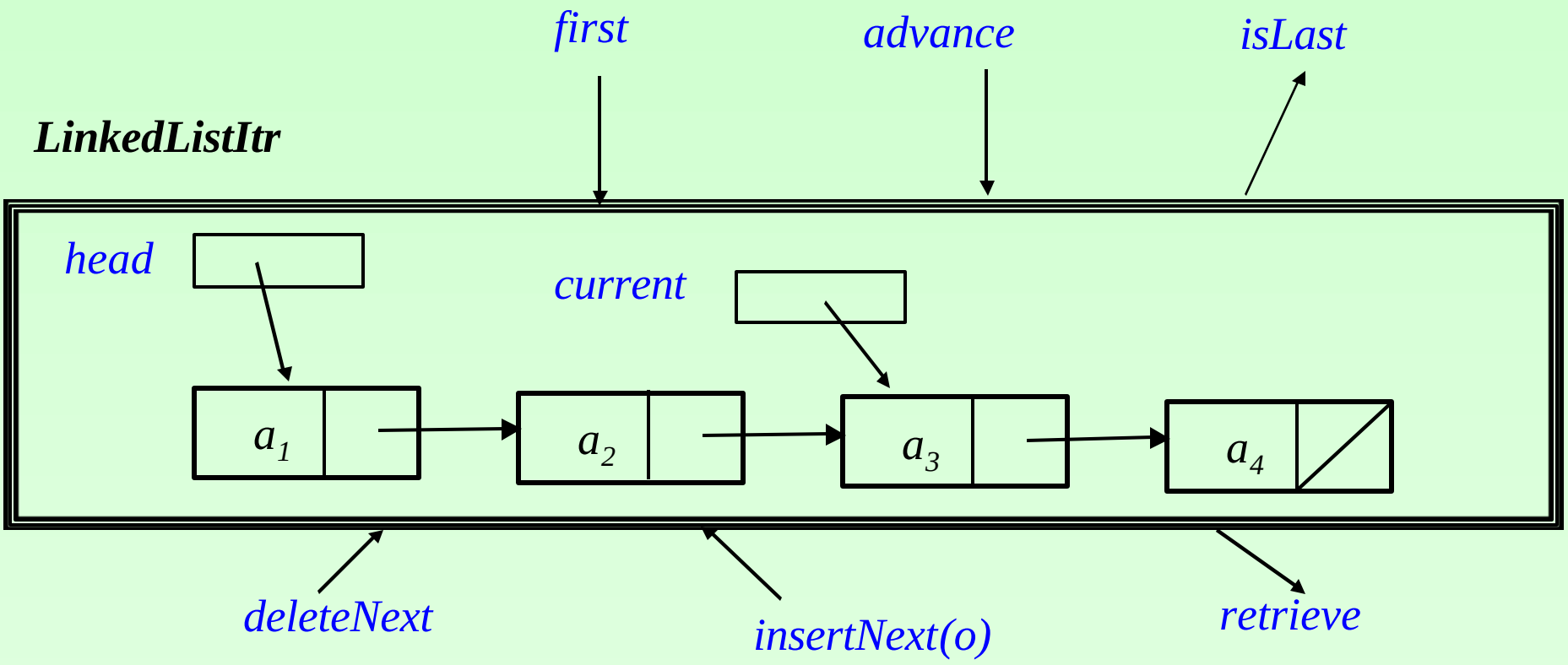
```
LinkedList list = new LinkedList();
list.addHead("a4");
list.addHead("a3");
list.addHead("a2");
list.addHead("a1");
```



```
class LinkedList {
  protected ListNode head ;

  public LinkedList() {head = null; }
  public boolean isEmpty()
      {return (head == null); }
  public void addHead(Object o)
      {head = new ListNode(o,head); }
  public void deleteHead() throws ItemNotFound
      {if (head ==null)
           {throw new ItemNotFound("DeleteHead fails");}
       else head = head.next;}
  };

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

Linked-List Iteration ADT

To support better access to our linked-list, we propose to build a linked-list iteration ADT.



Declaration — Linked-List Iteration ADT

```
class LinkedListItr extends LinkedList{

   private ListNode current;
   private boolean zeroflag;

   public LinkedListItr() {… }
   public void zeroth() { … }
   public void first() { … }
   public void advance() { … }
   public boolean isLast() { … }
   public boolean isInList() { … }

   public Object retrieve() { … }
   public void insertNext(Object o) { … }
   public void deleteNext() { … }
}
```

data structure

access or change current pointer

access or change nodes

```
public void zeroth()      // set position prior to first element
      { current = null;
        zeroflag = true;
      }
```

Why zeroflag? – To distinguish "zeroth position" from "beyond list position".

Zeroth Position

```
(current == null) && (zeroflag==true)
```

InList

```
(current != null)
```

Beyond List

```
(current == null) && (zeroflag==false)
```

```
public void first() throws ItemNotFound
      // set position to first element
      { current = head;
        zeroflag = false;
        if (current == null)
            {throw new ItemNotFound("No first element");};
      }

public void advance()   // advance to next item
    {if (current != null) {current = current.next }
     else {if (zeroflag) {current = head;
                          zeroflag = false;}
           else {}
          };
    }
```

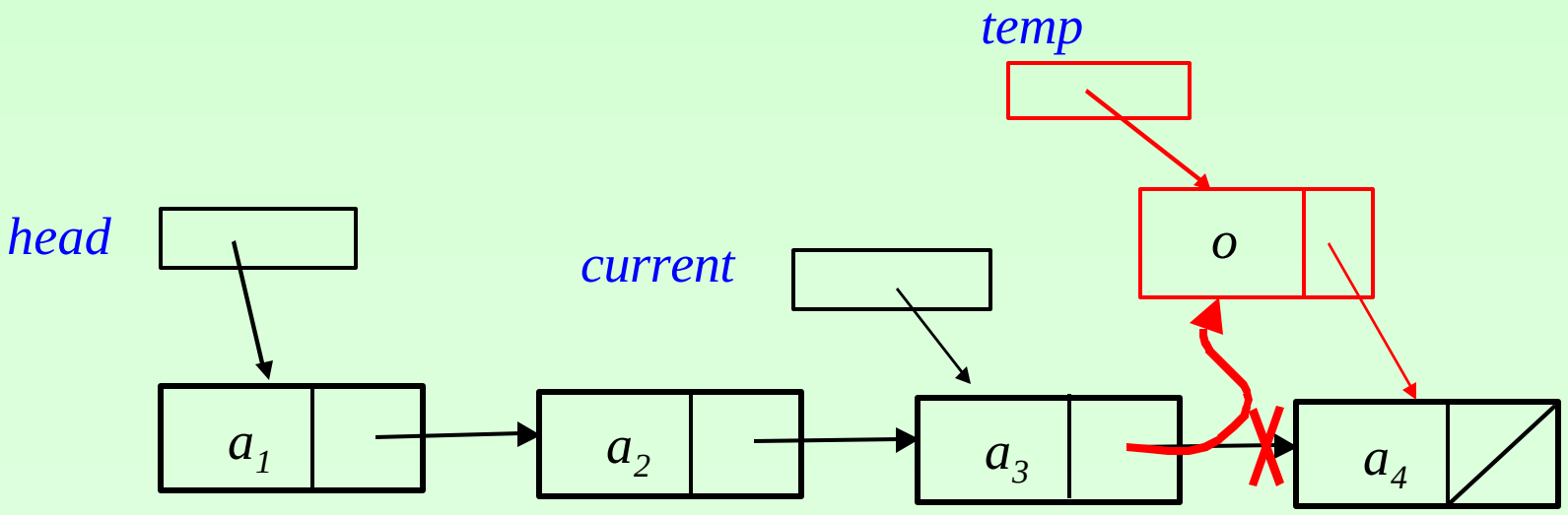
Coding Linked-List Iteration ADT

