CSC316

Data Structures for Computer Scientists

List Abstract Data Type

Dr. Jason King

DEFINITION

Abstract Data Type (ADT)

- model for behaviors from the point of view of a user of the data
- possible operations on data

Does not describe HOW the set of operations is implemented!

No rule for what operations must be supported → that is a design decision!

REVIEW

Barbara Liskov

- **Born**: November 7, 1939
- First described the concept of *abstract data types* in 1974
- Work led to development of *object-oriented programming*
- **Turing Award** in 2008

QUESTION

What is a list?

What are some behaviors that users can perform with lists?

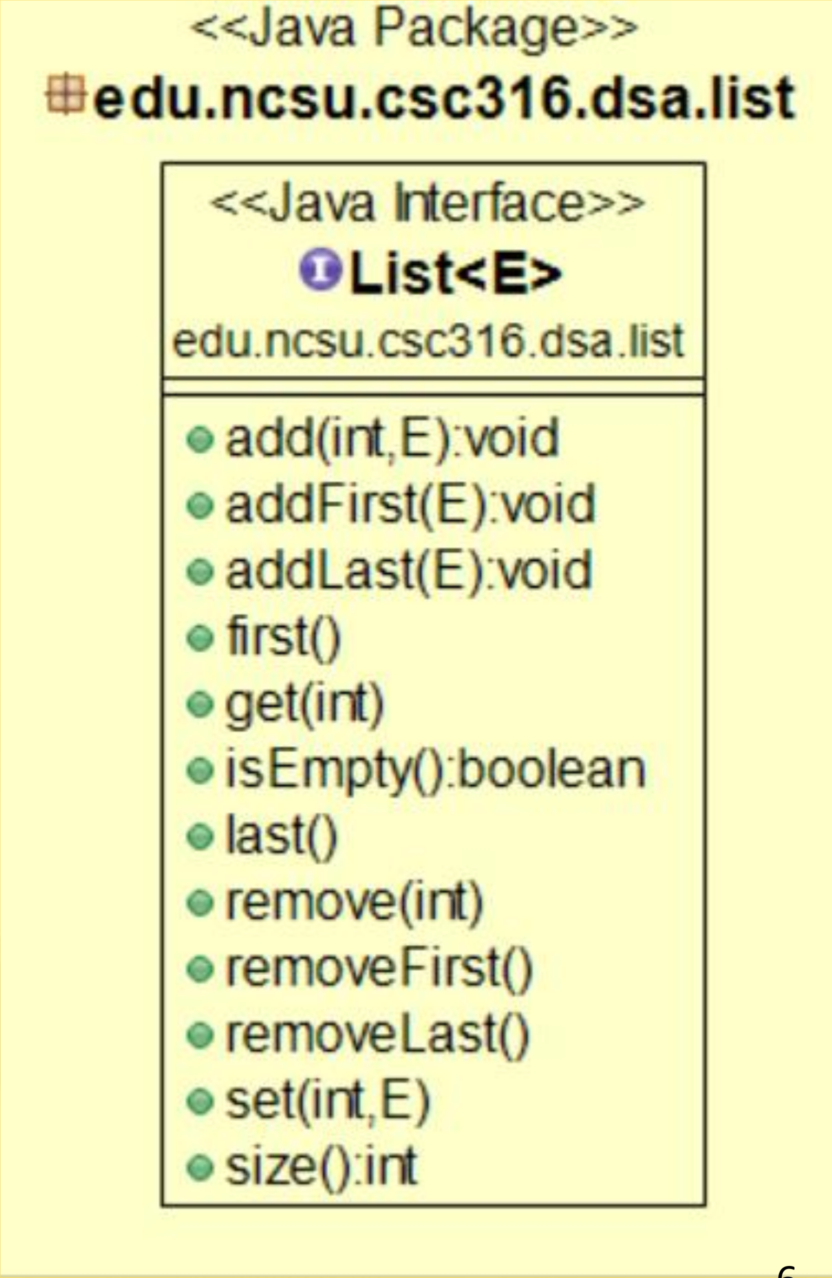
List

ABSTRACT DATA TYPE

Operation	**Description**
add(index,e)	Adds the element at the specified index in the list
addFirst(e)	Adds the element to the front of the list
addLast(e)	Adds the element to the end of the list
first()	Returns, but does not remove, the first element in the list
get(index)	Returns, but does not remove, the element at the
isEmpty()	Returns true if the list is empty; otherwise, returns false
last()	Returns, but does not remove, the last element in the list
remove(index)	Removes and returns the element at the specified index
removeFirst()	Removes and returns the first element in the list
removeLast()	Removes and returns the last element in the list
set(index,e)	Replaces the element at the specified index with the new element e
size()	Returns the number of elements in the list

IMPORTANT

- You can think about ADTs as being similar to Java *interfaces*
  - They define behaviors/actions, but do not provide concrete implementations of those behaviors!

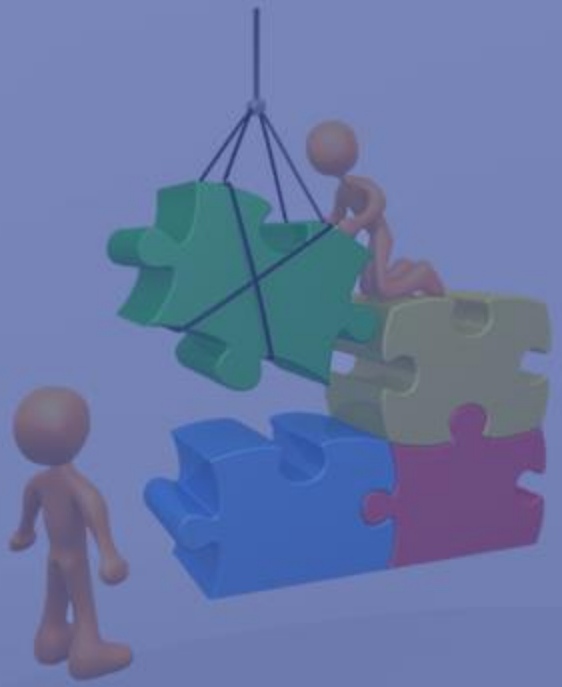


Algorithms & ADTs

- Use ADT behaviors when documenting algorithms
- You can then choose efficient data structures based on the ADT behaviors that are used within the algorithm

```
Algorithm doubleEvensAndRemoveOdds(L)
Input a List, L, of n integers
Output the list of even integers in L, but doubled
F ← new empty List
while NOT L.isEmpty() do
    value ← L.removeFirst()
    if value % 2 = 0
        F.addLast( value × 2)
return F
```

Choose a **List** data structure that *most* efficiently implements these behaviors



DATA STRUCTURE

- **Array-based list**
  - *Contiguous memory representation*
  - List elements stored in an array
  - Adjacency in the array → adjacency in the list
- **Linked list**
  - *Linked memory representation*
  - List elements can be scattered arbitrarily in memory
  - List elements reference neighbor(s)

