CS201: Data Structures and Discrete Mathematics I

Linked List, Stacks and Queues

Data Structure

- A *construct* that can be defined within a programming language to store a collection of data
  - one may store some data in an array of integers, an array of objects, or an array of arrays

Abstract Data Type (ADT)

- Definition: a collection of *data* together with a set of *operations* on that data
  - specifications indicate *what* ADT operations do, but not *how* to implement them
  - data structures are part of an ADT's implementation
- Programmer can use an ADT without knowing its implementation.

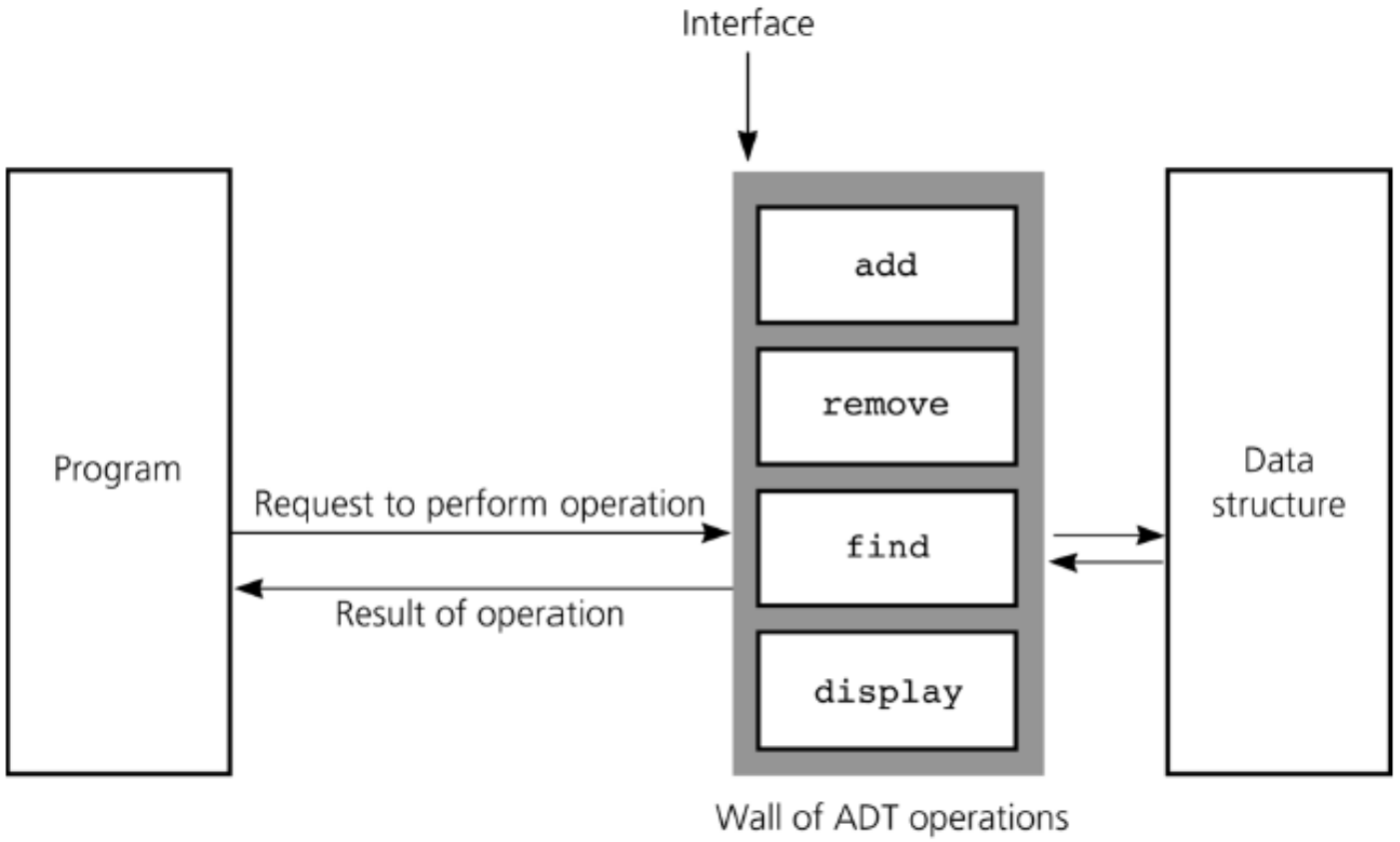
Typical Operations on Data

- Add data to a data collection
- Remove data from a data collection
- Ask questions about the data in a data collection. E.g., what is the value at a particular location.

Why ADT

- Hide the unnecessary details
- Help manage software complexity
- Easier software maintenance
- Functionalities are less likely to change
- Localised rather than global changes

Illustration



Linked Lists

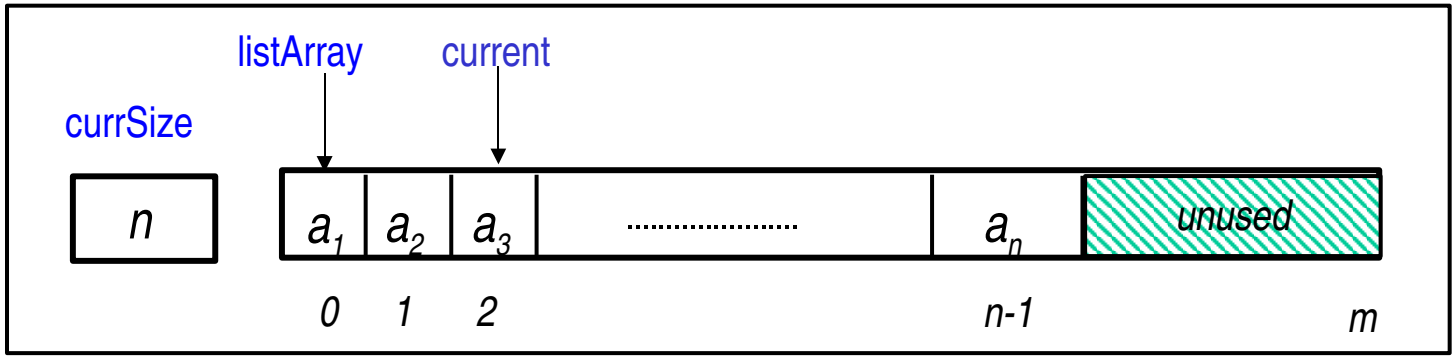
Lists

- List: a finite sequence of data items

  a1; a2; a3; : : : an
- Lists are pervasive in computing
  - e.g. class list, list of chars, list of events
- Typical operations:
  - Creation
  - Insert / remove an element
  - Test for emptiness
  - Find an element
  - Current element / next / previous
  - Find k-th element
  - Print the entire list

Array-Based List Implementation

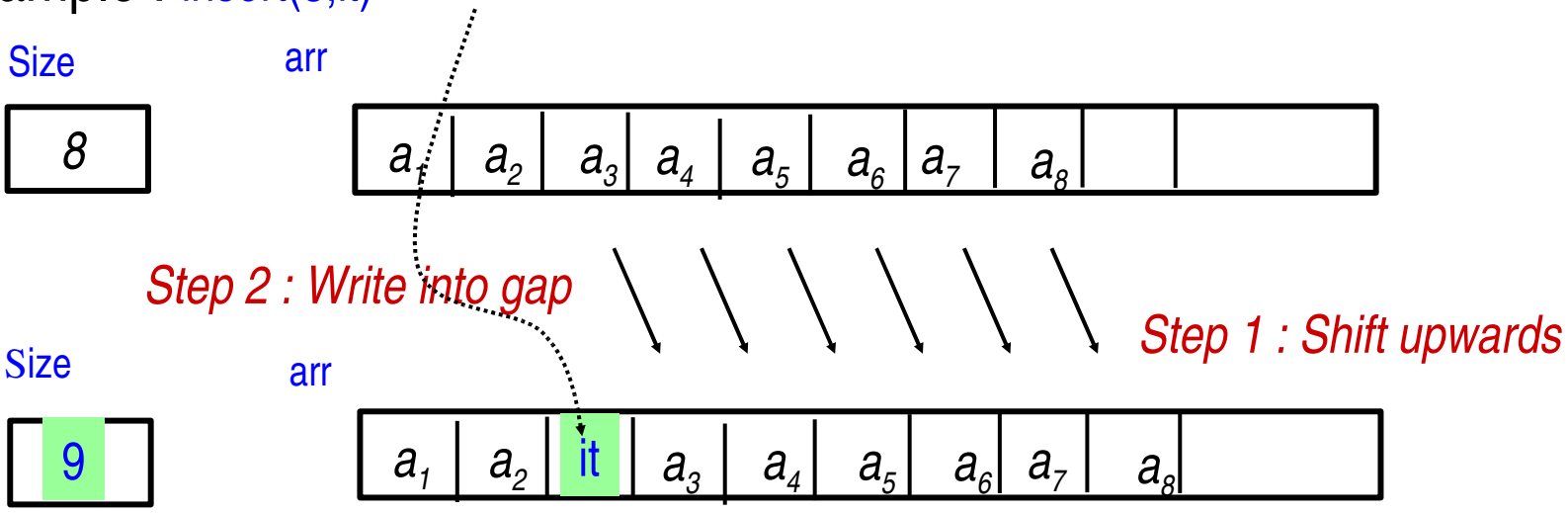
- One simple implementation is to use java arrays
  - A sequence of n-elements
- Maximum size must be anticipated a priori.
- Internal variables:
  - Maximum size *maxSize* (m)
  - Current size *currSize* (n)
  - Current index *current*
  - Array of elements listArray



Inserting Into an Array

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

Example : insert(3,it)