```
public boolean isLast() // check if it current is at the last node
      {if (current!=null) return (current.next == null);
       else return false;
      }

public boolean isInList() // check if it current is at some node
      (return (current != null);
      }

public Object retrieve() throws ItemNotFound
                // current object at current position
      { if (current!=null) { return current.element; }
        else { throw new ItemNotFound("retrieve fails"); };
      }
```

Insertion

Linked-List Iteration ADT

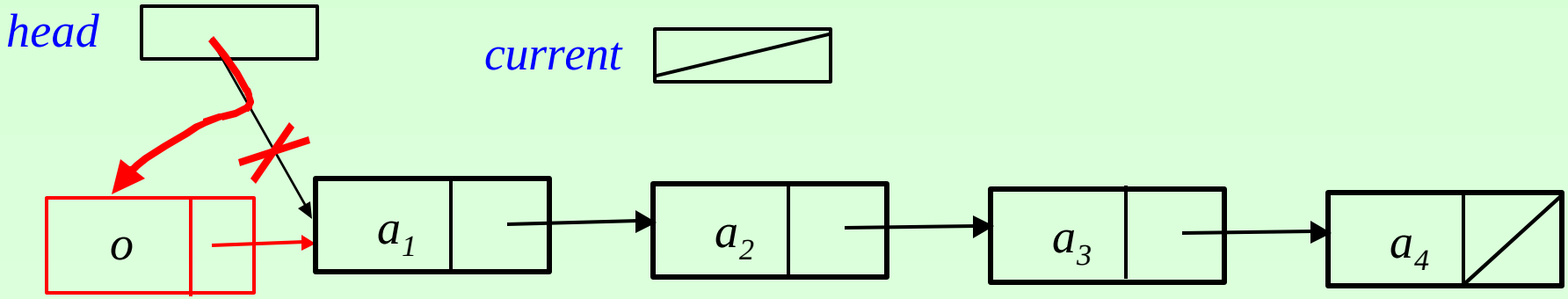
```
public void insertNext(Object o) throws ItemNotFound
                    // insert after current position
   { ListNode temp;
     if (current!=null) {temp = new ListNode(o,current.next);
                         current.next = temp;}
     else if (zeroflag) { head = new ListNode(o,head); }
          else { throw new ItemNotFound("insert fails"); };
   }
```



Insertion

Linked-List Iteration ADT

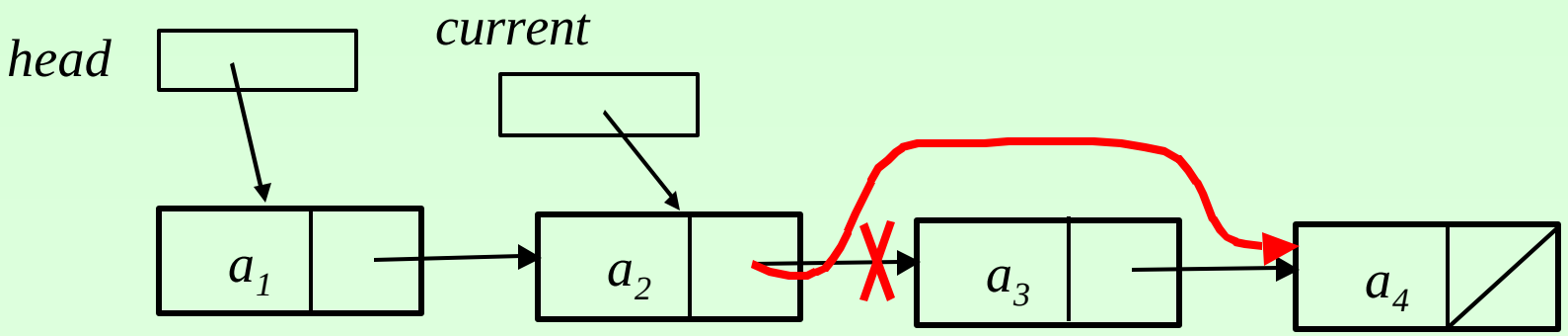
```
public void insertNext(Object o) throws ItemNotFound
                                  // insert after current position
   { ListNode temp;
     if (current!=null) {temp = new ListNode(o,current.next);
                          current.next = temp;}
     else if (zeroflag) { head = new ListNode(o,head); }
          else { throw new ItemNotFound("insert fails"); };
   }
```



Deletion

Linked-List Iteration ADT

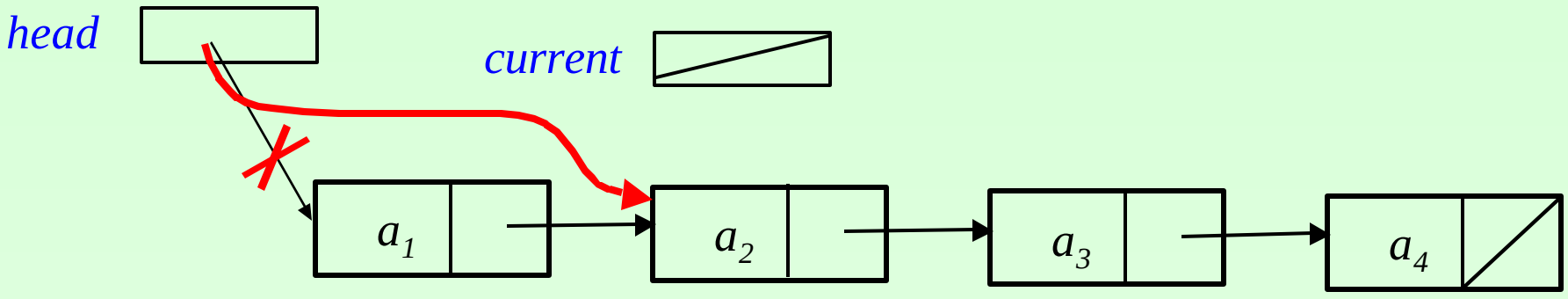
```
public void deleteNext() throws ItemNotFound
                        // delete node after current position
{ if (current!=null)
       {if (current.next!=null)
             {current.next = current.next.next;}
        else {throw new ItemNotFound("No Next Node to Delete");};
       }
  else if (zeroflag && head!=null) {head=head.next;}
       else {throw new ItemNotFound("No Next Node to Delete");};
 }
```



Deletion

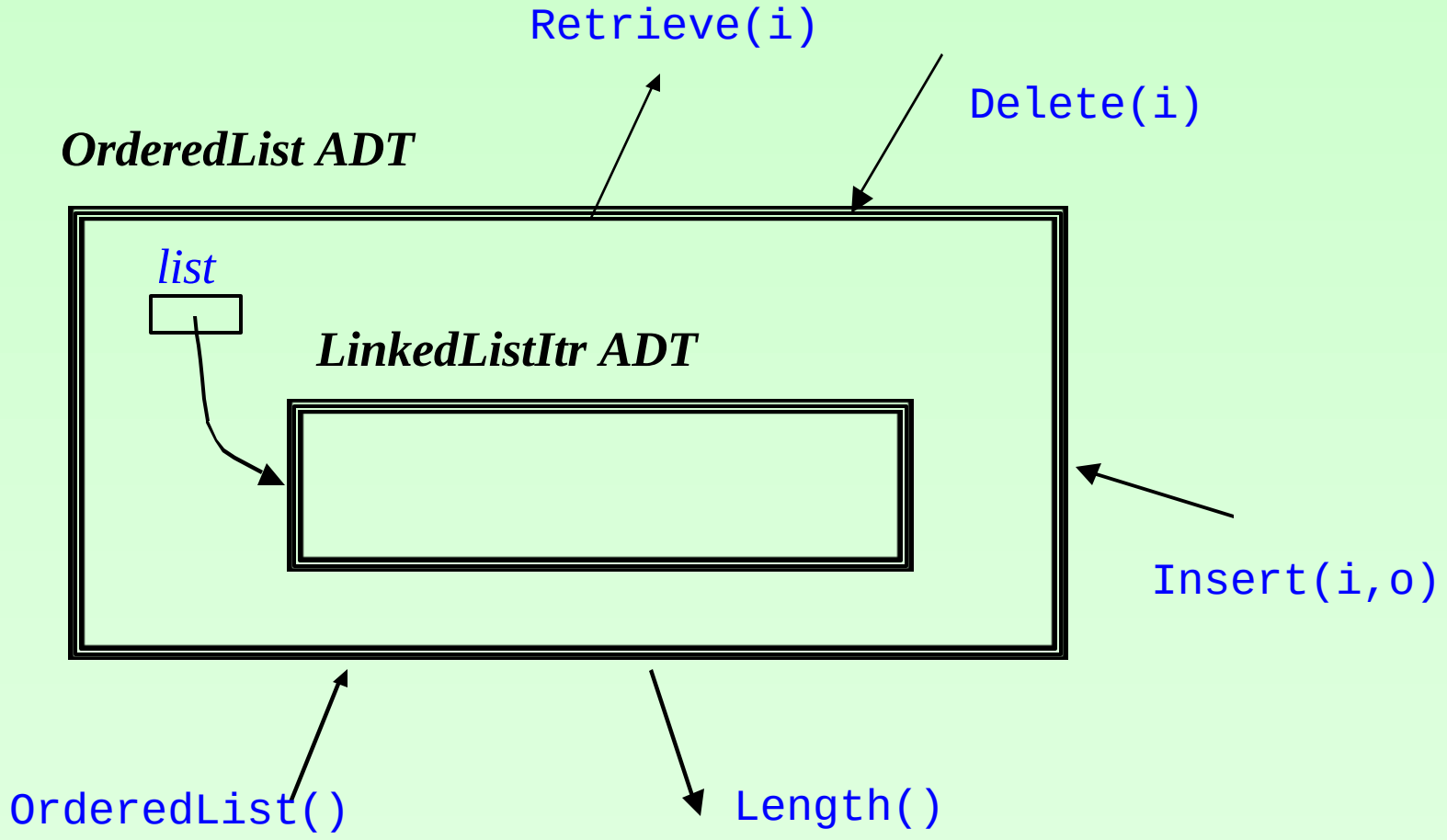
Linked-List Iteration ADT