Array-Based List

Array-Based List

- **Benefits**
  - Access to each element in O(1) Time
  - Memory overhead: Only list data elements are stored
- **Problems**
  - Array sizes are fixed
    - Memory waste, not for dynamic applications
  - Inserting/removing at the middle
    - Massive data movement: O(n) time

Inserting into an Array-based List

Problem

- Array size is fixed!
- Run out of array space

Solution

- Grow the array!
- … but how does this affect runtime?

```
Algorithm addLast(D, e)
Input an array of elements D
      an element e
Output the list with the element added
ensureCapacity(D, size() + 1)
D[size()] ←e
```

```
Algorithm ensureCapacity(D, newCapacity)
Input an array of elements D
      the new capacity that must be supported
Output an array that supports the newCapacity
if newCapacity > length(D) then
   T ← new array with new length of ???
for i ← 0 to n-1 do
   T[i] ← D[i]
D ← T
```

Analyzing ensureCapacity

```
Algorithm ensureCapacity(D, newCapacity)
Input an array of elements D
      the new capacity that must be supported
Output an array that supports the newCapacity
if newCapacity > length(D) then
   T ← new array with new length of ???
for i ← 0 to n-1 do
   T[i] ← D[i]
D ← T
```

O(1)

O(n)

O(n)

- → `ensureCapacity` is O(n) … so is `addLast(e)` also O(n)?
- → Stating the worst-case may be too pessimistic and not representative of the true runtime efficiency since growing the array does not happen on *every* addLast
- → Let's consider what happens over a *series* of `addLast(e)` operations…

Amortized Cost

- ***business definition*:**
  - to gradually reduce or write off the cost or value of (as an asset)
- ***computer science definition*:**
  - to gradually reduce the cost of an operation over a sequence of operations

- **General Idea:**
  - some expensive operations may “pay” for future operations by somehow limiting the number or cost of expensive operations that can happen in the near future

Amortized Analysis

- Is **NOT**:

  “The amortized cost of adding a value to the end of an array-based list is O(1), so when I insert '45' into this list, the cost will be O(1).”

- Instead, **amortized cost is**:
- *“The amortized cost of adding a value to the end of an array-based list is O(1); therefore, the total running time of adding n values to the end of an array-based list is O(n)”*

Analyzing addLast(e)

→ What is the **amortized cost?**

→ Start with 1 element

→ We can determine $\frac{T(n)}{n}$ to find amortized cost

→ This is ***not*** average cost!

→ Average cost is in terms of a *single* operation, not a series of operations

```
Algorithm mystery(D, n)
Input an array D of integers
      the number of elements to add to D
Output the list with m elements added
for i ← 0 to n-1 do
   addLast(E, i)
```

```
Algorithm addLast(D, e)
Input an array of elements D
      an element e
Output the list with the element added
ensureCapacity(D, size() + 1)
D[size()] ←e
```

Growing an Array

Let's consider two strategies:

- **Incremental:** grow the array by a constant amount each time

	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c

- **Doubling:** grow the array by doubling the length each time

	c	c		c				c							

Analysis: Incremental Strategy

How many times do we grow the following array of 16 elements?

	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c

→ We grow $\left\lceil \frac{n-1}{c} \right\rceil$ times, where *n* = # elements and *c* = 1= how much we grow each time

→ Then the runtime of `addLast(e)` over n elements:

$$T(n) = 1 + \boxed{(c+1)} + (2c+1) + (3c+1) + \cdots + ((n-1)c+1)$$

Copy 1, then we can add 1 more before having to copy again

$$= (1 + 1 + 1 + \cdots + 1) + c(1 + 2 + 3 + 4 + \cdots + (n-1))$$

$$= n + c(1 + 2 + 3 + 4 + \cdots + (n-1))$$

$$= n + c\sum_{i=1}^{n-1} i = n + c\frac{(n-1)(n)}{2} = n + c\frac{n^2-n}{2}$$

→T(n) is O(n^2)

→**Amortized cost of addLast** <u>over *n* operations</u>: $\frac{T(n)}{n} = \frac{n^2}{n} = n$ → O(n) each

Analysis: Doubling Strategy

How many times do we grow the following array of 16 elements?

→ We grow where $2^c = n \;\; \rightarrow c = \lceil \log_2 n \rceil$ times, where *n* = # of elements

→ the 1st time we grow (c=1), we end up with capacity of 2 → $2^c = 2$

→ the 2nd time we grow (c=2), we end up with capacity of 4 → $2^c = 4$

→ the 3rd time we grow (c=3), we end up with capacity of 8 → $2^c = 8$

→ the 4th time we grow (c=4), we end up with capacity of 16 → $2^c = 16$

→ The runtime of `addLast(e)` over n elements is:

$$\mathrm{T(n)} = 1 + \boxed{(1+1)} + (2+2) + (4+4) + \cdots + (2^{c-1} + 2^{c-1})$$

Copy 1, then we can add 1 more before having to copy again

Analysis: Doubling Strategy

$$\sum_{i=a}^{b} c^i = \begin{cases} \dfrac{c^a - c^{b+1}}{1-c} & \text{if } c \neq 1 \\ b - a + 1 & \text{if } c = 1 \end{cases}$$

How many times do we grow the following array of 16 elements?

→ We grow where $2^c = n \;\; \rightarrow c = \lceil \log_2 n \rceil$ times, where *n* = # of elements

→ The runtime of `addLast(e)` over n elements is:

→ $\mathrm{T(n)} = 1 + (1+1) + (2+2) + (4+4) + \cdots + (2^{c-1} + 2^{c-1})$

$$= 1 + 2 \times \sum_{i=0}^{c-1} 2^i$$

$$= 1 + 2 \times \frac{2^0 - 2^c}{1-2} = 1 + 2 \times (2^c - 1) = 1 + 2 \times 2^c - 2 = -1 + 2 \times 2^{\log_2 n}$$

$$= -1 + 2 \times n = 2n - 1$$

→T(n) is O(n)

→**Amortized Cost of addLast** <u>over *n* operations</u> is $\dfrac{T(n)}{n} = \dfrac{n}{n} = 1$ → O(1) each

IMPORTANT

- Amortized cost is NOT the same as average cost
  - amortized cost is over a *series* of operations

- **Average cost example:**
  - lookUp in an array-based list requires at least 1 comparison, at most n comparisons
  - An average of (n/2) comparisons
  - Average cost of lookUp is O(n)

