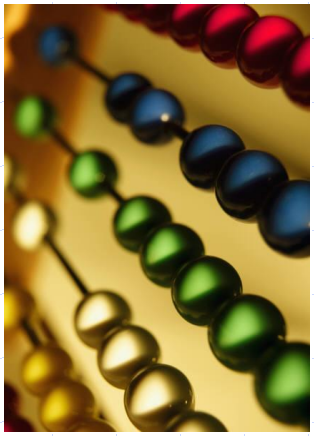
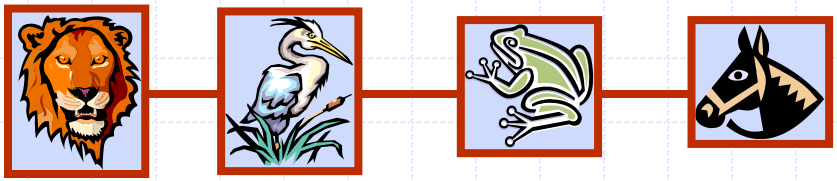
Pre-Course Discussion

- ADTs, Why study DS and Algs?, OOP Basics
- Questions/Feedback about quiz and PS?
- Asymptotic Runtime Analysis, Big-Oh Notation
- Recursion

Modified version of the presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

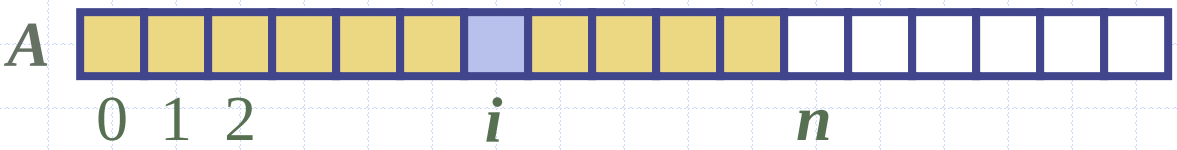
Arrays, Lists and Iterators





Array Definition

- An ***array*** is a sequenced collection of variables all of the same type*. Each variable, or ***cell***, in an array has an ***index***, which uniquely refers to the value stored in that cell. The cells of an array, *A*, are numbered 0, 1, 2, and so on.
- Each value stored in an array is often called an ***element*** of that array.



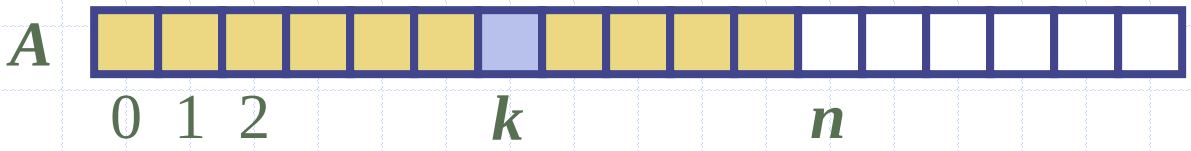
*Does not have to be the case but it is true for Java, C, C++ etc.

Arrays

- Basic Operations:
  - Setting a value
  - Getting (retrieving) a value
- Properties
  - Each element behaves like a **variable**
  - Elements with distinct indices are **disjoint**

Array Length and Capacity

- Since the length of an array determines the maximum number of things that can be stored in the array, we will sometimes refer to the length of an array as its ***capacity***.
- In Java, the length of an array named *a* can be accessed using the syntax ***A*.length**. Thus, the cells of an array, *A*, are numbered 0, 1, 2, and so on, up through *A*.length−1, and the cell with index *k* (< *A*.length) can be accessed with syntax *a*[*k*].



Declaring Arrays (first way)

- The first way to create an array is to use an assignment to a literal form when initially declaring the array, using a syntax as:

$$elementType[]\ arrayName = \{initialValue_0, initialValue_1, \ldots, initialValue_{N-1}\};$$

- The *elementType* can be any Java base type or class name, and *arrayName* can be any valid Java identifier. The initial values must be of the same type as the array.