Step 3 : Update Size

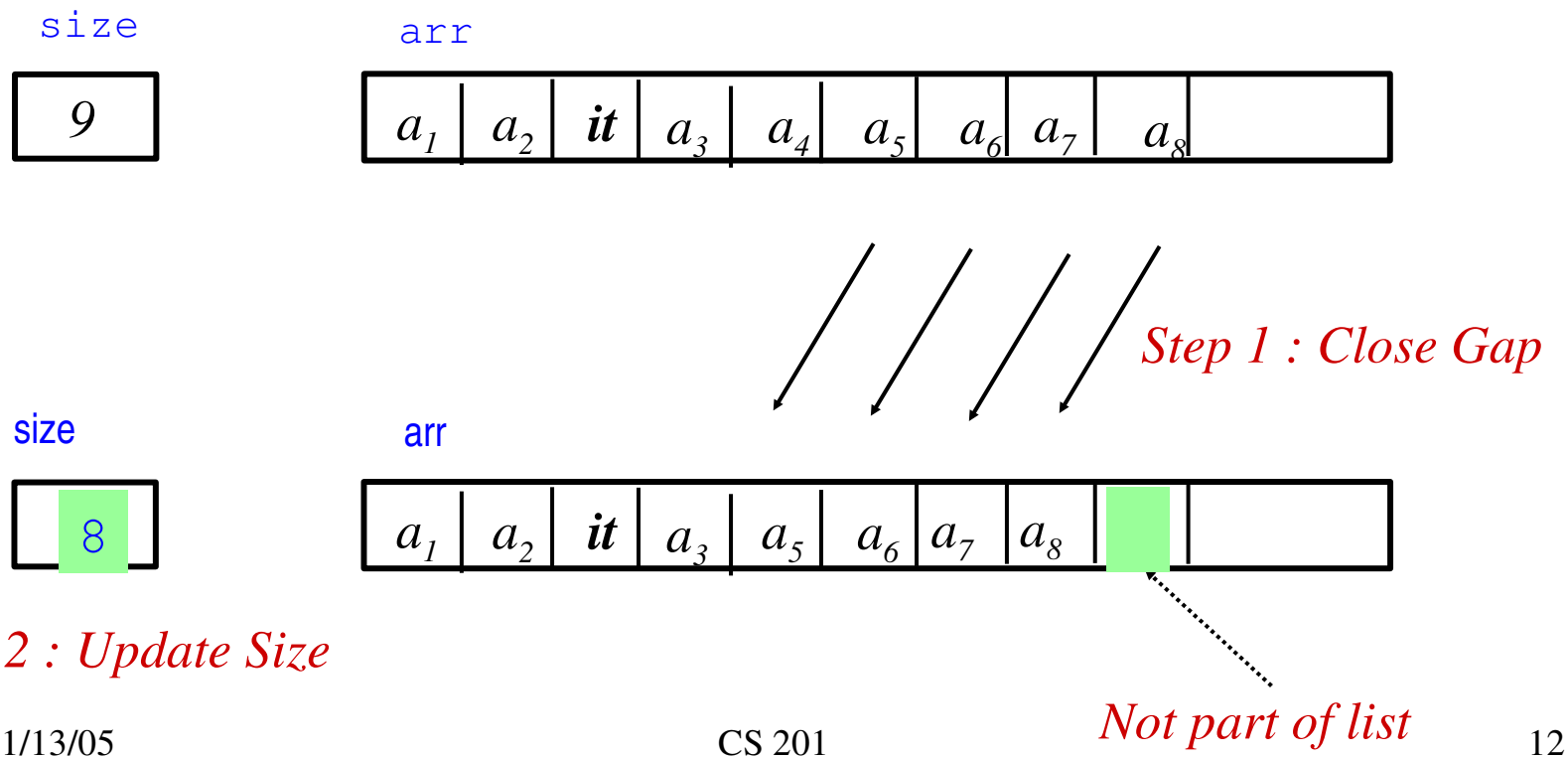
Coding

```
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)`

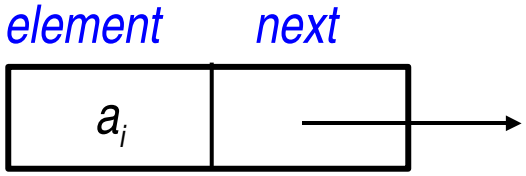


Coding

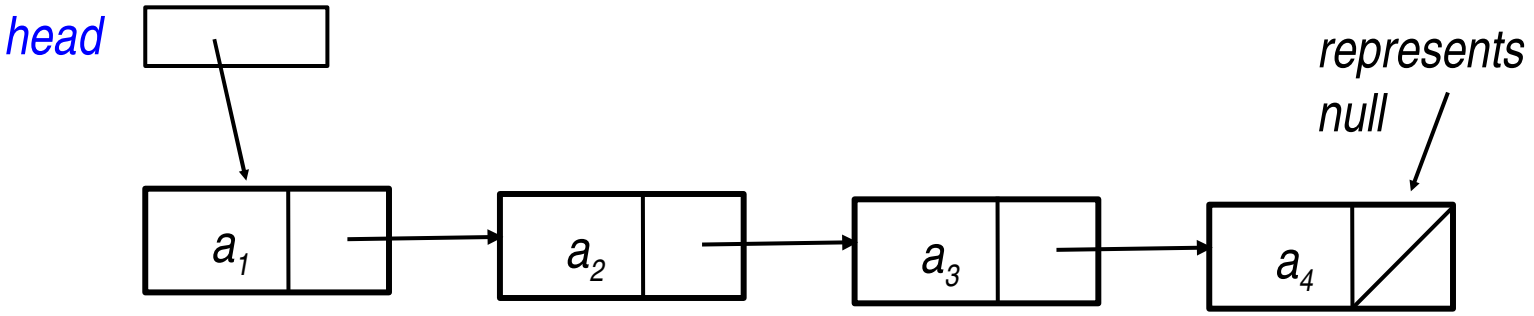
```
public void delete(int j)
{                                  // pre : 1<=j<=size
   for (i=j+1; i<=size; i=i+1)
      { arr[i-i]=arr[i]; }; // Step1: Close gap