Linked Lists

Linked-Memory Representation

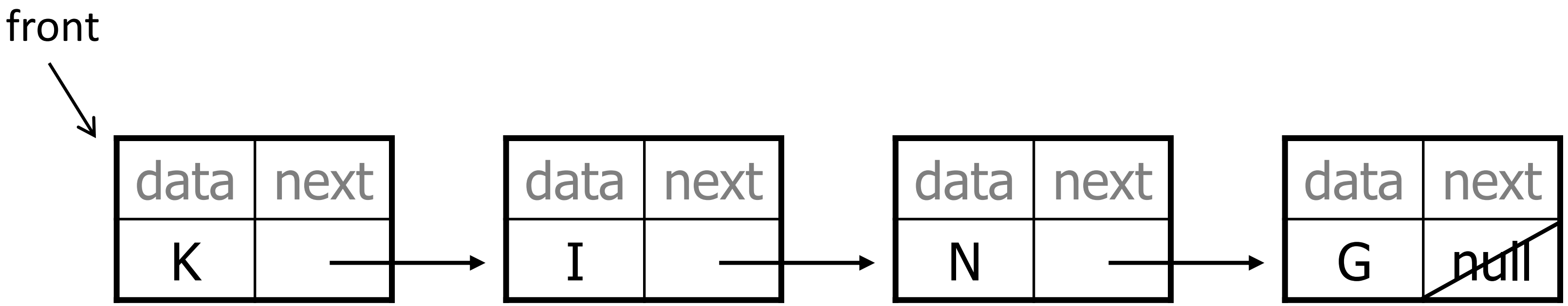
- **Benefits**
  - Insertions/deletions do not involve moving/shifting
  - Dynamic allocation of memory on an as-needed basis
- **Problems:**
  - No random access
    - Access time is $O(n)$

Singly Linked List

A data structure consisting of a sequence of **nodes**

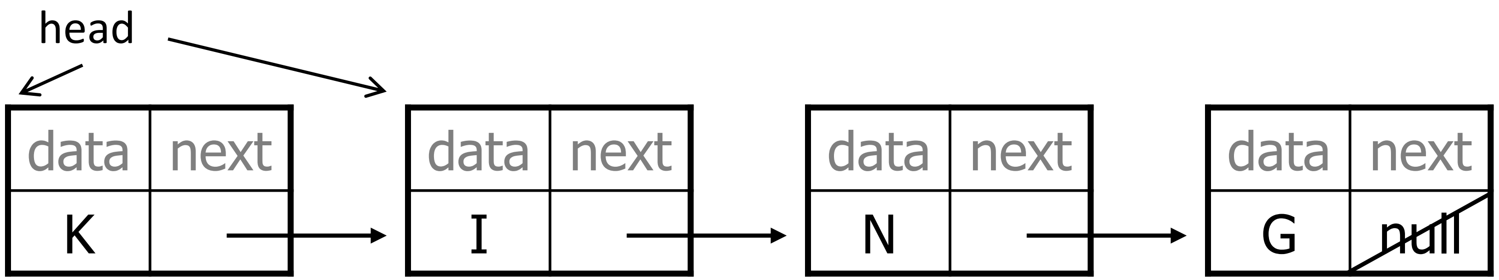
Each node stores

- A data element
- A link to the **next** node



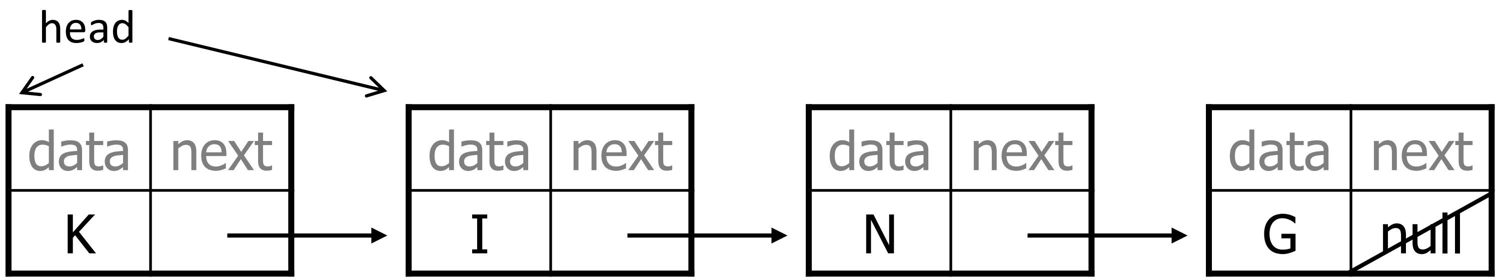
Inserting at the Head

- Allocate a new node
  - Insert the new data element
  - Have the new node point to the old head
- Update head to point to the new node

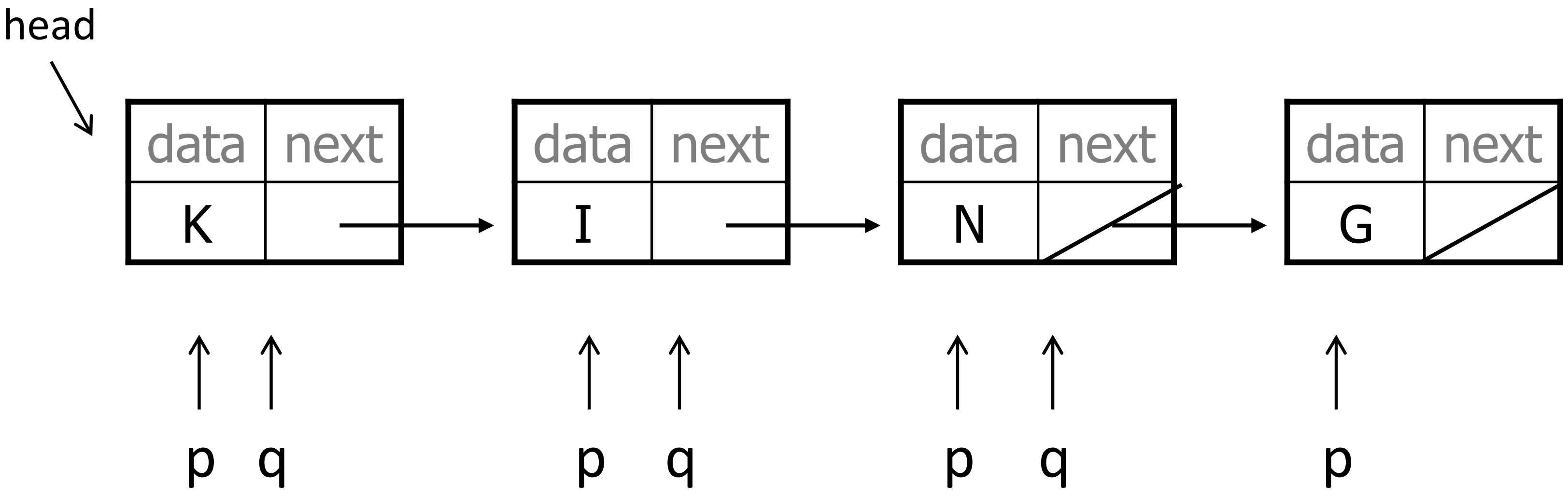


Removing at the Head

- Update head to point to the next node in the list
- Allow garbage collector to reclaim former first node



Inserting at the End of the List



Algorithm: insert at the end

```
Algorithm insert(head, x)
   Input the head node,
         the element x to insert
   p ← head
   if p = null then
      head ← newNode(x, null)
   else
      p ← next(head)
      q ← head
      while p ≠ null do
         p ← next(p)
         q ← next(q)
      next(q) ← newNode(x, null)
```