Declaring Arrays (second way)

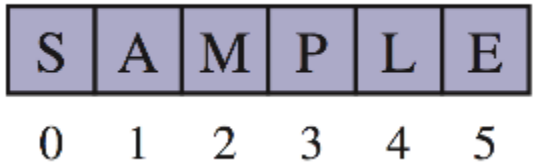
- The second way to create an array is to use the **new** operator.
  - However, because an array is not an instance of a class, we do not use a typical constructor. Instead we use the syntax:

new *elementType*[*length*]

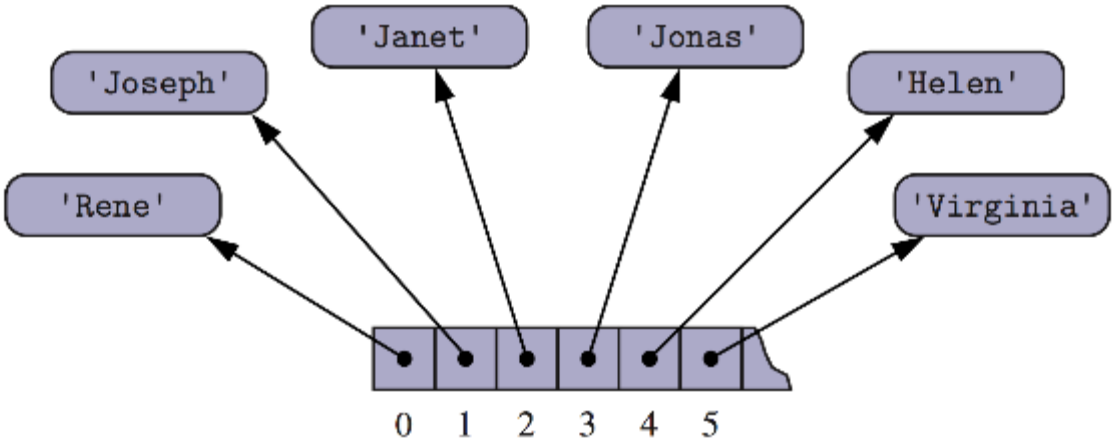
- *length* is a positive integer denoting the length of the new array.
- The **new** operator returns a reference to the new array, and typically this would be assigned to an array variable.

Arrays of Primitives or Object References

- An array can store primitive (base types) elements, such as characters.



- An array can also store references to objects.



Lists

- Countable number of ordered values
- In addition to setting and getting, an element can be inserted or removed by specifying its index (number of elements preceding it)
- Applications:
  - Direct usage as ordered collection of objects
  - Auxiliary data structure for algorithms
  - Component of other data structures

The java.util.List ADT

- The java.util.List interface includes the following methods:

size( ):	Returns the number of elements in the list.
isEmpty( ):	Returns a boolean indicating whether the list is empty.
get(i):	Returns the element of the list having index i; an error condition occurs if i is not in range $[0, \text{size}( ) - 1]$.
set(i, e):	Replaces the element at index i with e, and returns the old element that was replaced; an error condition occurs if i is not in range $[0, \text{size}( ) - 1]$.
add(i, e):	Inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list; an error condition occurs if i is not in range $[0, \text{size}( )]$.
remove(i):	Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error condition occurs if i is not in range $[0, \text{size}( ) - 1]$.

main methods

Example

- A sequence of List operations:

Method	Return Value	List Contents
add(0, A)		
add(0, B)		
get(1)		
set(2, C)		
add(2, C)		
add(4, D)		
remove(1)		
add(1, D)		
add(1, E)		
get(4)		
add(4, F)		
set(2, G)		
get(2)		

Example

- A sequence of List operations:

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)		
add(2, C)		
add(4, D)		
remove(1)		
add(1, D)		
add(1, E)		
get(4)		
add(4, F)		
set(2, G)		
get(2)		

Example

- A sequence of List operations:

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	"error"	(B, A)
add(2, C)		
add(4, D)		
remove(1)		
add(1, D)		
add(1, E)		
get(4)		
add(4, F)		
set(2, G)		
get(2)		

Example

- A sequence of List operations:

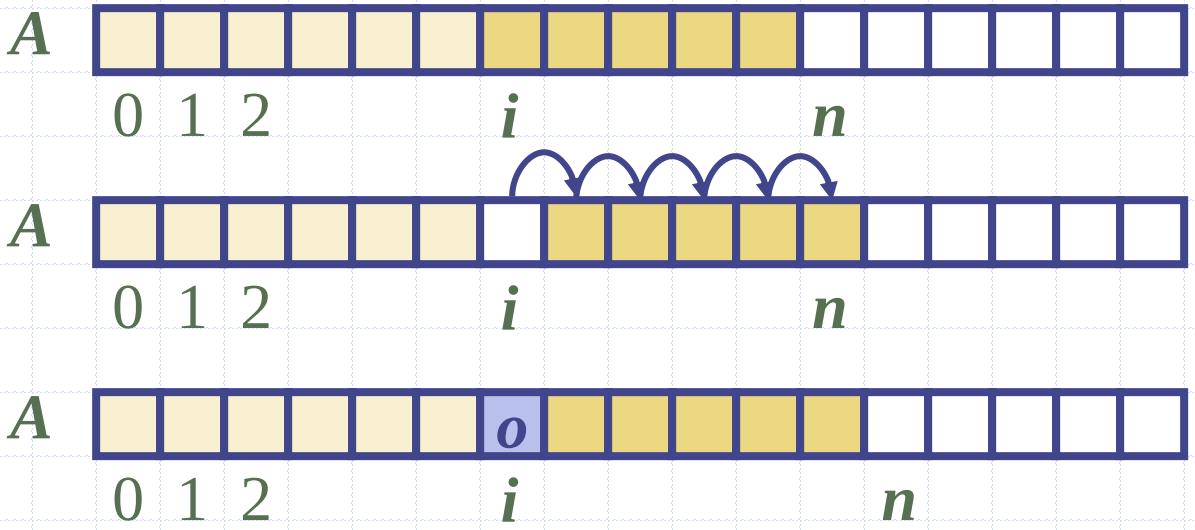
Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	"error"	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	"error"	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	"error"	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

List Implementations

- **Array Lists**

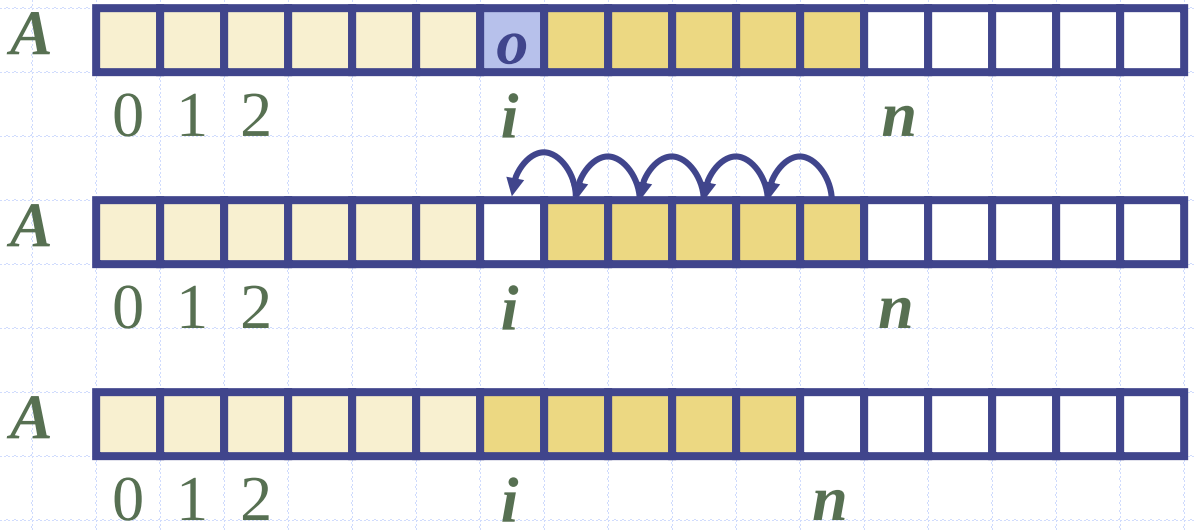
Adding an Element (Insertion)

- In an operation $add(i, o)$, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Removing an Element

- In an operation ***remove***(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Java Implementation

```
11    // public methods
12    /** Returns the number of elements in the array list. */
13    public int size( ) { return size; }
14    /** Returns whether the array list is empty. */
15    public boolean isEmpty( ) { return size == 0; }
16    /** Returns (but does not remove) the element at index i. */
17    public E get(int i) throws IndexOutOfBoundsException {
18      checkIndex(i, size);
19      return data[i];
20    }
21    /** Replaces the element at index i with e, and returns the replaced element. */
22    public E set(int i, E e) throws IndexOutOfBoundsException {
23      checkIndex(i, size);
24      E temp = data[i];
25      data[i] = e;
26      return temp;
27    }
```