      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



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



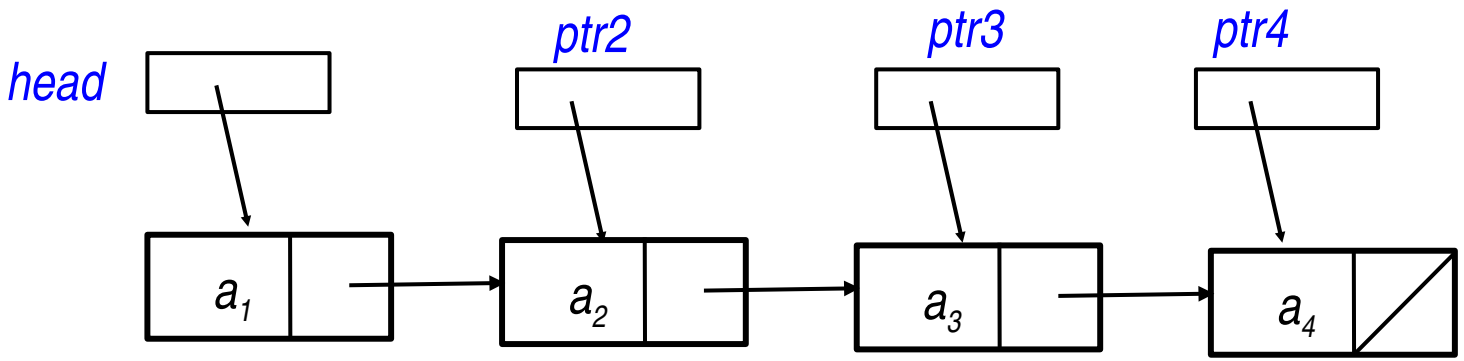
Coding

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

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

      { element = o;
        next = n; }
   }
```

Add elementz to list

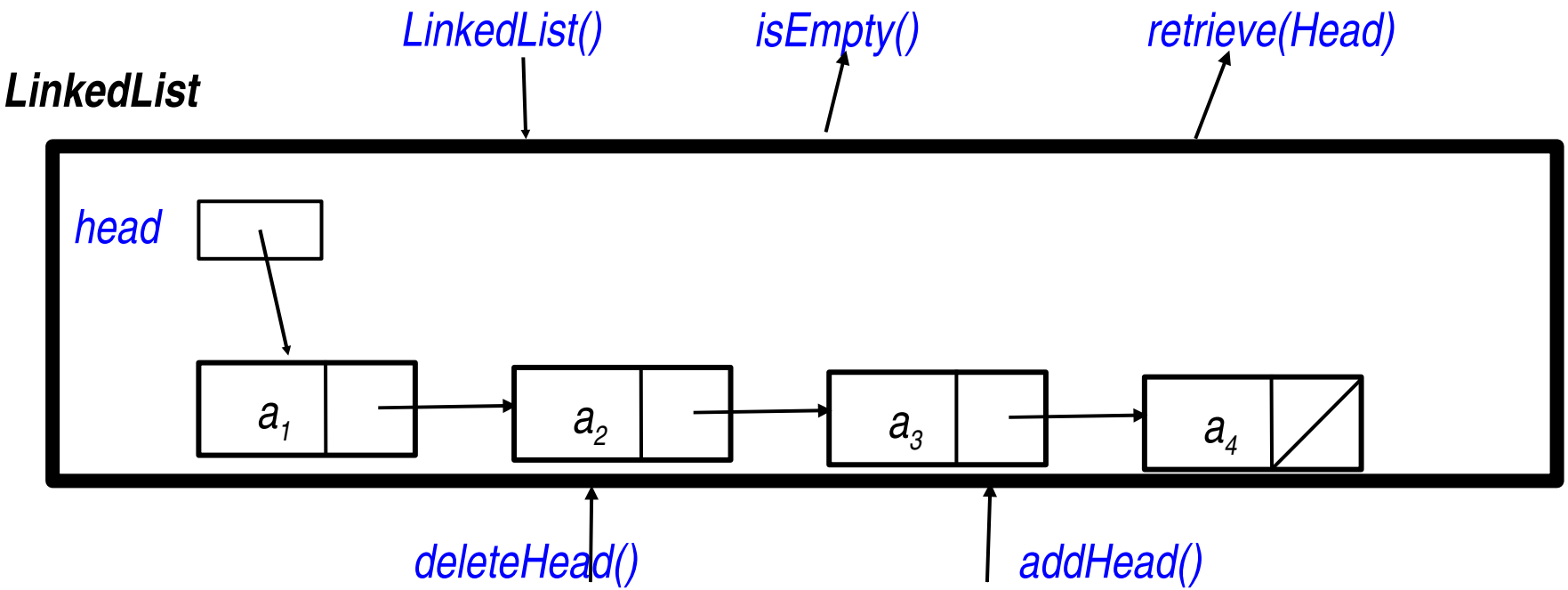
The earlier sequence can be built by:

- 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

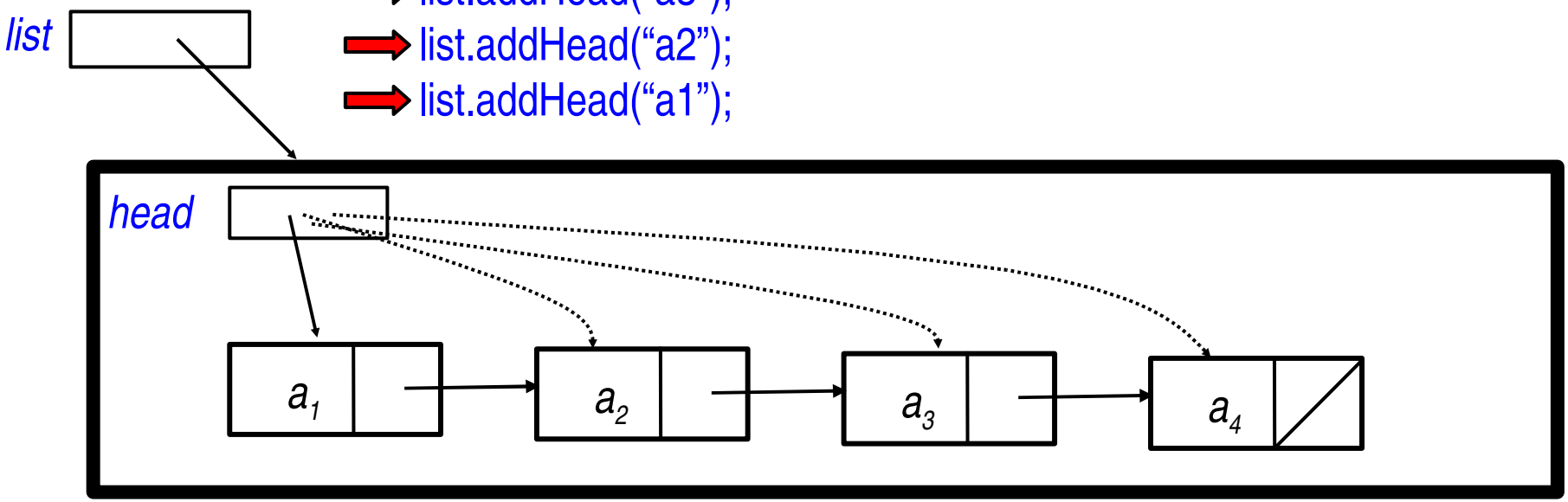
- We can provide an ADT for linked-list.
  - This can help hide unnecessary internal details



Sample

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");

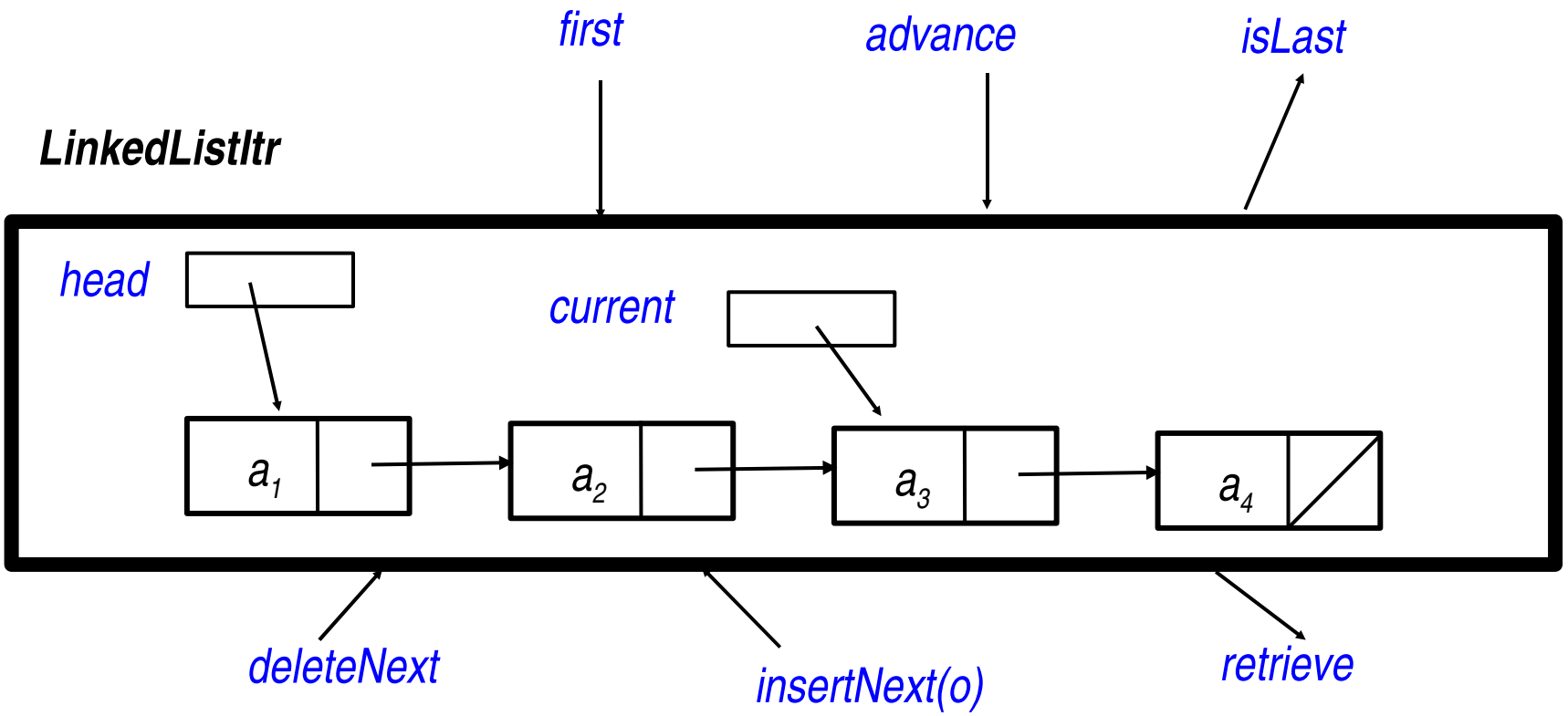


Coding

```
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

```
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

Coding

```
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)

More coding

```
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 {}
           };
       }
```

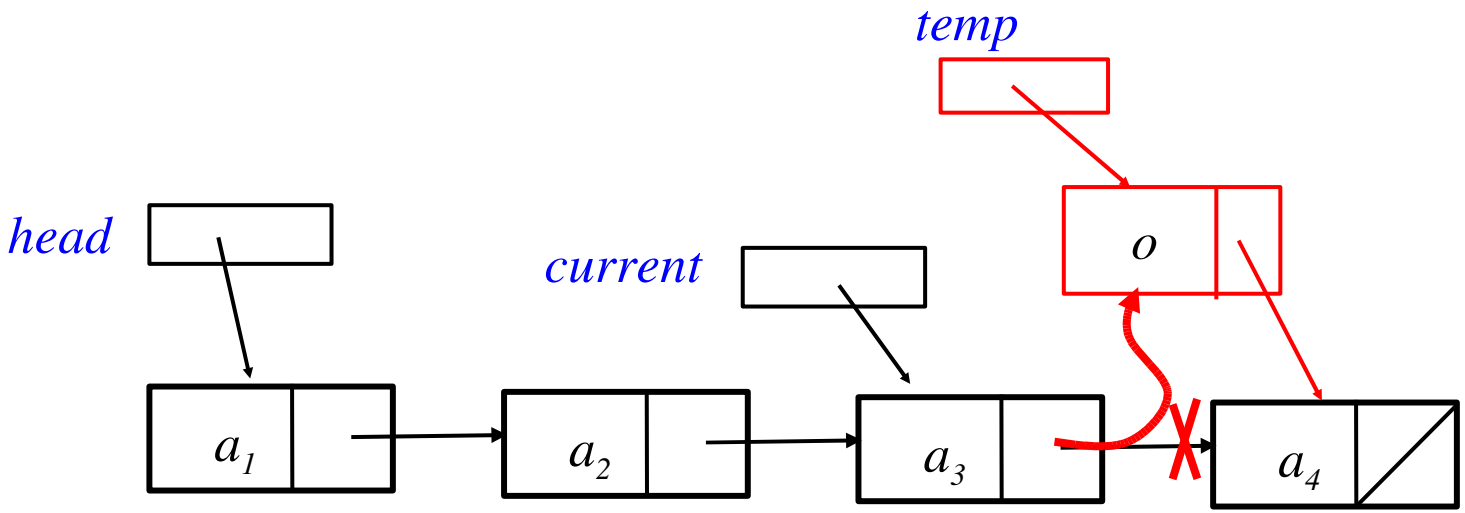
More coding

```
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 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