```
public void deleteNext() throws ItemNotFound
                        // delete node after current position
{ if (current!=null)
        {if (current.next!=null)
              {current.next = current.next.next;}
         else {throw new ItemNotFound("No Next Node to Delete");};
        }
   else if (zeroflag && head!=null) {head=head.next;}
        else {throw new ItemNotFound("No Next Node to Delete");};
 }
```



Ordered List ADT

We can use *LinkedListItr* ADT to construct *OrderedList* ADT.



Implementation — Ordered-List ADT

```
class OrderedList {
 private LinkedListItr list;

 public OrderedList()
       // pre : true
       // post : this = <>
  { list = new LinkedListItr(); }

 public int Length()
       // pre : this = <a1,...,an>
       // post : returns n
   { int cnt = 0;
     try {
     for (list.first(); list.isInList();  list.advance())
         { cnt++; };
     } catch (ItemNotFound e) { cnt=0; };
     return cnt;
   }
```

```
public Object Retrieve(int i) throws ItemNotFound
       // pre : this = < a1,...,an >
       // post : return  ai
       //      : or throws error  if not(1<=i<=n)
 { if (this.setPosn(i))
     { return list.retrieve() }
   else
     { throw new ItemNotFound("retrieve fails"); };
 }
```

`setPosn(i)` is a method to move the current pointer to the `i`th position of `list`. It returns false if the pointer goes beyond the end of the list.

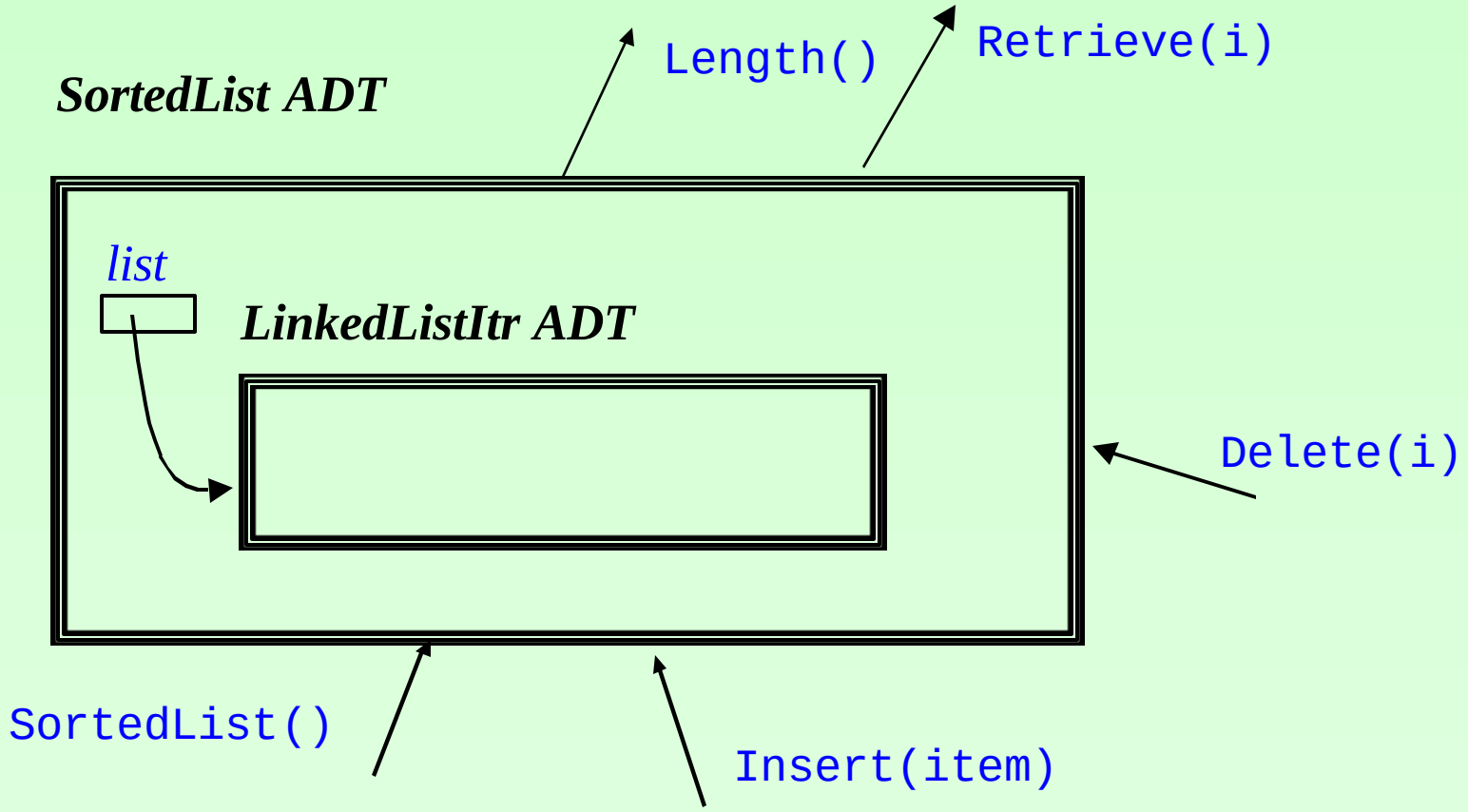
```
private boolean setPosn(int i)
 { // pre : i >= 0
   list.zeroth();
   for (j=1; j<=i; j++)
     {list.advance();
      if !(list.isInList()) return false;
     }
   return true;
 }
```

Insertion/Deletion — Ordered-List ADT