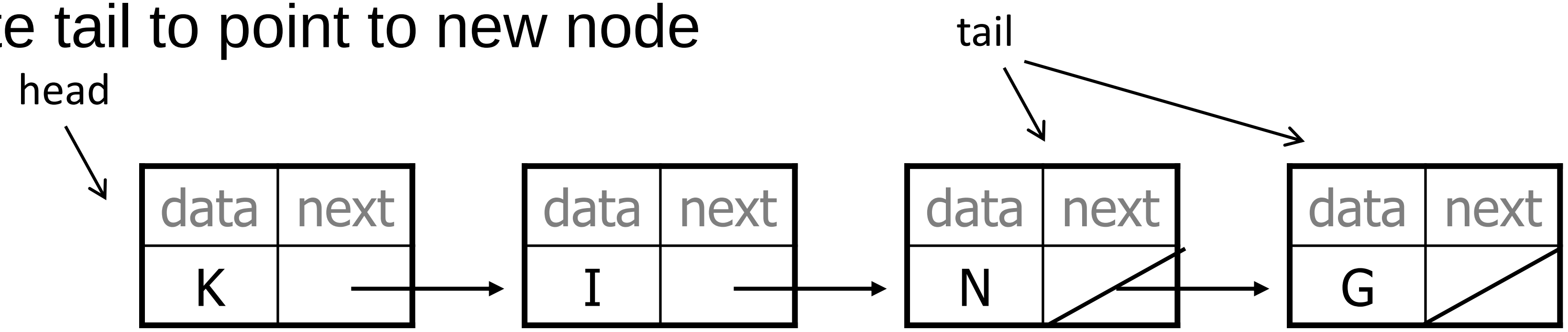
Running Time *T(n)* is

QUESTION

How can we improve the worst-case running time of inserting at the end of a singly-linked list?

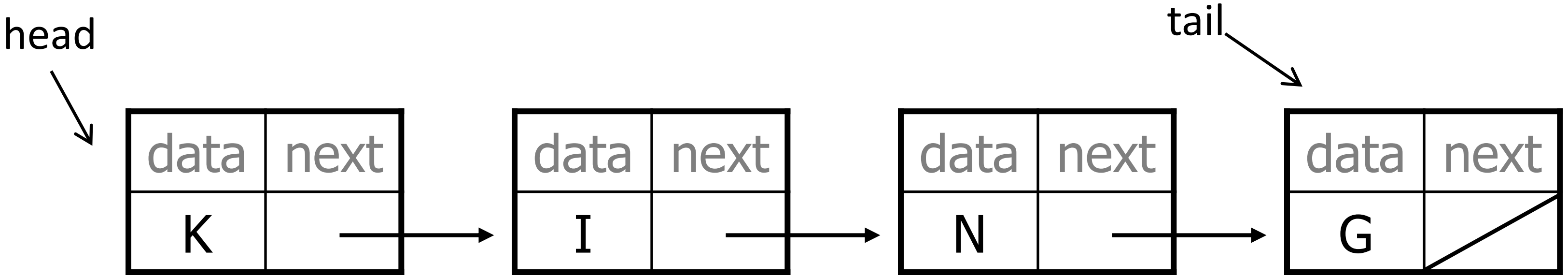
Inserting at the End with a Tail Pointer

- Allocate a new node
  - Insert the new data element
  - Have new node point to null
- Have old last node point to new node
- Update tail to point to new node



Removing at the End with Tail

- Removing at the tail is **not** efficient
- No constant-time way to update tail to point to the previous node



DEFINITION

Dummy "sentinel" node

- No data element stored → null value
- Next points to the *true* beginning of the list

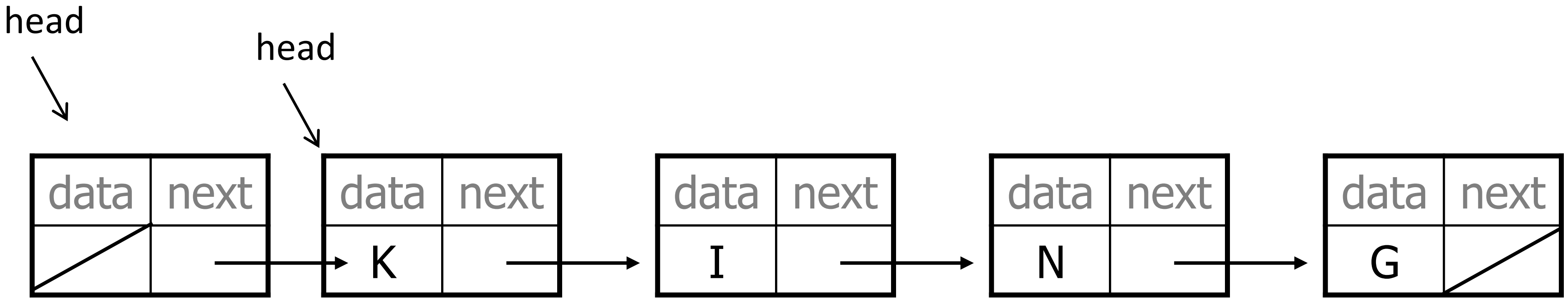
Advantages

- Code for operations more uniform
- Easier to write recursive versions of operations

Singly Linked List w/ Dummy Head

Dummy head node

- Next points to the *true* beginning of the list

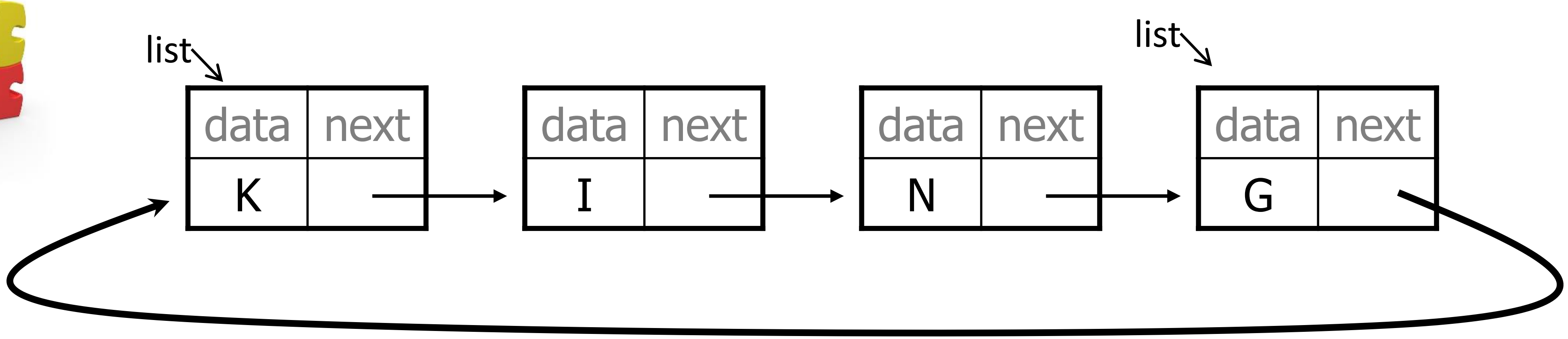


Example: insert(node, element)