```
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");};
   }
```

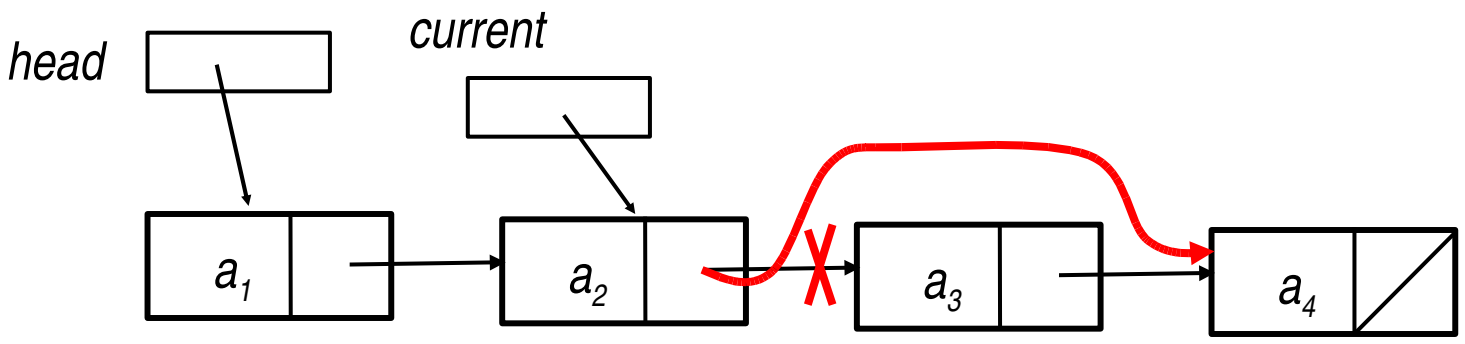
?



Delete

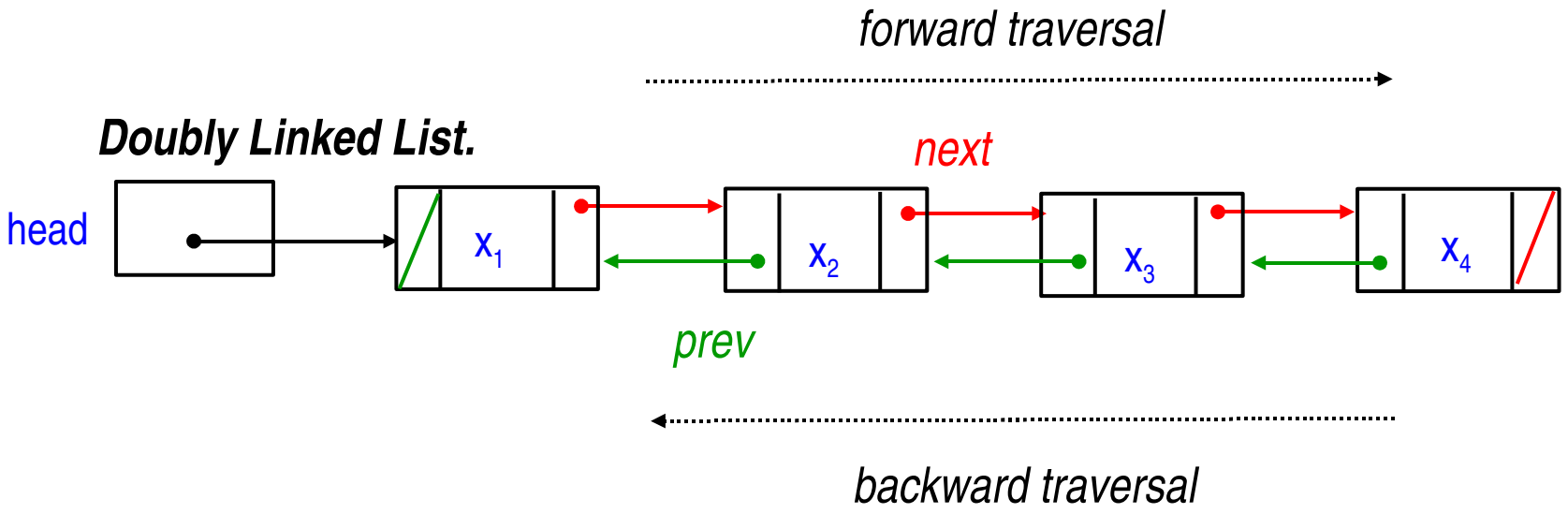
```
public void deleteNext() throws ItemNotFound
                // delete node after current position
{ if (current!=null)
     {if (current.next!=null)
         {current.next = current.next.next;}
      else {throw new ItemNotFound("No Next Node to Delete");};
   }
  else if (zeroflag && head!=null)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

?



Doubly Liked Lists

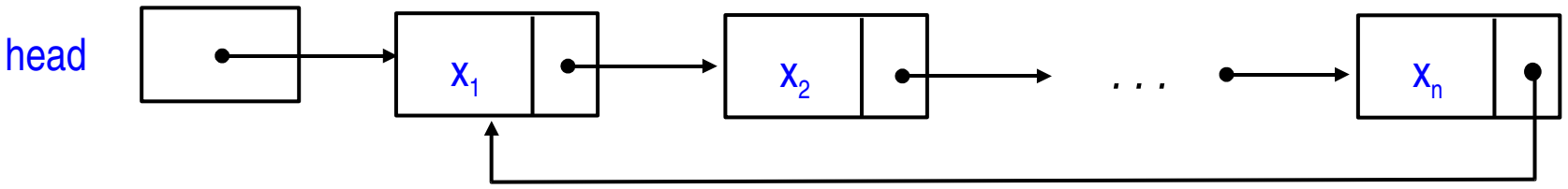
- Frequently, we need to traverse a sequence in BOTH directions efficiently
- *Solution* : Use doubly-linked list where each node has two pointers



Circular Linked Lists

- 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

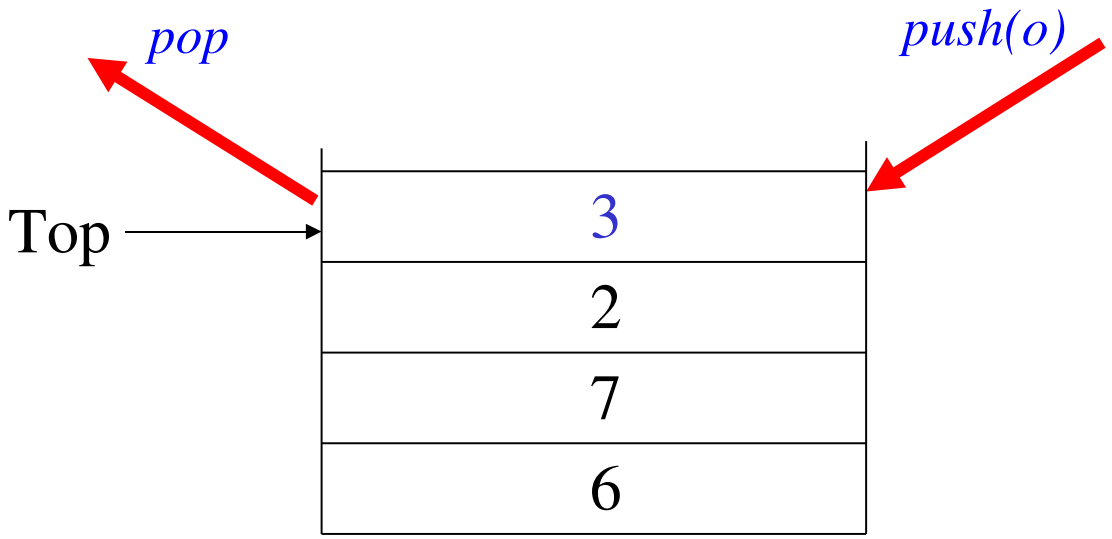
Circular Linked List.



Stacks

What is a Stack?

- A *stack* is a list with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called the *top*.
- The operations: push (insert) and pop (delete)



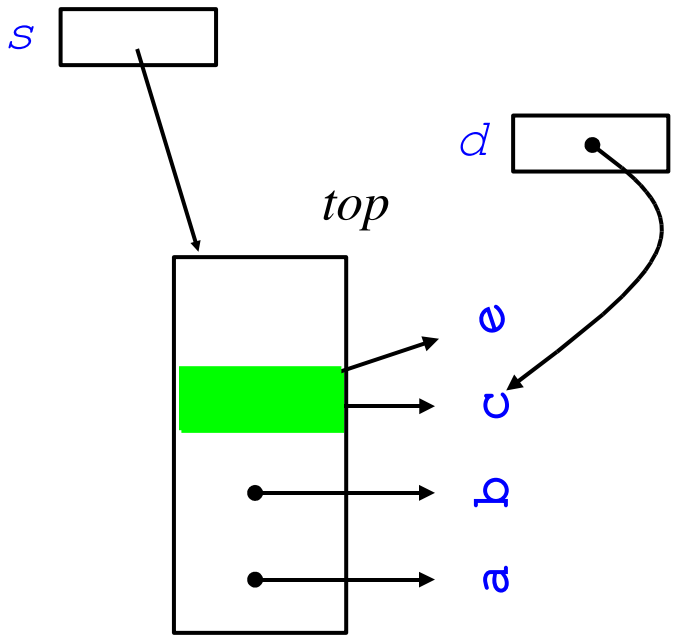
Stack ADT Interface

- We can use Java Interface to specify Stack ADT Interface

```
interface Stack {

 boolean isEmpty();                          // return true if empty

 void push(Object o);                  // insert o into stack

 void pop() throws Underflow;                // remove most recent item

 void popAll();                              // remove all items from stack

 Object top() throws Underflow;        // retrieve most recent item

 Object topAndPop() throws Underflow;  // return & remove most recent item
}
```

Sample Operation

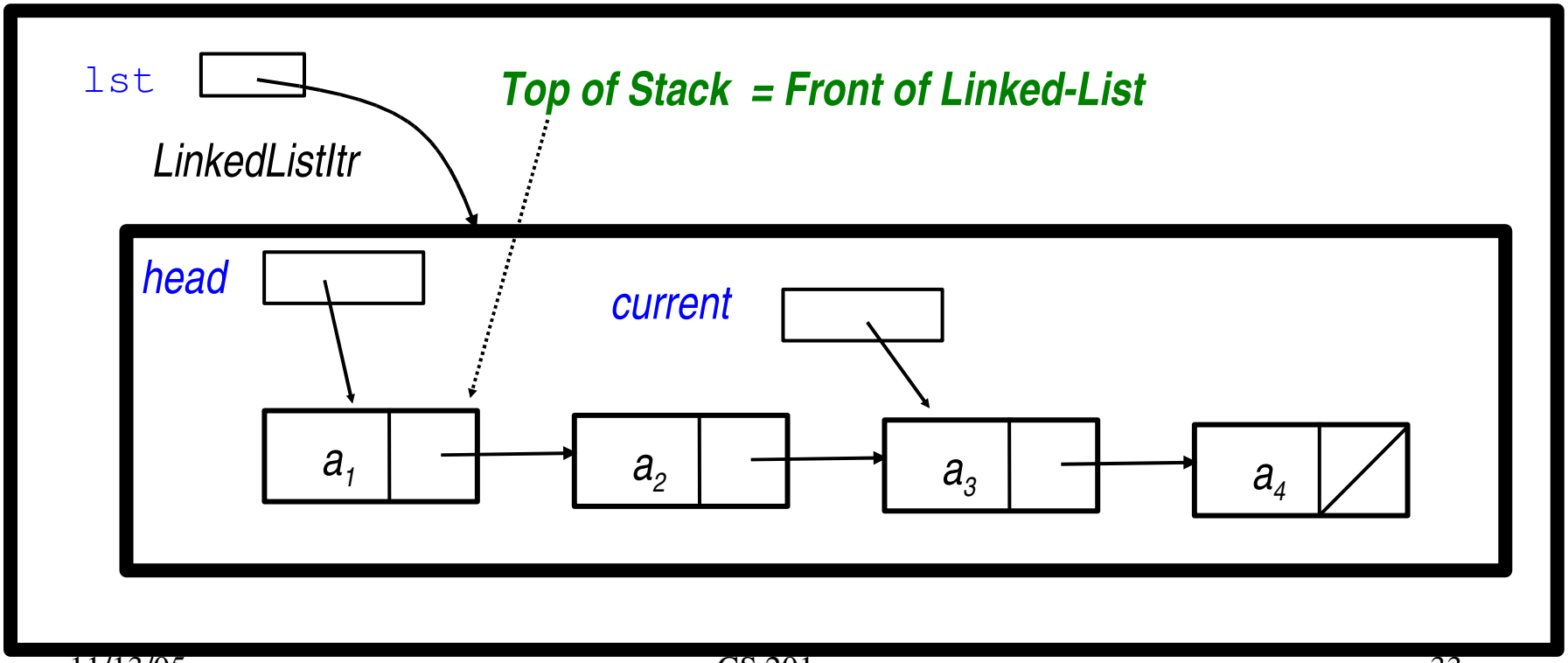
- `Stack s = makeStack();`
- `s.push("a");`
- `s.push("b");`
- `s.push("c");`
- `d=s.top();`
- `s.pop();`
- `s.push("e");`
- `s.pop();`