```
public void Insert(int i, Object o) throws ItemNotFound
       // pre : this = < a1,...,an >
       // post : this = < a1,..,ai-1,o,ai,..,an >
       //      : or throws error  if not(1<=i<=n+1)
 { if (this.setPosn(i-1))
     { list.insertNext(o); }
   else
     { throw new ItemNotFound("insert fails – bad position"); }
 }

public void Delete(int i) throws ItemNotFound
       // pre : this = < a1,...,an >
       // post : this = < a1,..,ai-1,ai+1,..,an >
       //      : or throws error  if not(1<=i<=n)
 { if (this.setPosn(i-1))
     { list.deleteNext(); }
   else
     { throw new ItemNotFound("delete fails – no item"); }
 }
```

Sorted List ADT

We can also use *LinkedListItr* ADT to construct *SortedList* ADT.



The operations to **`retrieve`** *and* **`delete`** *the ith item are similar to those given for* ***OrderedList*** *ADT.*

The **`insert`** *operation for* ***SortedList*** *ADT requires a lookahead capability. This requires* ***LinkedList Iteration*** *ADT to be supplied with an extra operation, called* **`retrieveNext.`**

```
public Object retrieveNext() throws ItemNotFound
            // retrieve object at next position
  { if (current!=null)
            { if (current.next!=null)
                    { return current.next.element; }
              else { throw new ItemNotFound("retrieve fails");};}
    else if (zeroflag && head!=null)
                    {return head.element;}
         else { throw new ItemNotFound("retrieve fails");};
  }
```

Linked-List Implementation *Sorted-List ADT*

```
class SortedList {
 private LinkedListItr list;
        :
 public void Insert(Comparable item)
       // inserts a new item into a sorted position
 {try { list.zeroth;
        boolean exitflag = false;
        do { Object nextobj = list.retrieveNext();
             if (nextobj.compareTo(item)<0) { list.advance }
             else {exitflag = true};
           }
        while (!(exitflag))
      } catch (ItemNotFound e) { // do nothing
                                 };
  try { list.insertNext(item);}
       catch (Exception e) {// should not be possible
                            };
 }
```

retrieve next object without changes to the current pointer

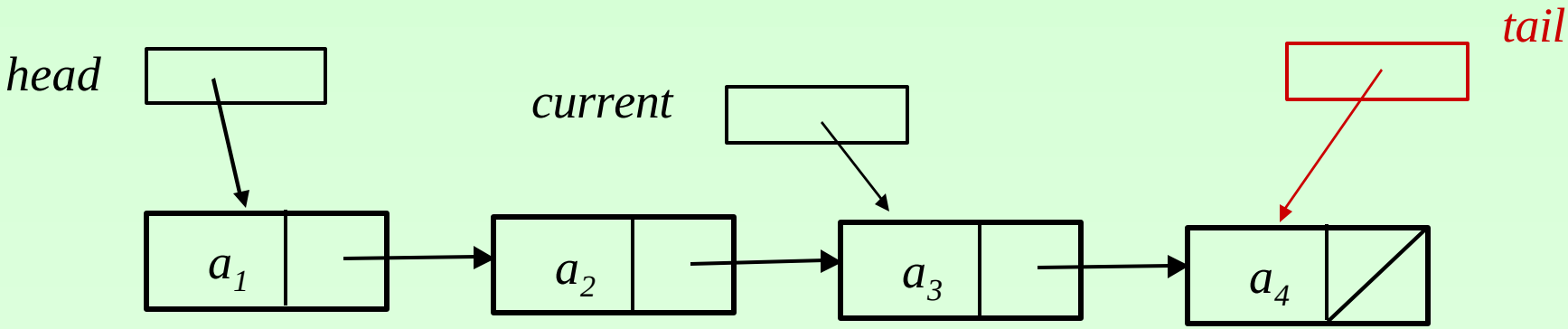
Variations of Linked-Lists

- with tail pointer
- with dummy node
- doubly linked-list
- circular linked-list

With Tail-Pointer

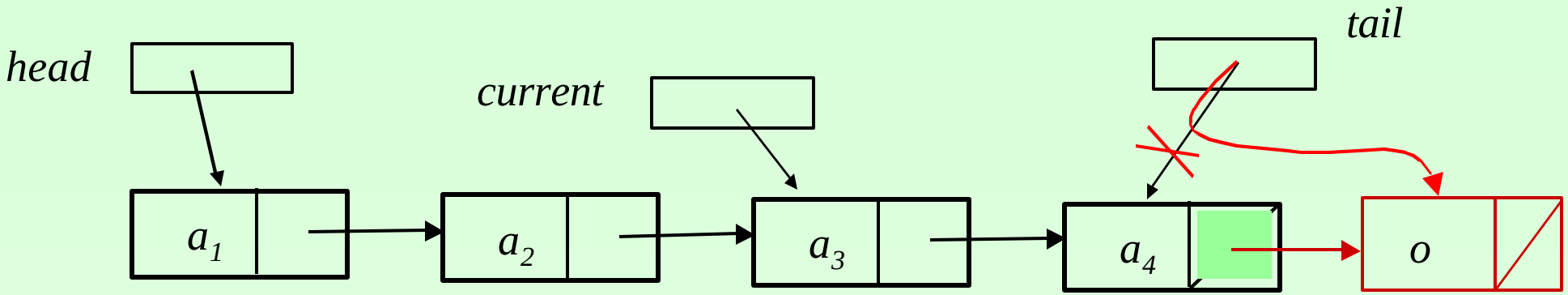
Allows more efficient access (i.e. insertion) at the end of linked list.

Useful for queue-like structures.



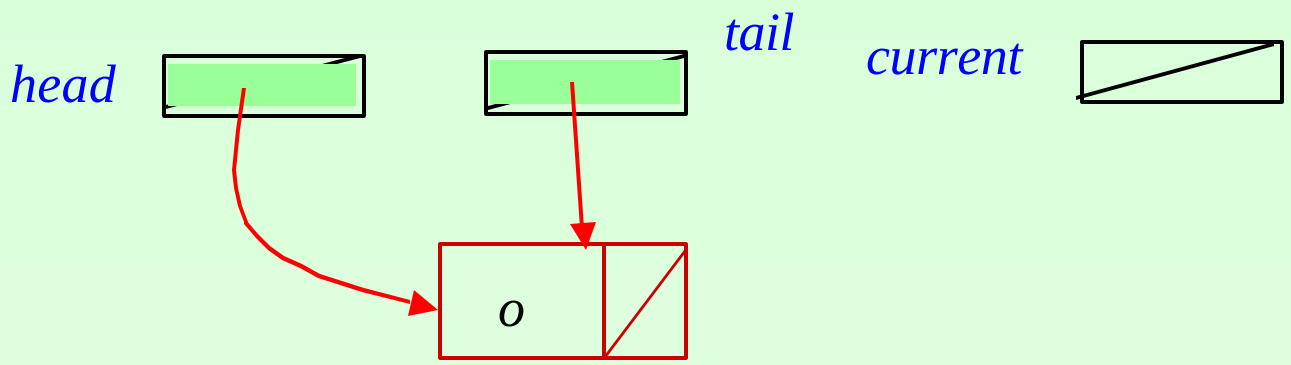
New Function With Tail Pointer