- Linked list with <u>dummy head node</u>
- <u>List is sorted</u> in ascending order of elements
- List is accessed by head pointer, <u>no tail pointer</u>

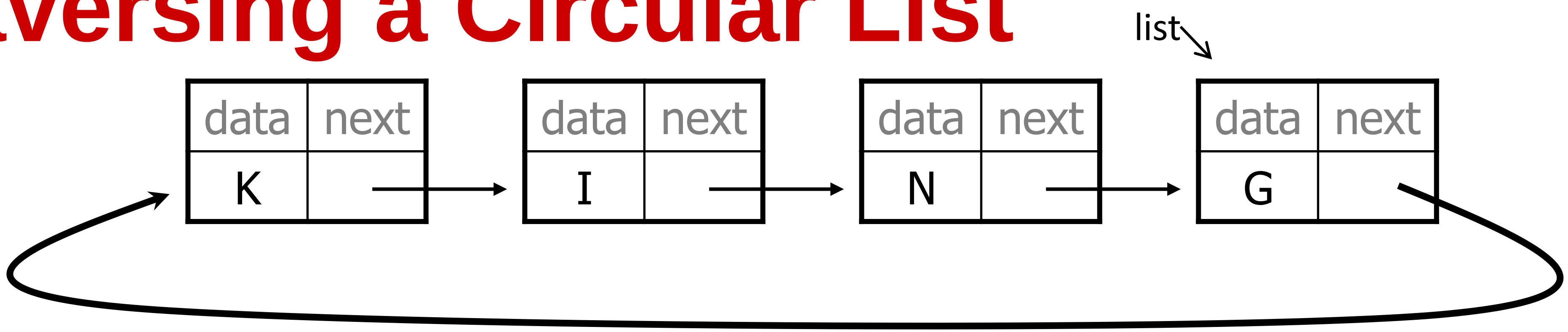
```
Algorithm insert(head, x)
   Input the head node, the element x to insert
   insertHelper(head, x)

Algorithm insertHelper(p, x)
   Input a node p, the element x to insert
   if next(p) = null then
      next(p) ← newNode(x, null)
   else if element(next(p)) ≥ x then
      next(p) ← newNode (x, next(p))
   else insertHelper(next(p),x)
```

Circular Linked List



- Where does the “pointer” go?
  - At the **front**?
    - Insert at the front:
    - Insert at the tail:
  - At the **tail**?
    - Insert at the front
    - Insert at the tail:
- Frequently used to implement queues

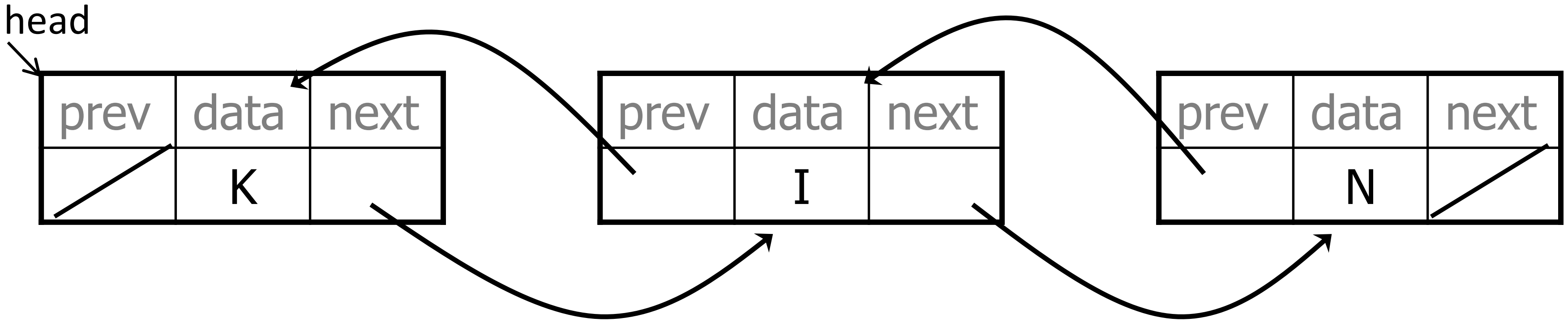
Traversing a Circular List




```
Algorithm traverse(list)
    Input a pointer list to the tail of the list
    if list ≠ null then
        p ← next(list)
        visit(p)
        while p ≠ list do
            p ← next(p)
            visit(p)
```

T(n) is O(n)

Doubly Linked Lists



Advantages

- Easy insertions/deletions
- Backtracking is easy!

Disadvantages

- – O(n) extra pointers

REVIEW

Contiguous memory list model

- Array-based list
  - Efficient insertions at the end of the list: O(1) amortized cost when doubling array
  - Requires costly shifts if inserting elsewhere
  - Direct access to any random index

Linked memory list model

- Linked list
  - Efficient insertions at the beginning of the list: O(1)
  - Can achieve efficient insertions at the end with tail pointer
  - Must traverse from head or tail to access random index

Operation	ArrayList	Singly LinkedList with Head	Singly LinkedList with Head & Tail	Circularly LinkedList with Tail	Doubly LinkedList with Head & Tail
add(index, e)					
addFirst(e)					
addLast(e)					
first()					
get(index)					
isEmpty()					
last()					
remove(index)					
removeFirst()					
removeLast()					
set(index, e)					
size()					

** assuming size is maintained as a field*

Operation	ArrayList	Singly LinkedList with Head	Singly LinkedList with Head & Tail	Circularly LinkedList with Tail	Doubly LinkedList with Head & Tail
add(index, e)	O(n)				
addFirst(e)	O(n)				
addLast(e)	O(1)*				
first()	O(1)				
get(index)	O(1)				
isEmpty()	O(1)				
last()	O(1)*				
remove(index)	O(n)				
removeFirst()	O(n)				
removeLast()	O(1)*				
set(index, e)	O(1)				
size()	O(1)*				

Amortized Costs for adding, assuming we double the length of the array each time we grow

** assuming size is maintained as a field*

Operation	ArrayList	Singly LinkedList with Head	Singly LinkedList with Head & Tail	Circularly LinkedList with Tail	Doubly LinkedList with Head & Tail
add(index, e)	O(n)	O(n)	O(n)	O(n)	O(n)
addFirst(e)	O(n)	O(1)	O(1)	O(1)	O(1)
addLast(e)	O(1)*	O(n)	O(1)	O(1)	O(1)
first()	O(1)	O(1)	O(1)	O(1)	O(1)
get(index)	O(1)	O(n)	O(n)	O(n)	O(n)
isEmpty()	O(1)	O(1)	O(1)	O(1)	O(1)
last()	O(1)*	O(n)	O(1)	O(1)	O(1)
remove(index)	O(n)	O(n)	O(n)	O(n)	O(n)
removeFirst()	O(n)	O(1)	O(1)	O(1)	O(1)
removeLast()	O(1)*	O(n)	O(n)	O(n)	O(1)
set(index, e)	O(1)	O(n)	O(n)	O(n)	O(n)
size()	O(1)*	O(1)*	O(1)*	O(1)*	O(1)*