Implementation by Linked Lists

- Can use LinkedListItr as implementation of stack

StackLL



Code

```
Class StackLL implements Stack {

 private LinkedListItr lst;

 public StackLL() { lst = new LinkedListItr(); }

 public Stack makeStack() { return new StackLL(); }

 public boolean isEmpty()              // return true if empty
  { return lst.isEmpty();  };

 public void push(Object o)            // add o into the stack
  { lst.addHead(o); }

 public void pop() throws Underflow // remove most recent item
  { try {lst.deleteHead();}
    catch (ItemNotFound e)
     {throw new Underflow("pop fails - empty stack")};
  }
```

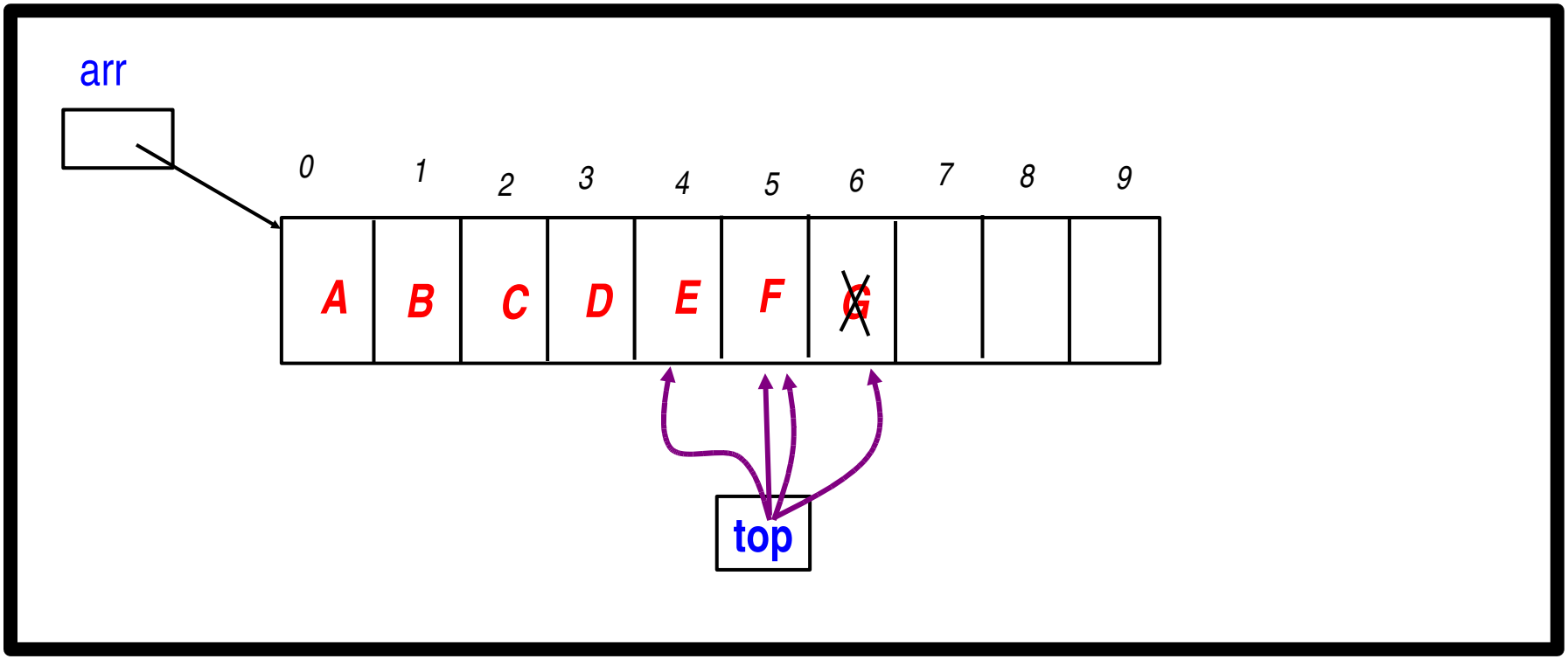
More code

```
public Object top() throws Underflow; // retrieve most recent
   item
  { try { lst.first();
      return lst.retrieve();
    } catch (ItemNotFound e)
   {throw new Underflow("top fails – empty stack");};
  }

 public Object topAndPop() throws Underflow;
                        // return & remove most recent item
  { try { lst.first();
       Object p=lst.retrieve();
       lst.deleteHead();
       return p;
    } catch (ItemNotFound e)
   {throw new Underflow("topAndPop fails – empty stack");};
  }
```

Implementation by Array

- Can use Array with a top index pointer as an implementation of stack

StackAr



Code

```
class StackAr implements Stack {

 private Object [] arr;
 private int top;
 private int maxSize;
 private final int initSize = 1000;
 private final int increment = 1000;

 public StackAr() { arr = new Object[initSize];
             top = -1;
             maxSize=initSize }

 public Stack makeStack() { return new StackAr(); }

 public boolean isEmpty()
  { return (top<0);  }

 private boolean isFull()
  { return (top>=maxSize);  }
```

More code

```
public void push(Object o)        // insert o
  { top++;
    if (this.isFull()) this.enlargeArr() ;
    arr[top]=o; }

 private void enlargeArr()        // enlarge the array
  {int newSize = maxSize+increment;
   Object [] barr = new Object[newSize];
   for (int j=0;j<maxSize;j++) {barr[j]=arr[j];};
   maxSize = newSize;   arr = barr;
  }
```

More code

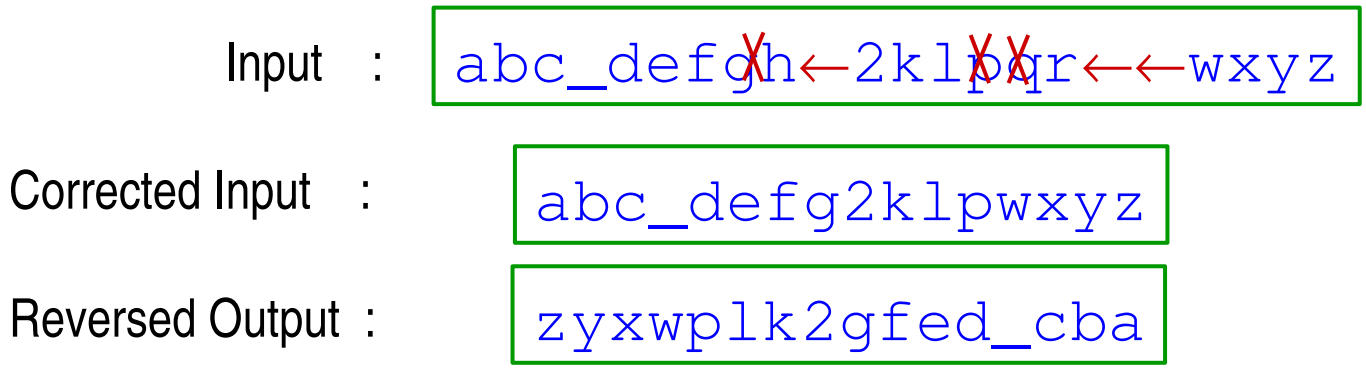
```
public void pop() throws Underflow
  { if (!this.isEmpty()) {top--;}
    else {throw new Underflow("pop fails – empty stack");};
  }

public Object top() throws Underflow
 { if (!this.isEmpty()) {return arr[top];}
   else {throw new Underflow("top fails – empty stack");};
 }

public Object topAndPop() throws Underflow
 { if (!this.isEmpty())  { Object t = arr[top];
                           top--;
                           return t;
                         };
  else {throw new Underflow("top&pop fails – empty stack");};
 }
```

Applications

- Many application areas use stacks:
  - line editing
  - bracket matching
  - postfix calculation
  - function call stack

Line Editing

- A line editor would place characters read into a buffer but may use a backspace symbol (denoted by ←) to do error correction
- *Refined Task*
  - read in a line
  - correct the errors via backspace
  - print the corrected line in reverse



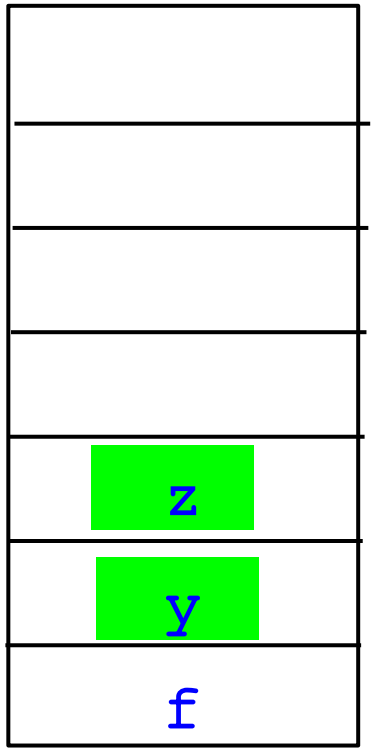
The Procedure

- Initialize a new stack
- For each character read:
  - if it is a backspace, *pop out last char entered*
  - if not a backspace, *push the char into stack*
- To print in reverse, pop out each char for output