```
class LinkedListItr {
   private ListNode head;
   private ListNode tail;
   private ListNode current;
   private boolean zeroflag;
       :

   public void addTail (Object o) {
        if (tail!=null) {tail.next = new ListNode(o);
                         tail = tail.next;}
        else {tail = new ListNode(o);
              head = tail; };   }
```



New Function With Tail Pointer

```
class LinkedListItr {
   private ListNode head;
   private ListNode tail;
   private ListNode current;
   private boolean iszeroth;
       :

   public void addTail (Object o) {
        if (tail!=null)  {tail.next = new ListNode(o);
                          tail = tail.next;}
        else {tail = new ListNode(o);
              head = tail; };   }
```



Code Changes With Tail Pointer

Functions, such as **zeroth**, **first**, **advance**, **retrieve** which do not change the nodes of the linked-list are unchanged.

However, functions that add or delete nodes may affect the tail-pointers. We must provide *extra code* to cater to this possibility.

```
public void insertNext(Object o) throws ItemNotFound
                // insert after current position
   { ListNode temp;
     if (current!=null) {temp = new ListNode(o,current.next);
                          if (current.next==null) {tail=temp};
                         current.next = temp;}
     else if (zeroflag) { head = new ListNode(o,head);
                          if (head.next==null) {tail=head};
                        }
     else { throw ItemNotFound("insert fails"); }
   }
```

Code Changes With Tail Pointer

```
public void deleteNext() throws ItemNotFound
                // delete node after current position
{ if (current!=null)
       {if current.next!=null
              {current.next = current.next.next;
                if (current.next==null) {tail=current};
               }
        else {throw new ItemNotFound("No Next Node to Delete")}
       }
  else if (zeroflag && head!=null)
              {head=head.next;
                if (head==null) {tail=null};
               }
       else {throw new ItemNotFound("No Next Node to Delete");}
 }
```

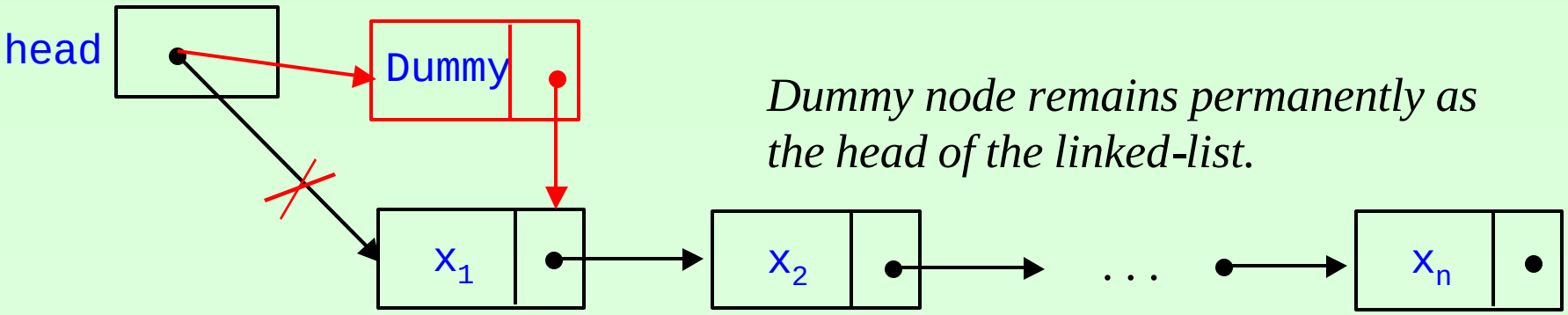
With Dummy Header

Need special codes for

- empty list
- inserting to the front of list

Solution : Use an extra dummy header node to simplify coding.

Linked List with Dummy Node

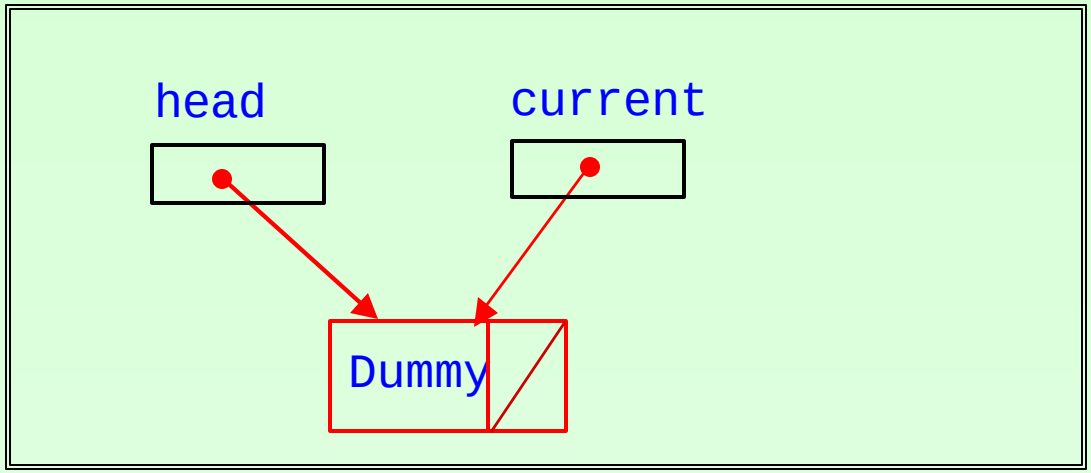


Dummy node remains permanently as the head of the linked-list.

Implementation *With Dummy Node*

```
class LinkedListItr {
   private ListNode head;
   private ListNode current;

   public LinkedListItr()
    {head = new ListNode((Object) “Dummy”,null);
     current = head;}
```

LinkedListItr



Implementation With Dummy Node