** assuming size is maintained as a field*

Choosing a Data Structure

```
Algorithm doubleEvensAndRemoveOdds(L)
Input a List, L, of n integers
Output the list of even integers in L, but doubled
F ← new empty List
while NOT L.isEmpty() do
    value ← L.removeFirst()
    if value % 2 = 0
        F.addLast( value × 2)
return F
```

Operation	ArrayList	Singly LinkedList with Head	Singly LinkedList with Head & Tail	Circularly LinkedList with Tail	Doubly LinkedList with Head & Tail
add(index, e)	O(n)	O(n)	O(n)	O(n)	O(n)
addFirst(e)	O(n)	O(1)	O(1)	O(1)	O(1)
addLast(e)	O(1)*	O(n)	O(1)	O(1)	O(1)
first()	O(1)	O(1)	O(1)	O(1)	O(1)
get(index)	O(1)	O(n)	O(n)	O(n)	O(n)
isEmpty()	O(1)	O(1)	O(1)	O(1)	O(1)
last()	O(1)*	O(n)	O(1)	O(1)	O(1)
remove(index)	O(n)	O(n)	O(n)	O(n)	O(n)
removeFirst()	O(n)	O(1)	O(1)	O(1)	O(1)
removeLast()	O(1)*	O(n)	O(n)	O(n)	O(1)
set(index, e)	O(1)	O(n)	O(n)	O(n)	O(n)
size()	O(1)*	O(1)*	O(1)*	O(1)*	O(1)*

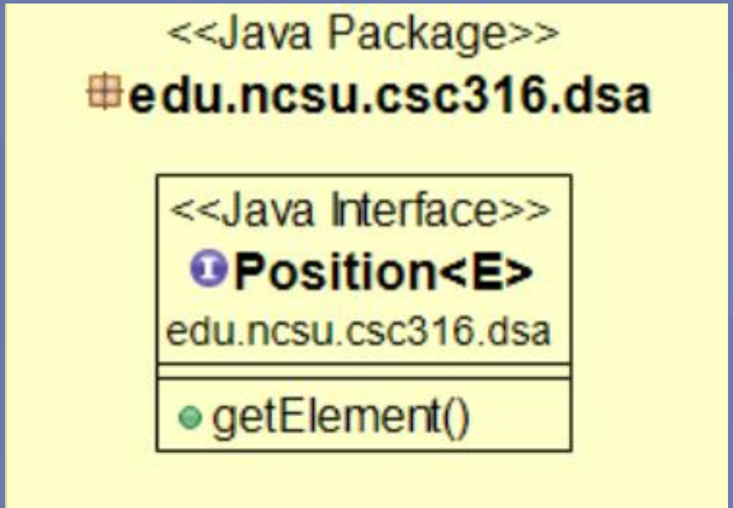
** assuming size is maintained as a field*

Positional List Abstract Data Type

DEFINITION

Positional List

- Special abstraction of a list that has the ability to identify the location/**position** of an element
- A **position** is any marker/node within the list
  - *position is the entire "node", not just the element*
- Behaviors are **position**-based, not index-based