Input : `fgh←r←←yz`

Corrected Input : `fyz`

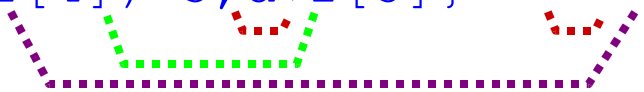
Reversed Output : `zyf`



Stack

Bracket Matching Problem

- Ensures that pairs of brackets are properly matched
  - An Example: `{a,(b+f[4])*3,d+f[5]}`



  - Bad Examples:

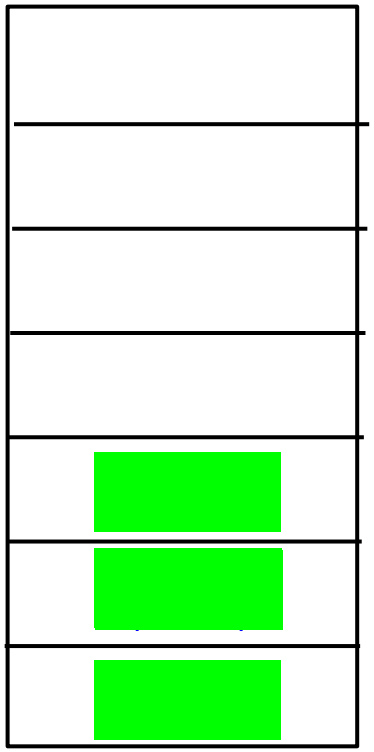
```
(..)..)          // too many closing brackets

(..(..)          // too many open brackets

[..(..]..)       // mismatched brackets
```

Informal Procedure

Initialize the stack to empty
For every char read
    if open bracket then *push onto stack*
    if close bracket, then
        return & remove most recent item
            from *the stack*
        if doesn't match then *flag error*
    if non-bracket, *skip the char read*

Example

```
{a,(b+f[4])*3,d+f[5]}
```

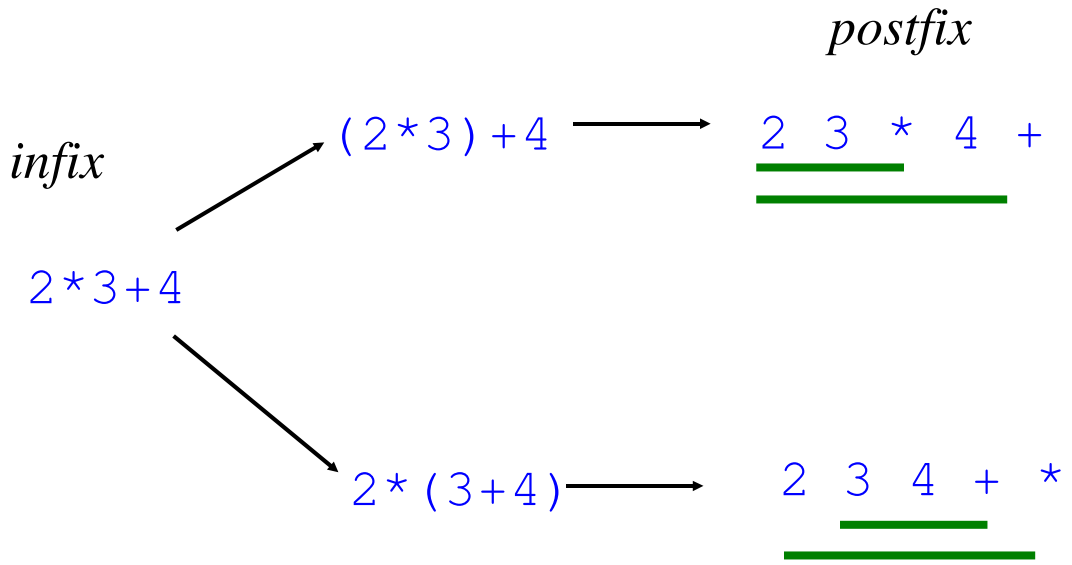


Stack

Postfix Calculator

- Computation of arithmetic expressions can be efficiently carried out in Postfix notation with the help of a stack.

Infix - arg1 **op** arg2
Prefix - **op** arg1 arg2
Postfix - arg1 arg2 **op**



Informal Procedure

Initialise stack
For each item read.
    If it is an operand,
        push on the stack
    If it is an operator,
        pop arguments from stack;
        perform operation;
        push result onto the stack

Expr	
2	`s.push(2)`
3	`s.push(3)`
4	`s.push(4)`

+	`arg2=s.topAndPop()`
	`arg1=s.topAndPop()`
	`s.push(arg1+arg2)`

*	`arg2=s.topAndPop()`
	`arg1=s.topAndPop()`
	`s.push(arg1*arg2)`

Stack

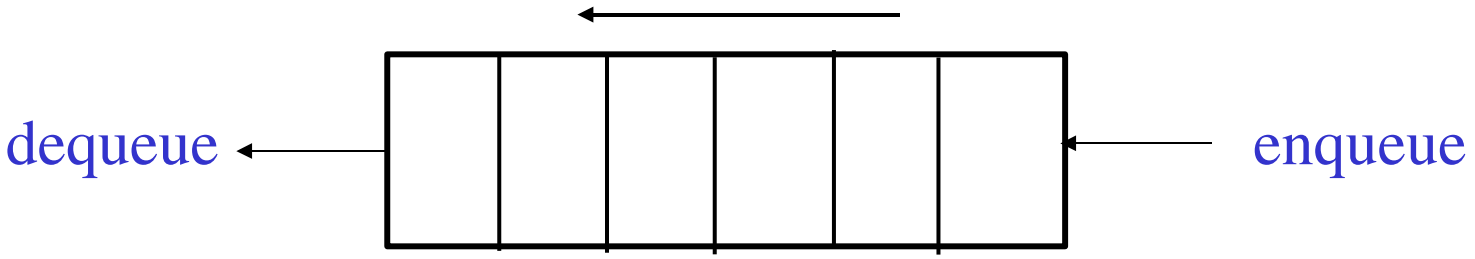
Summary

- The ADT stack operations have a last-in, first-out (LIFO) behavior
- Stack has many applications
  - algorithms that operate on algebraic expressions
  - a strong relationship between recursion and stacks exists
- Stack can be implemented by arrays and linked lists

Queues

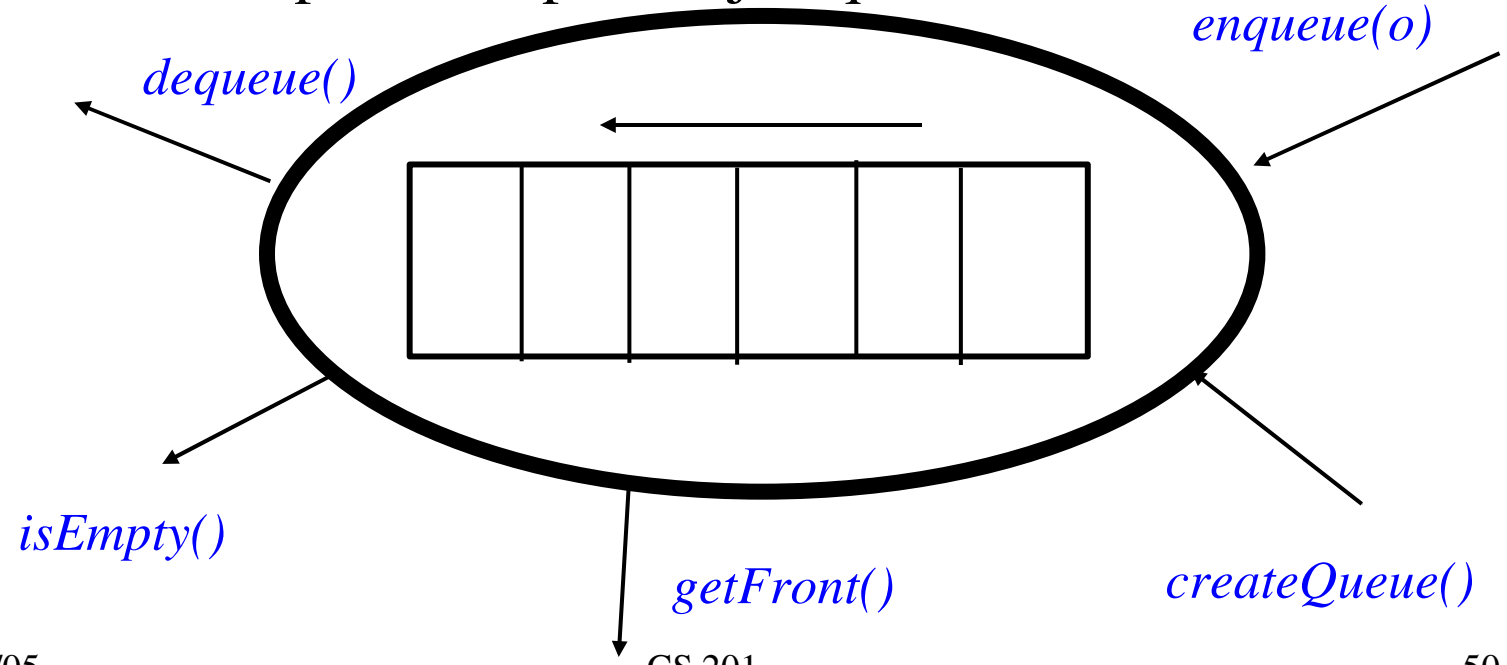
What is a Queue?