```
public void zeroth()    // set position prior first element
 { current = head;
 }

public void first() throws ItemNotFound
                        // set position to first element
 { current = head.next;
   if (current == null)
        throw new ItemNotFound("No first element");
 }
public void advance()          // advance to next item
 {if (current != null) {current = current.next }
  else {if (zeroflag) {current = head;
                       zeroflag = false)
        else {}
       }
 }
```

Simpler Insertion With Dummy Node

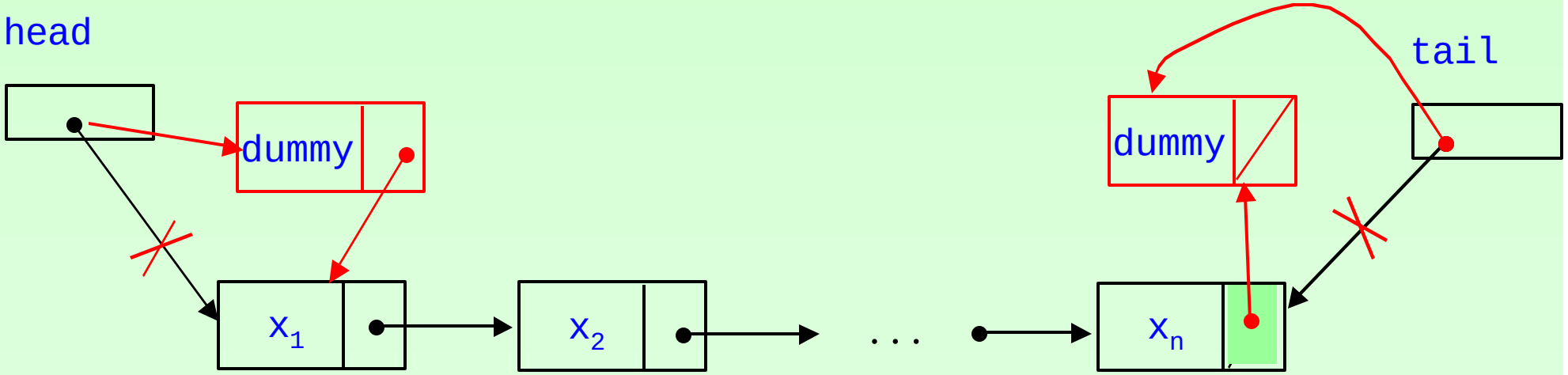
Simpler **`insertNext`** Code *with* Dummy Node

```
public void insertNext(Object o) throws ItemNotFound
                              // insert after current position
   { ListNode temp;
     if (current!=null) {temp = new ListNode(o,current.next)
                          current.next = temp;}
     else if (zeroflag) {head = new ListNode(o,head); }
     else { throw new ItemNotFound("insert fails"); }
   }
```

Trailer Node With Dummy Node

If we have a tail-pointer, we may also wish to have a *dummy trailer node* too!

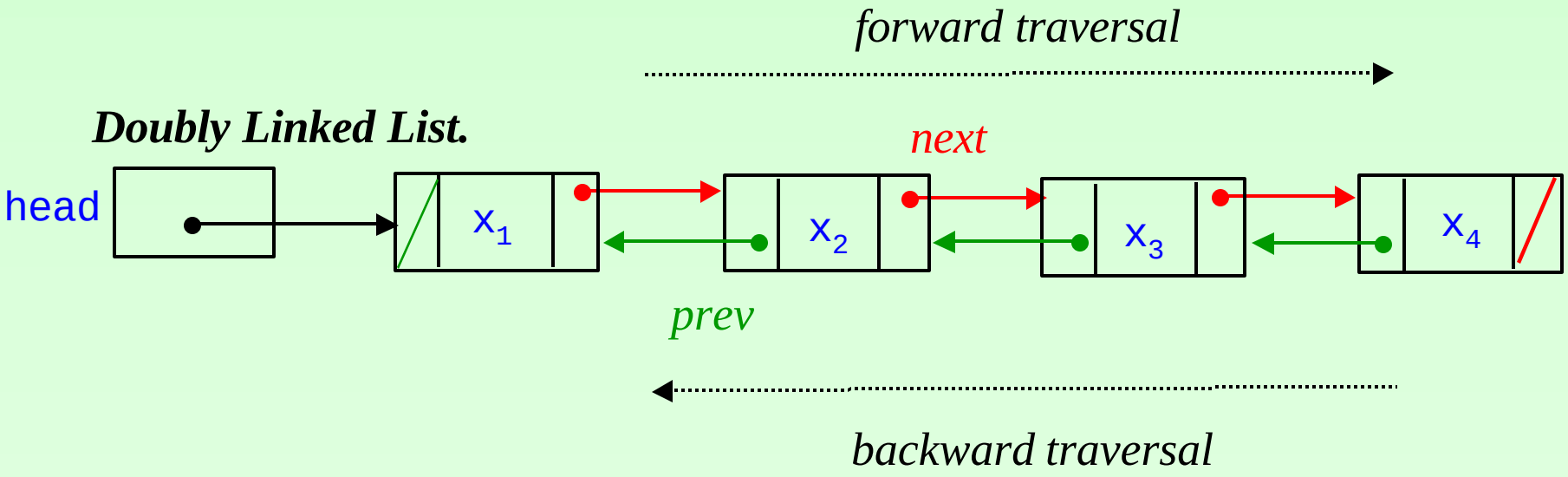
Linked List with Dummy Header & Trailer Node



Doubly Linked-List

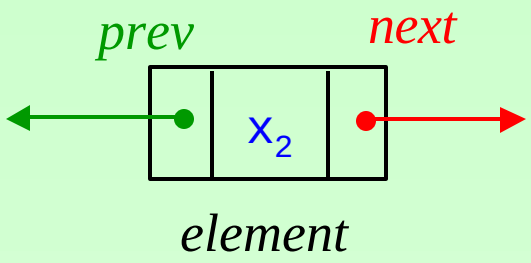
Frequently, we need to traverse a sequence in BOTH directions efficiently.

Solution : Use doubly-linked list where each node has two pointers.



Definition for Node — Doubly Linked-List

Each doubly-linked node has two pointers.



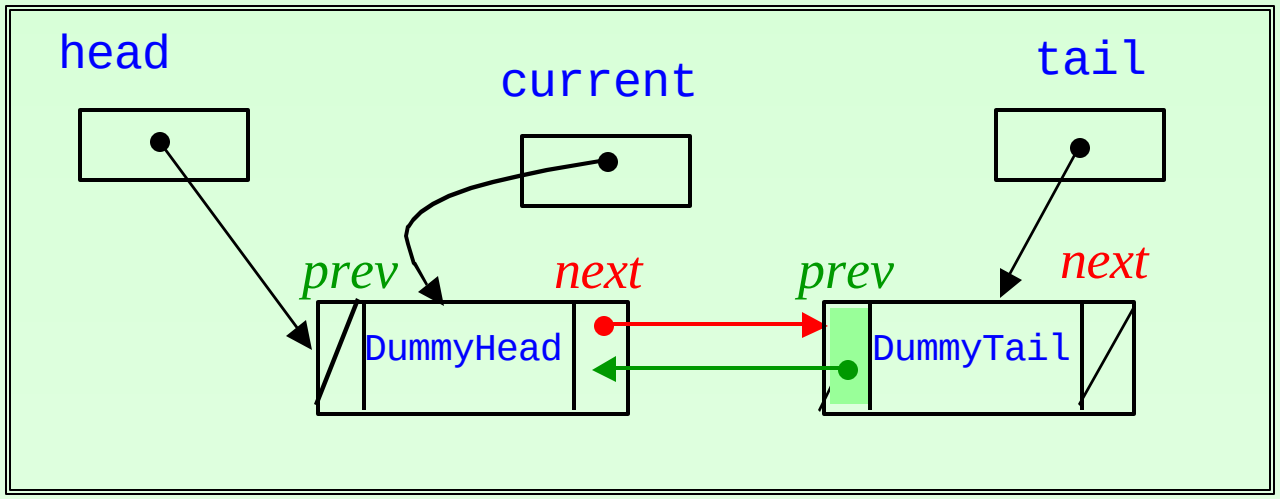
Class declaration for above node:

```
class DListNode {
  Object element;
  DListNode next;
  DListNode prev;

  public DListNode(Object o, DListNode n, DListNode p)
    { element= o; prev=p; next = n; };
  }
```

Implementation — Doubly Linked-List

```
class LinkedListItr {
   private DListNode head;
   private DListNode tail;
   private DListNode current;

   public LinkedListItr()
   {tail = new DListNode("DummyTail",null,null);
    head = new DListNode("DummyHead",tail,null);
    tail.prev = head;
    current = head;}
```

LinkedListItr



Implementation *Doubly Linked-List*

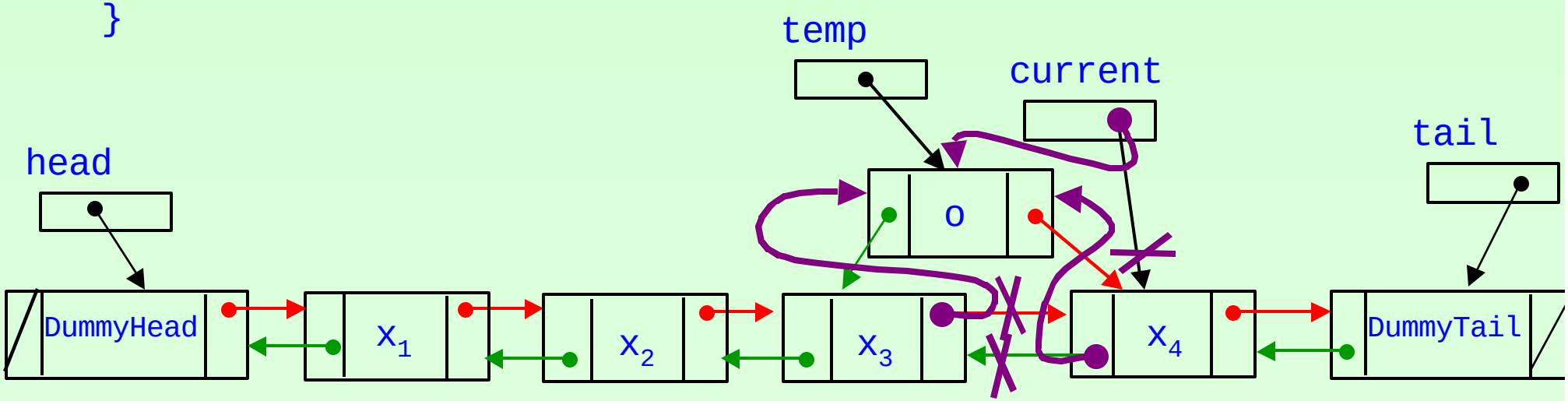
```
public void zeroth()    // set to zeroth position
  {  current = head;
  }

public void first() throws ItemNotFound
                        // set to zeroth position
  {  current = head.next;
     if (current==tail) {throw new ItemNotFound("Empty List");};
  }

public boolean isInList()    // checks if at a valid position
  {  returns (current!=null) && (current!=head)
                     && (current!=tail);
  }

public void advance()    // move to next position
  {  if (current!=null) {current=current.next;};
  }
```

New Functionalities — Doubly Linked-List

```
public void retreat ()    // retreat to previous item

  { if (current!=null) {current=current.prev;};
  }

public void insert (Object o) throws ItemNotFound
                            // insert into current position
  { ...
  }

public void delete () throws ItemNotFound
                           // delete node at current position
  { ...
  }
```

Insertion — Doubly Linked-List

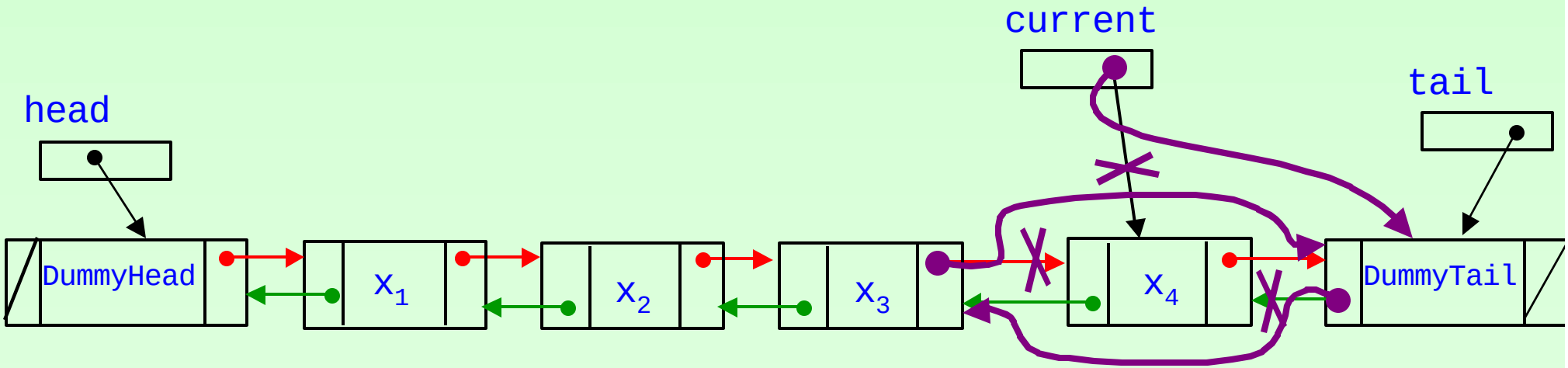
```
public void insert(Object o) throws ItemNotFound
                    // insert at current position
 { if (current != null) && (current != head)
        { DListNode temp = new DListNode(o,current,current.prev);
        current.prev.next = temp;
        current.prev = temp;
        current = temp;
          }
   else {throw new ItemNotFound("insert fails");}
 }
```



Deletion

Doubly Linked-List