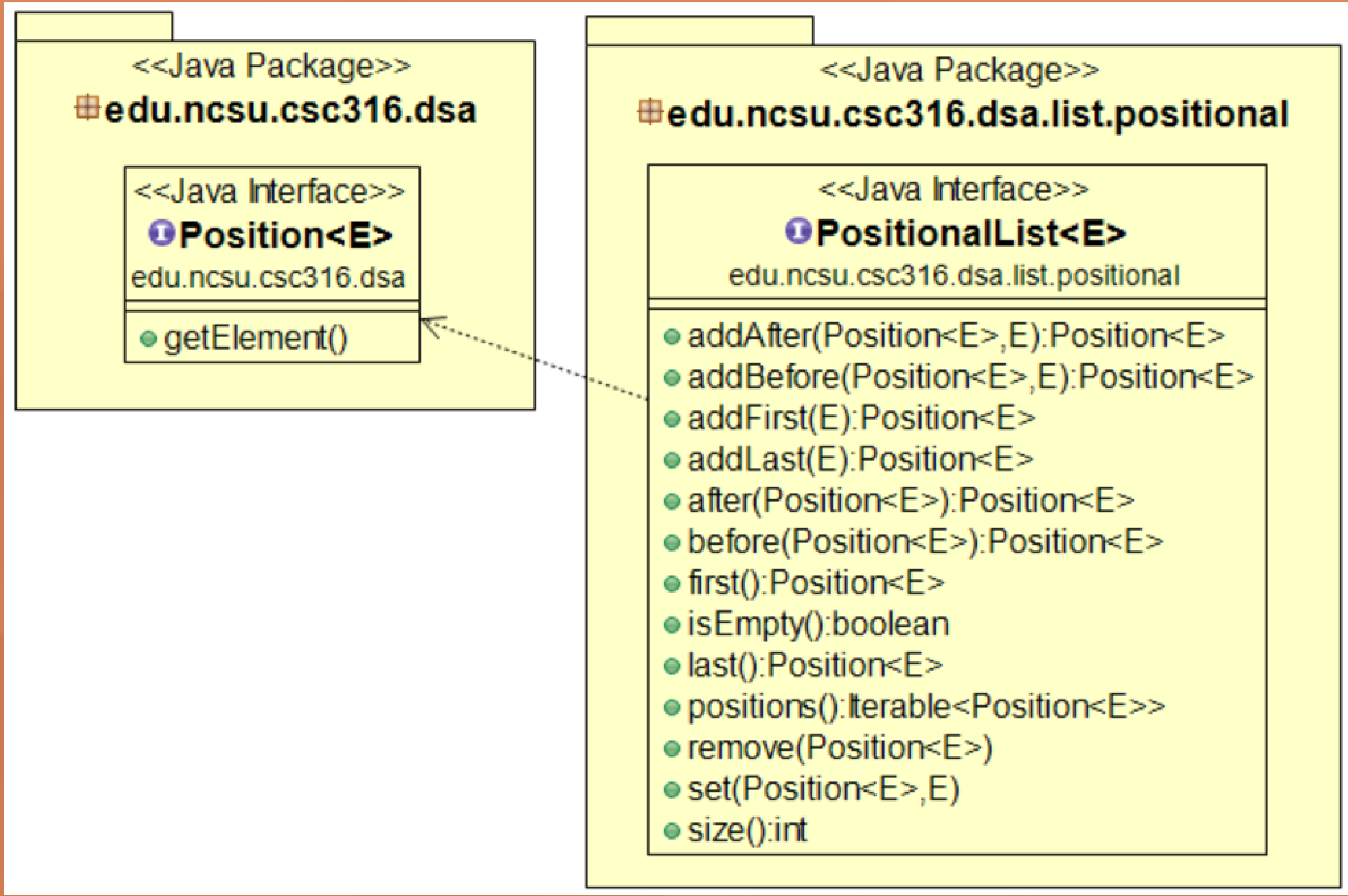


Positional List

ABSTRACT DATA TYPE

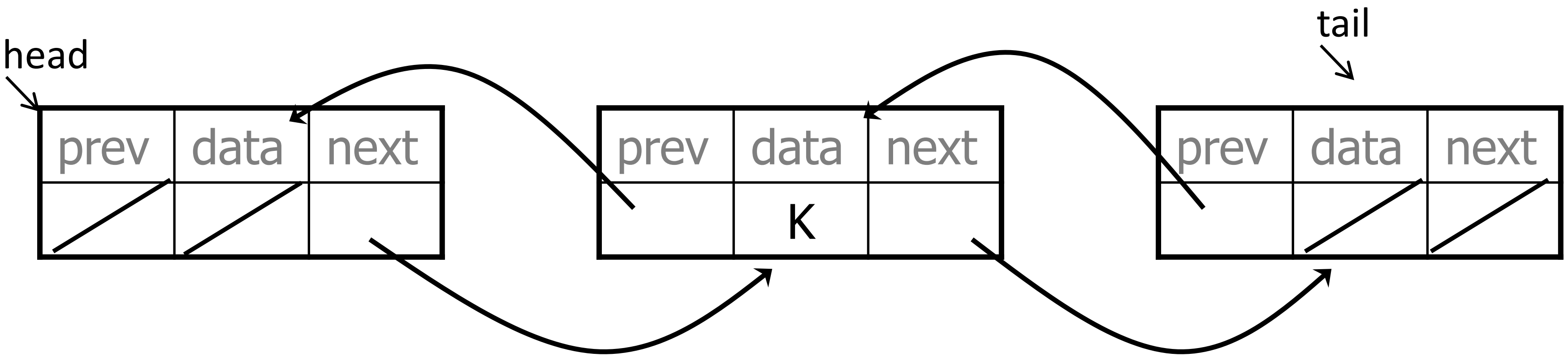
Operation	Description
first()	Returns, but does not remove, the position of the first element in the list
last()	Returns, but does not remove, the position of the last element in the list
before(p)	Returns the position immediately before the given position in the list
after(p)	Returns the position immediately after the given position in the list
addFirst(e)	Adds a new element at the front of the list, and returns the position of the new element
addLast(e)	Adds a new element at the end of the list, and returns the position of the new element
addBefore(p, e)	Adds a new element immediately before position p, and returns the position of the new element
addAfter(p, e)	Adds a new element immediately after position p, and retusn the position of the new element
set(p, e)	Replaces the element at position p, and returns the original element at the position
remove(p)	Removes and returns the element at position p (eliminating the position)
size()	Returns the number of elements in the list
isEmpty()	Returns true if the list is empty; otherwise, returns false

ABSTRACT DATA TYPE



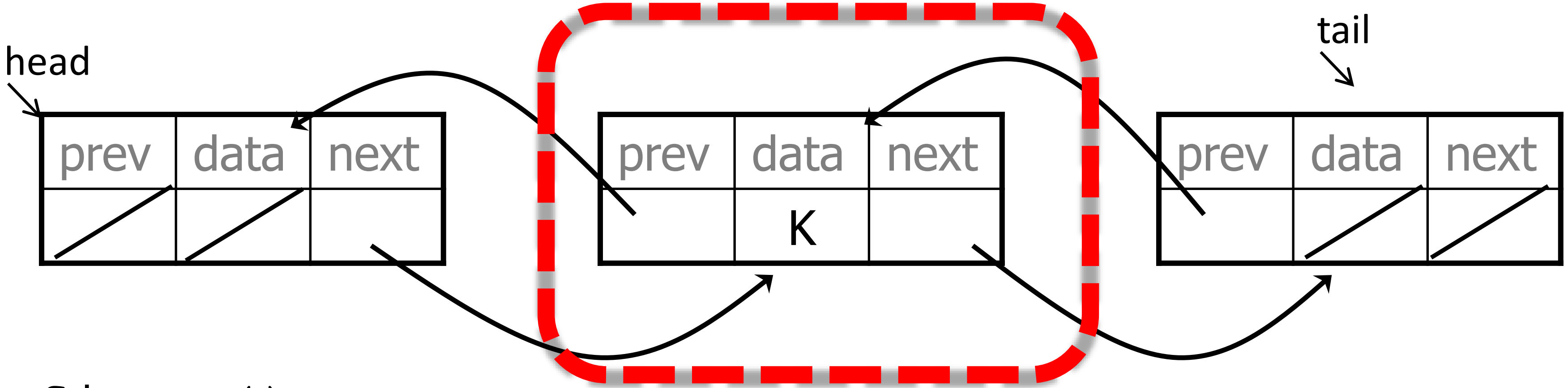
Positional Linked List

To efficiently support `before()`, `addBefore()`, etc., implement as a **doubly-linked list** with dummy sentinel head and tail nodes.



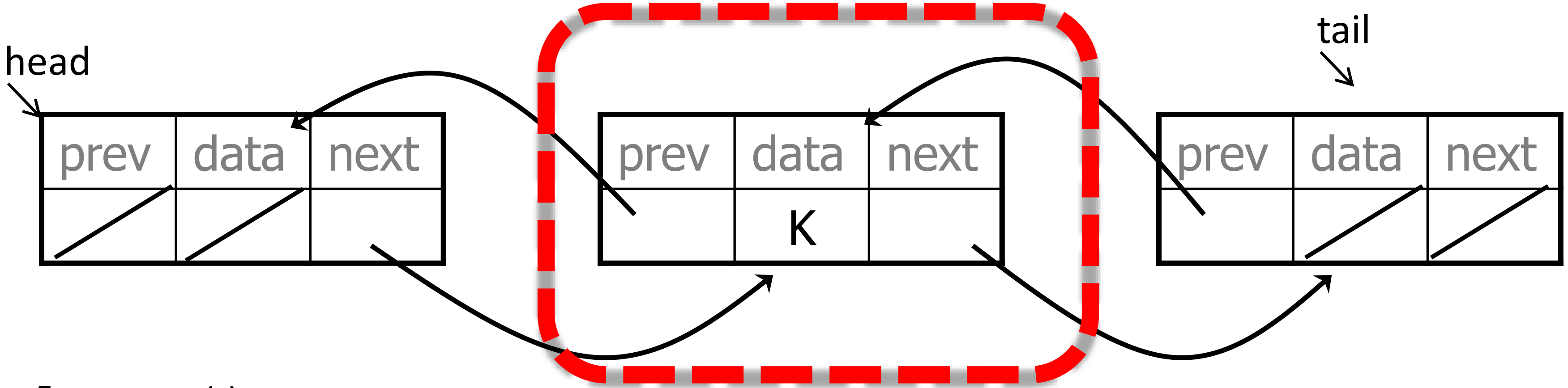
Returned values are the **positions**, not the *data elements* at each position.