- Like stacks, queues are lists. With a queue, however, insertion is done at one end whereas deletion is done at the other end.
- Queues implement the FIFO (first-in first-out) policy. E.g., a printer/job queue!
- Two basic operations of queues:
  - dequeue: remove an element from front
  - enqueue: add an element at the back



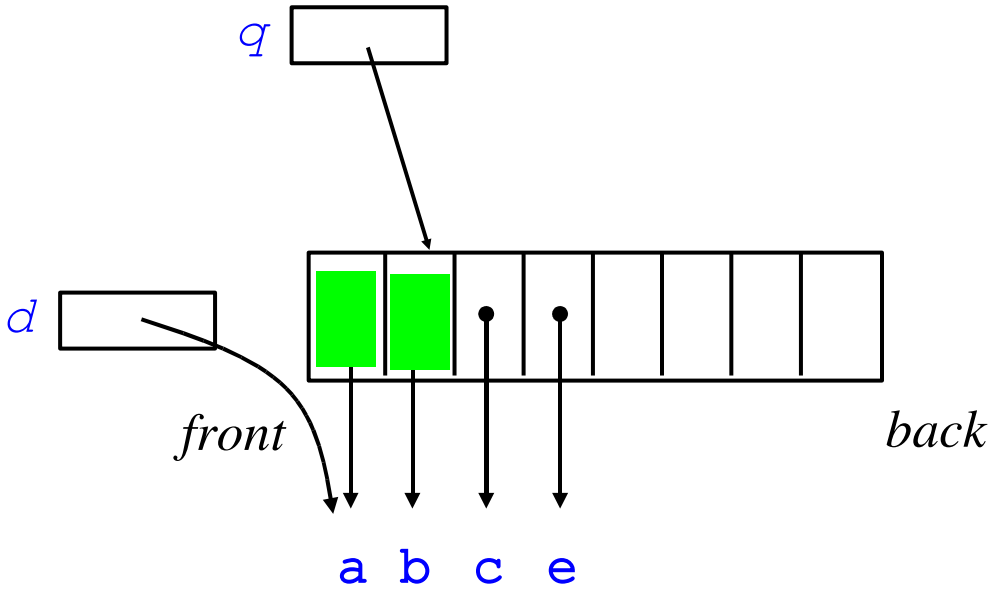
Queue ADT

- Queues implement the FIFO (first-in first-out) policy
  - An example is the printer/job queue!



Sample Operation

- `Queue q = createQueue();`
- `q.enqueue("a");`
- `q.enqueue("b");`
- `q.enqueue("c");`
- `d=q.getFront();`
- `q.dequeue();`
- `q.enqueue("e");`
- `q.dequeue();`



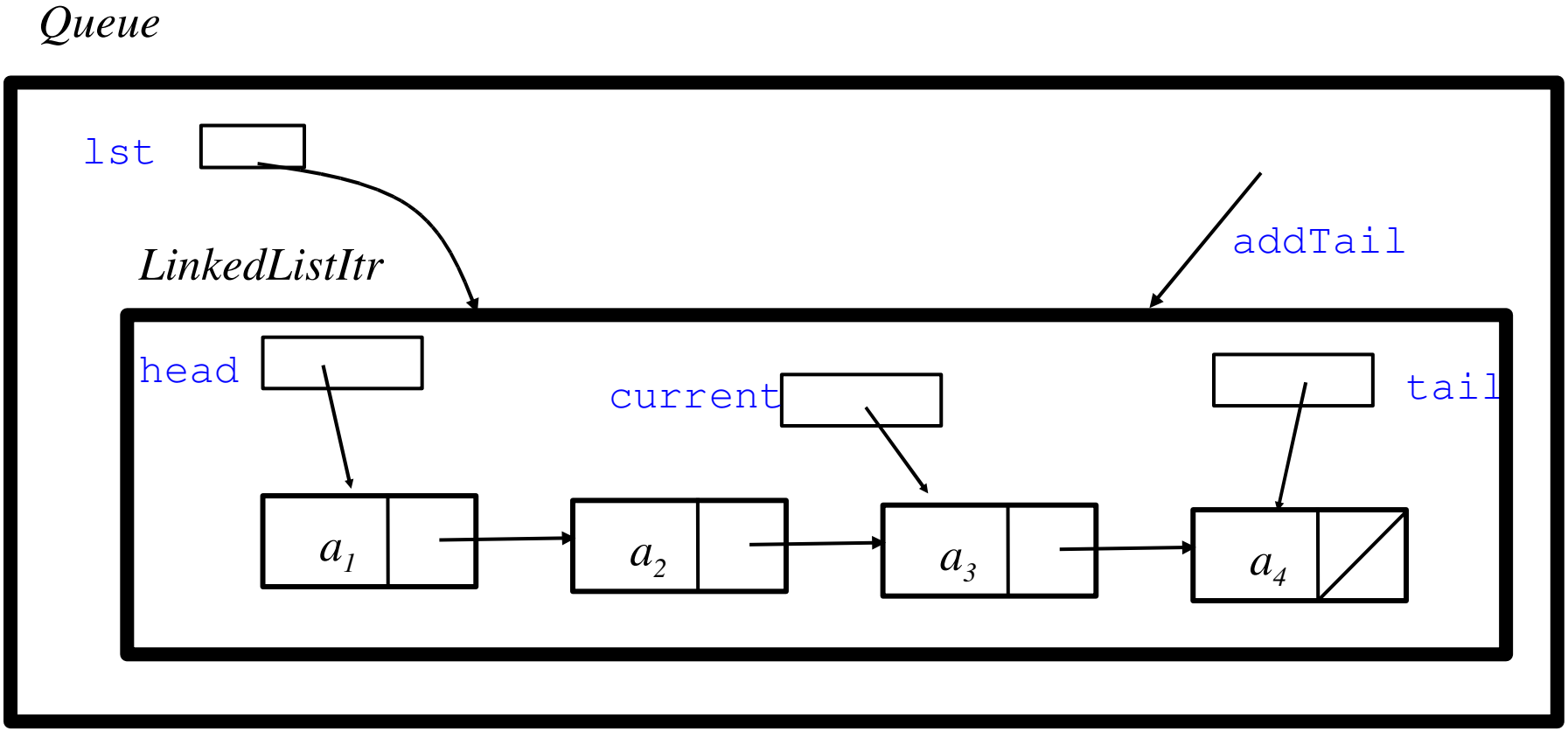
Java Interface

- We can also use Java Interface to specify Queue ADT Interface. This provides a more abstract mechanism to support simultaneous implementations

```
interface Queue {

 void enqueue(Object o)                 // insert o to back of Q

 void dequeue() throws Underflow;       // remove oldest item

 Object getFront() throws Underflow;    // retrieve oldest item

 boolean isEmpty();                     // checks if Q is empty

}
```

Implementation of Queue (Linked List)

- Can use LinkedListItr as underlying implementation of Queues



Code

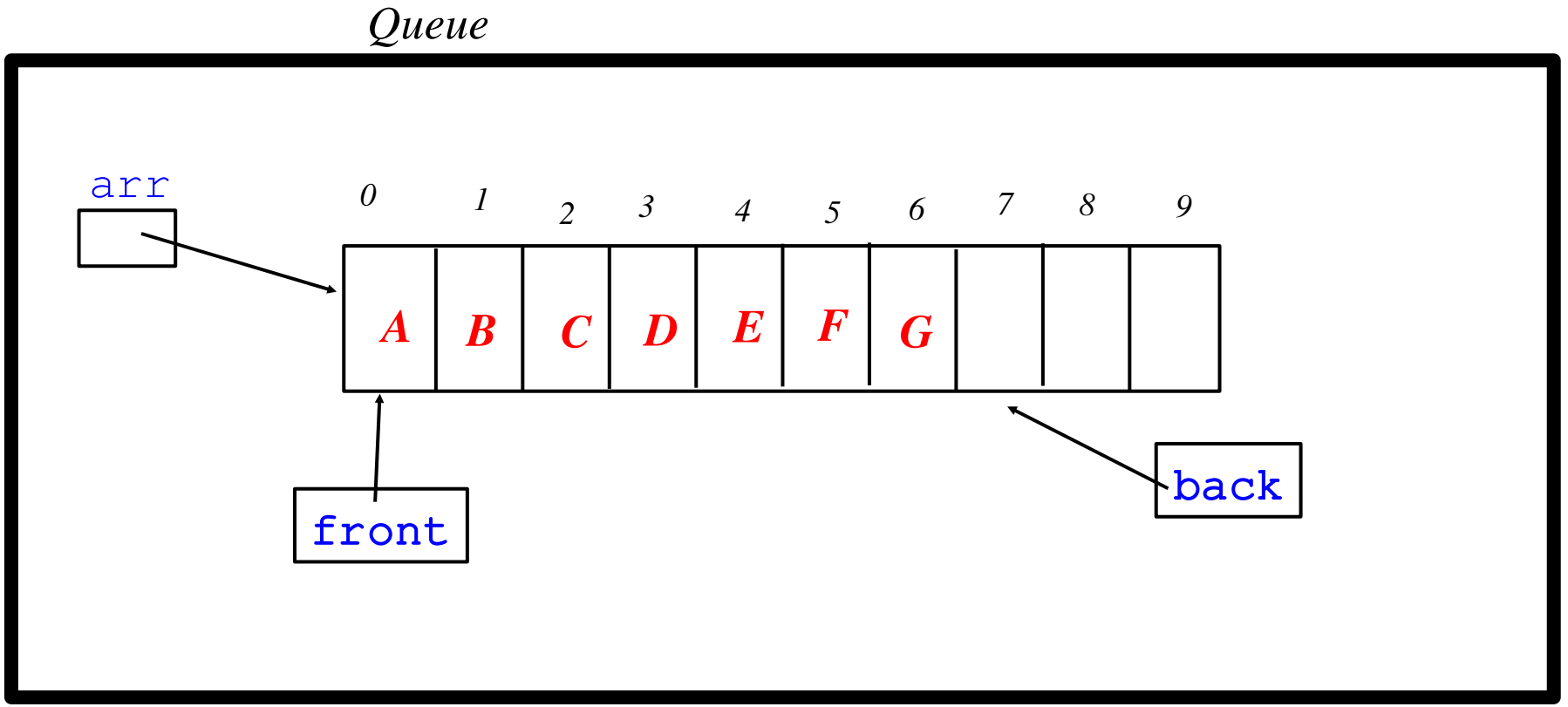
```
class QueueLL implements Queue {

 private LinkedListItr lst;

 public QueueLL() { lst = new LinkedListItr(); }

 public static Queue makeQueue()            // return a new empty queue
  { return new QueueLL();  }


 public void enqueue(Object o)              // add o to back of queue
  { lst.addTail(o); }

 public void dequeue() throws Underflow     // remove oldest item
    { try {lst.deleteHead();
      catch (ItemNotFound e)
     {throw new Underflow("dequeue fails - empty q";};
  }
```