```
public void delete() throws ItemNotFound
                // delete node at current position
{ if ((current!=null) && (current!=head) && (current != tail))
      { current.next.prev = current.prev;
      current.prev.next = current.next;
      current=current.next;
        }
  else { throw new ItemNotFound("No Node to Delete"); }
```

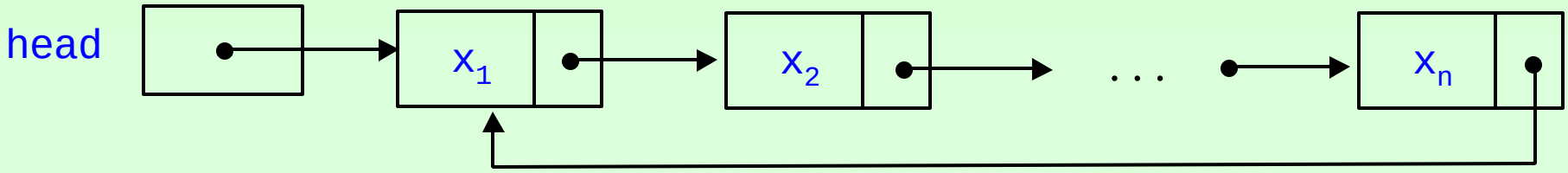


Circular Linked List

May need to cycle through a list repeatedly, e.g. round robin system for a shared resource.

Solution : Have the last node point to the first node.

Circular Linked List.

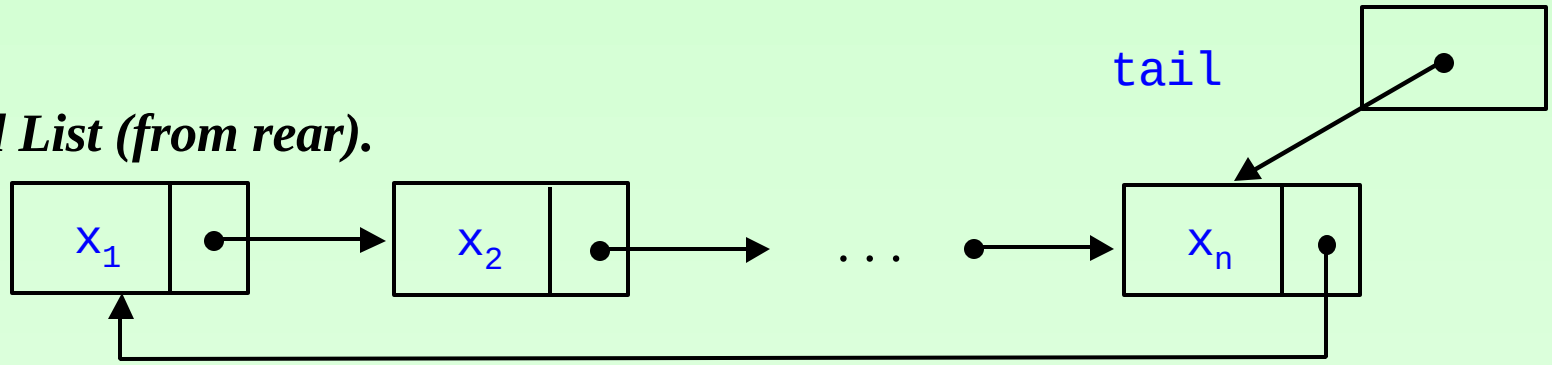


With Rear Pointer Circular Linked-List

For singly circular linked-list, it may be better to have a pointer from the rear.

Two pointers for the price of one : `(head == tail.next)`

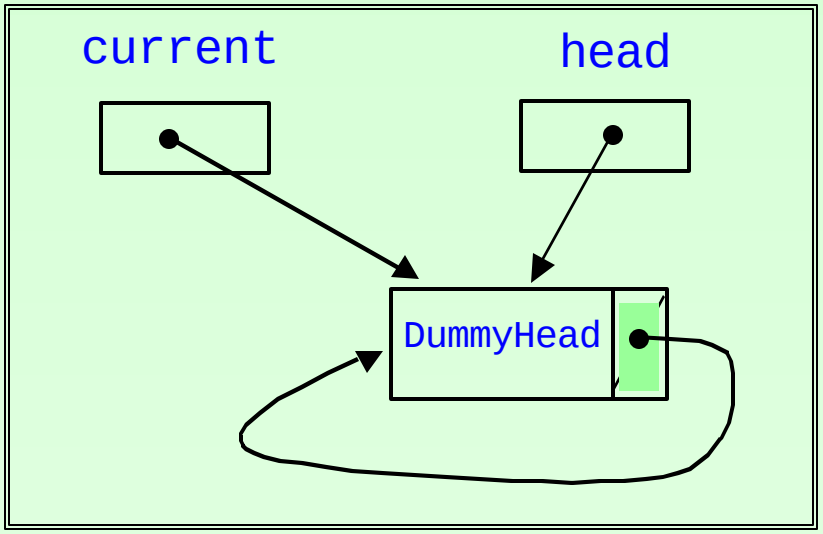
Circular Linked List (from rear).



With Dummy Header Circular Linked-List

```
class LinkedListItr extends LinkedList{
   private ListNode current;

   public LinkedListItr()
     {head = new ListNode("DummyHead",null);
     head.next = head;
     current = head; }
```

LinkedListItr



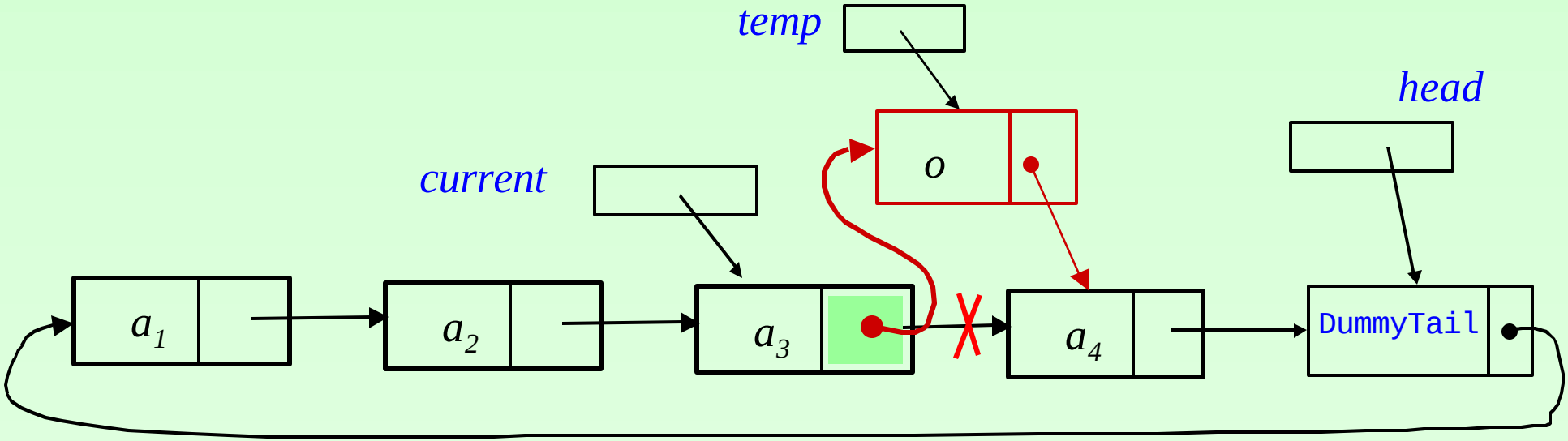
More Codes — Circular Linked-List

```
public void first() throws ItemNotFound
    { current = head.next;
      if (current == head)
           throw new ItemNotFound("No first element");
    }
public void isLast()
    { if (current!=null && current!=head)
           {return current.next == head;}
      else {return false;};
    }
public void advance()   // advance to next node
    { if (current!=null)
        {current = current.next;
         if (current==head) {current = null;};}
    }

public void advanceWrap()   // supports wrap around
    { current = current.next;
      if (current==head) {current = current.next;};
    }
```

Insertion

Circular Linked-List

```
public void insertNext(Object o) throws ItemNotFound
                       // insert after current position
   { ListNode temp;
     if (current!=null) {temp = new ListNode(o,current.next);
                         current.next = temp;}
     else { throw new ItemNotFound("insert fails"); };
   }
```



Deletion — Circular Linked-List

```
public void deleteNext() throws ItemNotFound
                        // delete node after current position
{ if (current!=null)
       {if (current.next!=head)
           {current.next = current.next.next;}
        else {throw new ItemNotFound("No Next Node to Delete");}
       }
  else {throw new ItemNotFound("No Next Node to Delete");};
 }
```