Positional Linked List Example



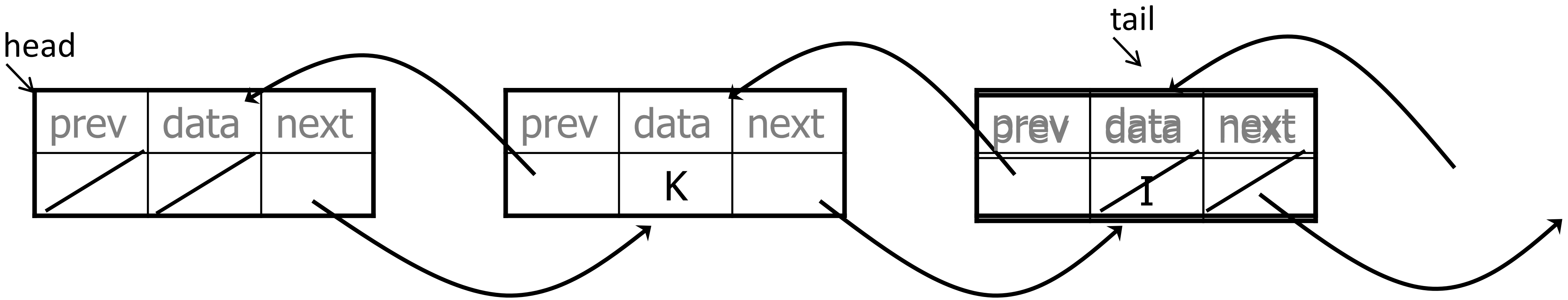
→ `PL.first()`

Positional Linked List Example



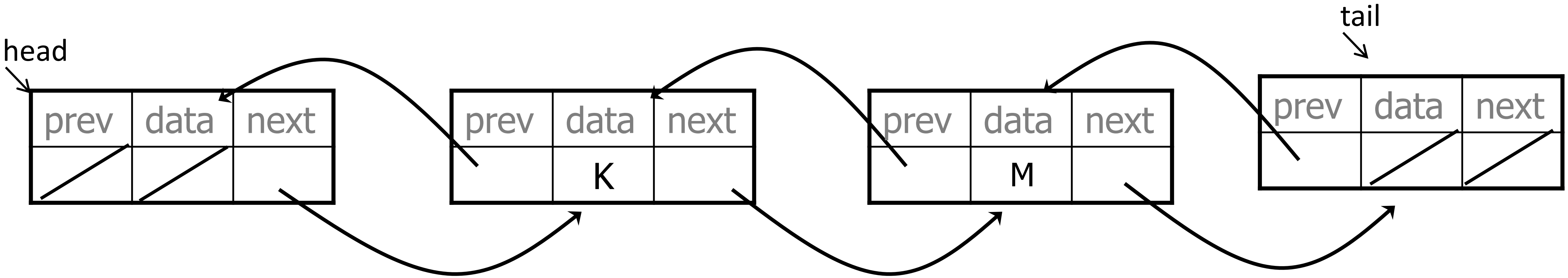
→ `PL.last()`

Positional Linked List Example



`Position p ← PL.addAfter(PL.last(), "I")`

Positional Linked List Example




```
Position p ← PL.addAfter(PL.last(), "I")
PL.set(PL.after(PL.before(PL.after(PL.before(p)))),"M")
```

REVIEW

Positions in a positional list do not change

→ **not** index-based like traditional lists

→ no index-based operations in the ADT

Positional List ADT operations return positions, instead of *elements* at those positions.

REVIEW

Operation	Doubly Linked Positional List
first()	
last()	
before(p)	
after(p)	
addFirst(e)	
addLast(e)	
addBefore(p, e)	
addAfter(p, e)	
set(p, e)	
remove(p)	
size()	
isEmpty()	

** assuming size is maintained as a field*

Iterators

Iterators

- Provide a common, efficient way to traverse <u>many</u> types of data structures
- For lists, keep track of the *current* position in the list
  - In Java, iterators are required if you want to use `for-each` loops

```
for i ← 0 to n-1 do
    E temp ← list.get(i)
```

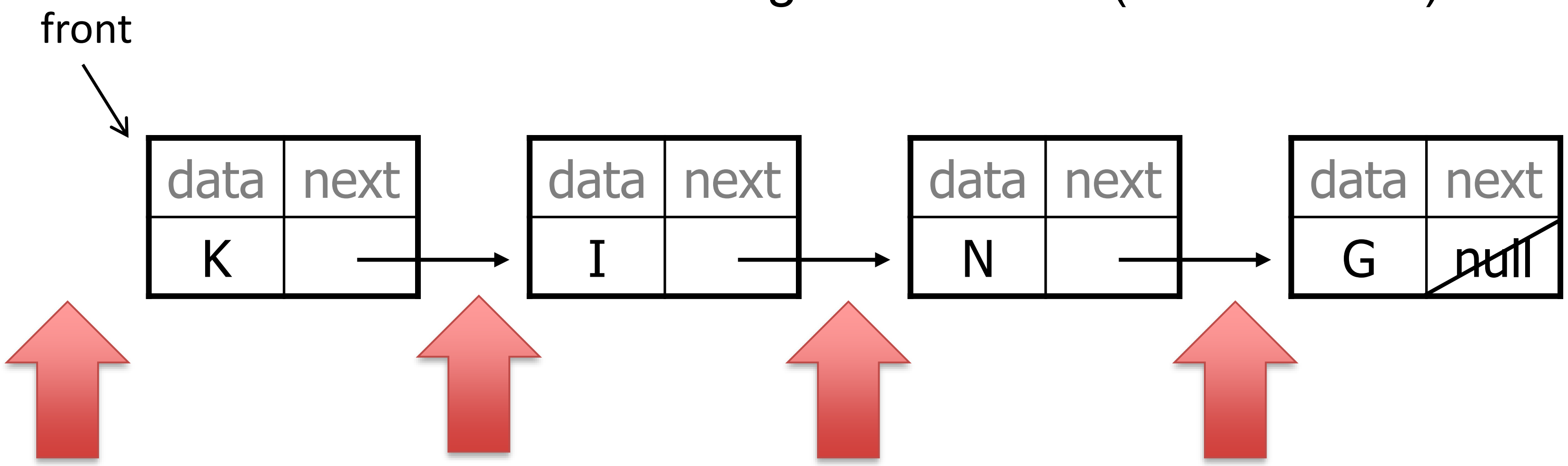
$O(n^2)$ for linked lists

```
while it.hasNext() do
    E temp ← it.next()
```

O(n) for all lists

Iterator Reminders

- Think of the cursor as being "between" elements in lists
- The remove() operation deletes the element that was last/previously returned by next()
  - Cannot perform more than 1 single remove() at a time
    - Need to track whether it's legal to remove ("removeOK")



Iterators in Algorithms

- In pseudocode, you can *imply* the use of iterators through “for-each” style loops

```
Algorithm sum(L)
Input a List, L, of n integer elements
Output the sum of elements in L
value ← 0
for each element e in L do
    value ← value + e
return value
```

REVIEW

Review CSC216 Materials on Iterators

- Dr. Heckman's Lecture Materials
- Dr. Schmidt's Lecture Materials

- Do not confuse simple `Iterator`s with the more advanced Java `ListIterators` you implemented in CSC216!

Remember you are allowed to reference **AND CITE** the CSC316 textbook