More code

```
public Object getFront() throws Underflow;   // retrieve
   oldest item
  { try { lst.first();
      return lst.retrieve();
    } catch (ItemNotFound e)
   {throw new Underflow("getFront fails – empty queue");};
  }

public boolean isEmpty()                  // return true if empty
  { return lst.isEmpty();  }
```

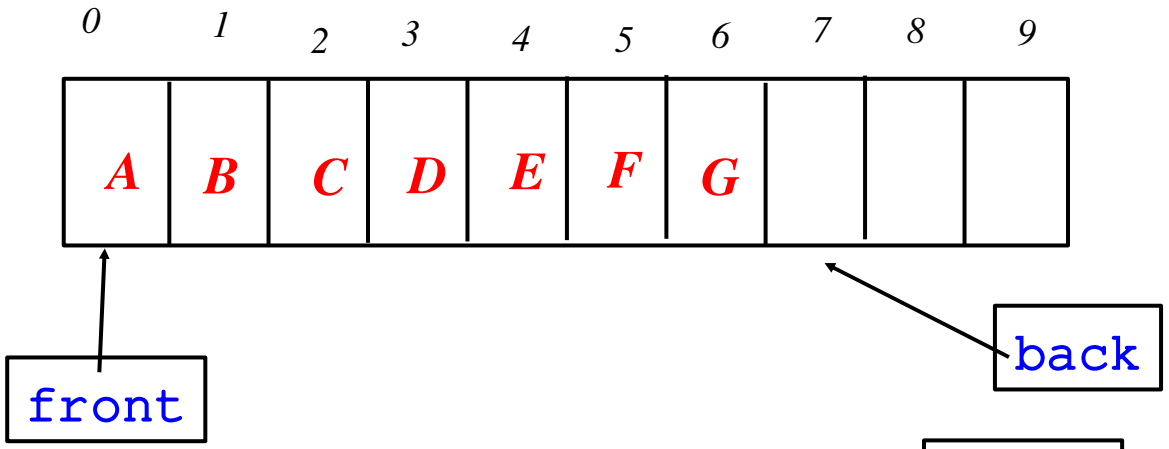
Implementation of Queue (Array)

- Can use Array with front and back pointers as implementation of queue

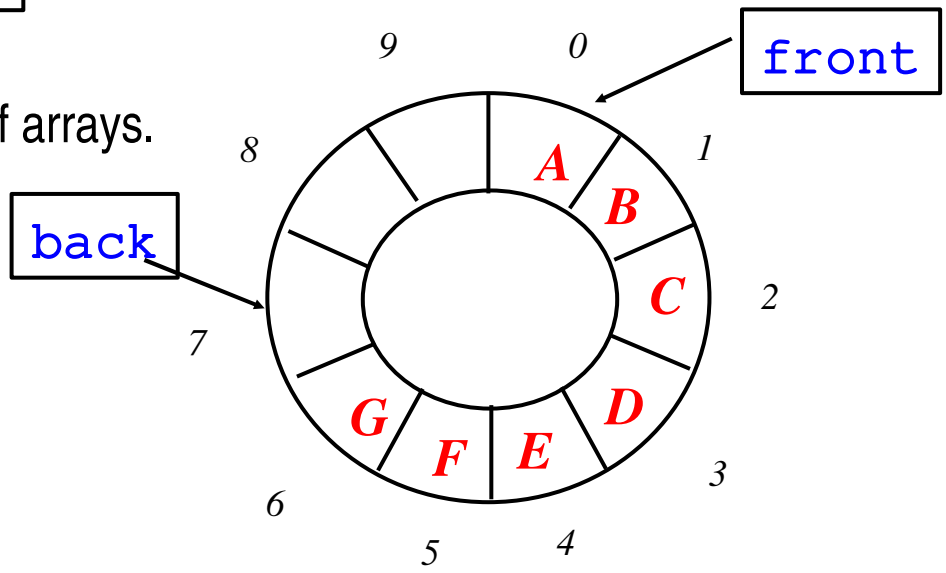


Circular Array

- To implement queue, it is best to view arrays as circular structure



Circular view of arrays.



How to Advance

- Both front & back pointers should make advancement until they reach end of the array. Then, they should re-point to beginning of the array

```
front = adv(front);
back  = adv(back);
```

```
public static int adv(int p)
 { int r = p+1;
   if (r<maxsize) return r;
   else return 0;
 }
```

upper bound of the array

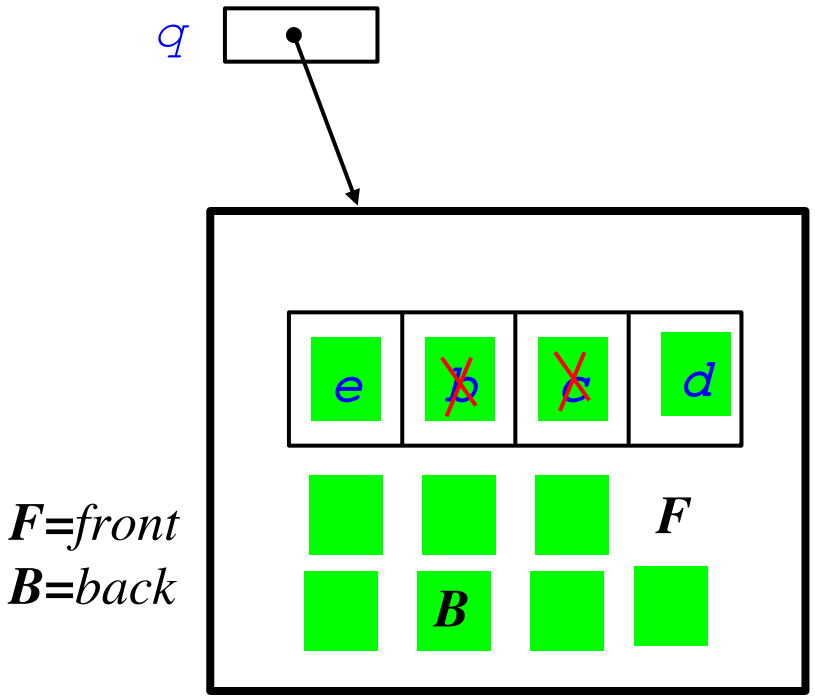
Alternatively, use modular arithmetic:

```
public static int adv(int p)
 { return ((p+1) % maxsize);
 }
```

mod operator

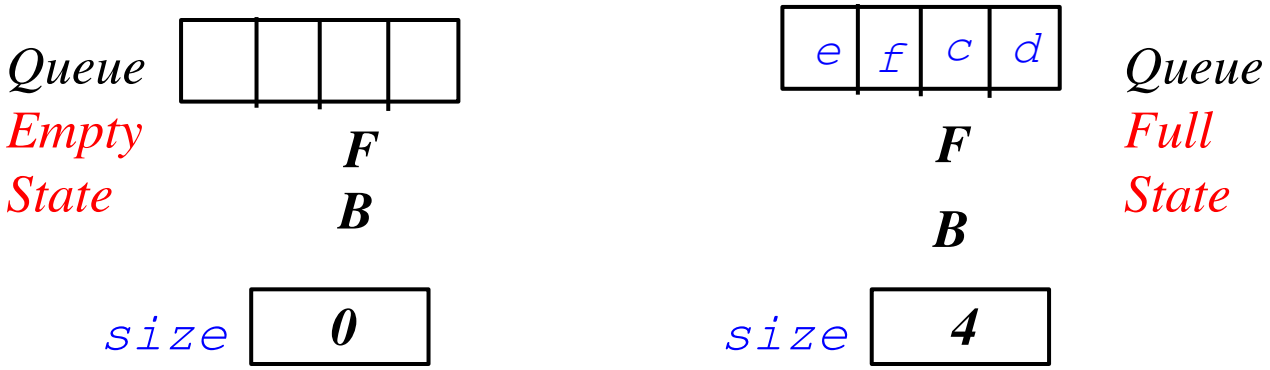
Sample

```
Queue q = QueueAR.makeQueue();
q.enqueue("a");
q.enqueue("b");
q.enqueue("c");
q.dequeue();
q.dequeue();
q.enqueue("d");
q.enqueue("e");
q.dequeue();
```



Checking for Full/Empty State

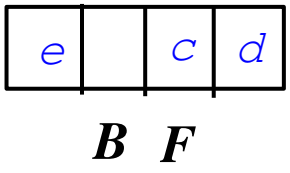
What does (F==B) denote?



Alternative - Leave a Deliberate Gap!

No need for `size` *field.*

Full Case : `(adv(B)==F)`



Code

```
class QueueAr implements Queue {

 private Object [] arr;
 private int front,back;
 private int maxSize;
 private final int initSize = 1000;
 private final int increment = 1000;

 public QueueAr()
      {arr = new Object[initSize]; front = 0; back=0; }

 public static Queue makeQueue() {return new QueueAr(); }

 public boolean isEmpty()      // check if queue is empty
    { return (front==back); }

 private boolean isFull()      // check if queue overflows
     { return (adv(back)==front); }
  }
```

More code

```
public void enqueue(Object o) // add o to back of queue
  { if (this.isFull()) this.enlarge();
    arr[back]=o;
    back=adv(back); }

 private void enlargeArr()        // enlarge the array
  {int newSize = maxSize+increment;
   Object [] barr = new Object[newSize];
   for (int j=0,k=front;j<=maxSize;j++,k=adv(k))
      {barr[j]=arr[k];};
   front=0; back=maxSize-1;
   maxSize = newSize; arr = barr;
  }
```

More code

```
public void dequeue() throws Underflow
  { if this.isEmpty()
       {throw new Underflow("dequeue fails – empty
   q");}
     else front=adv(front);
   }


public Object getFront() throws Underflow
 { if this.isEmpty()
      {throw new Underflow("getFront fails – empty
   q");}
     else return arr[front];
 }
```

Summary

- The definition of the queue operations gives the ADT queue first-in, first-out (FIFO) behavior
- The queue can be implemented by linked lists or by arrays
- There are many applications
  - Printer queues,
  - Telecommunication queues,
  - Simulations,
  - Etc.