Java Implementation, Contd.

```
28    /** Inserts element e to be at index i, shifting all subsequent elements later. */
29    public void add(int i, E e) throws IndexOutOfBoundsException,
30                                                    IllegalStateException {
31      checkIndex(i, size + 1);
32      if (size == data.length)                      // not enough capacity
33        throw new IllegalStateException("Array is full");
34      for (int k=size-1; k >= i; k--)               // start by shifting rightmost
35        data[k+1] = data[k];
36      data[i] = e;                                  // ready to place the new element
37      size++;
38    }
39
40
41
42
43
44
45
46
47
48
49
50
51    protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
52      if (i < 0 || i >= n)
53        throw new IndexOutOfBoundsException("Illegal index: " + i);
54    }
55  }
```

Java Implementation, Contd.

```
28    /** Inserts element e to be at index i, shifting all subsequent elements later. */
29    public void add(int i, E e) throws IndexOutOfBoundsException,
30                                                    IllegalStateException {
31      checkIndex(i, size + 1);
32      if (size == data.length)                      // not enough capacity
33        throw new IllegalStateException("Array is full");
34      for (int k=size-1; k >= i; k--)               // start by shifting rightmost
35        data[k+1] = data[k];
36      data[i] = e;                                  // ready to place the new element
37      size++;
38    }
39    /** Removes/returns the element at index i, shifting subsequent elements earlier. */
40    public E remove(int i) throws IndexOutOfBoundsException {
41      checkIndex(i, size);
42      E temp = data[i];
43      for (int k=i; k < size-1; k++)                // shift elements to fill hole
44        data[k] = data[k+1];
45      data[size-1] = null;                          // help garbage collection
46      size--;
47      return temp;
48    }
49    // utility method
50    /** Checks whether the given index is in the range [0, n-1]. */
51    protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
52      if (i < 0 || i >= n)
53        throw new IndexOutOfBoundsException("Illegal index: " + i);
54    }
55  }
```

Growable Array-based Array List

- In an add(o) operation (without an index), we always add at the end
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

Algorithm *add*(o)
  if $t = S.length - 1$ **then**
    $A \leftarrow$ **new array of size …**
    for $i \leftarrow 0$ **to** $n-1$ **do**
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $n \leftarrow n + 1$
  $S[n-1] \leftarrow o$

Comparison of the Strategies

- We compare them by analyzing the total time $T(n)$ needed to perform a series of n add(o) operations
- We assume that we start with an empty stack represented by an array of size 1
- We call amortized time of an add operation the average time taken by an add over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$1 + c + 2c + 3c + 4c + \ldots + kc =$$

$$1 + c(1 + 2 + 3 + \ldots + k) =$$

$$1 + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(k^2)$, i.e., $O(n^2)$
- The amortized time of an add operation is $O(n)$

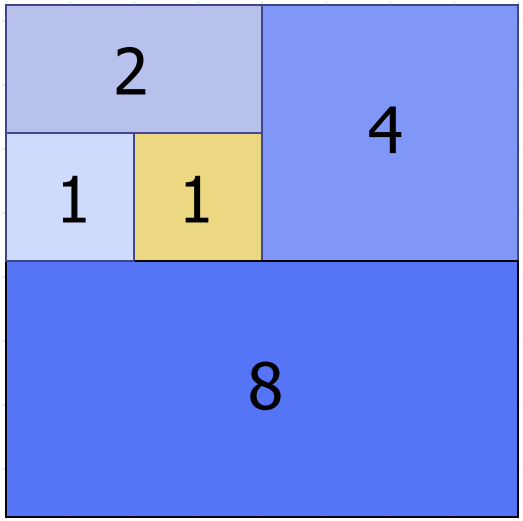
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$1 + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$1 + 2^{k+1} - 1 =$$
$$2n$$

- $T(n)$ is $O(n)$
- The amortized time of an add operation is $O(1)$

geometric series

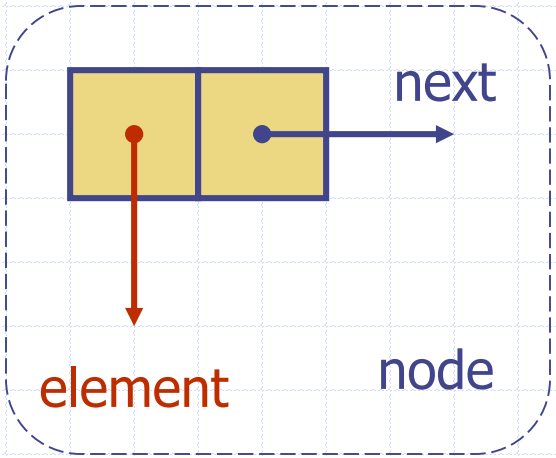


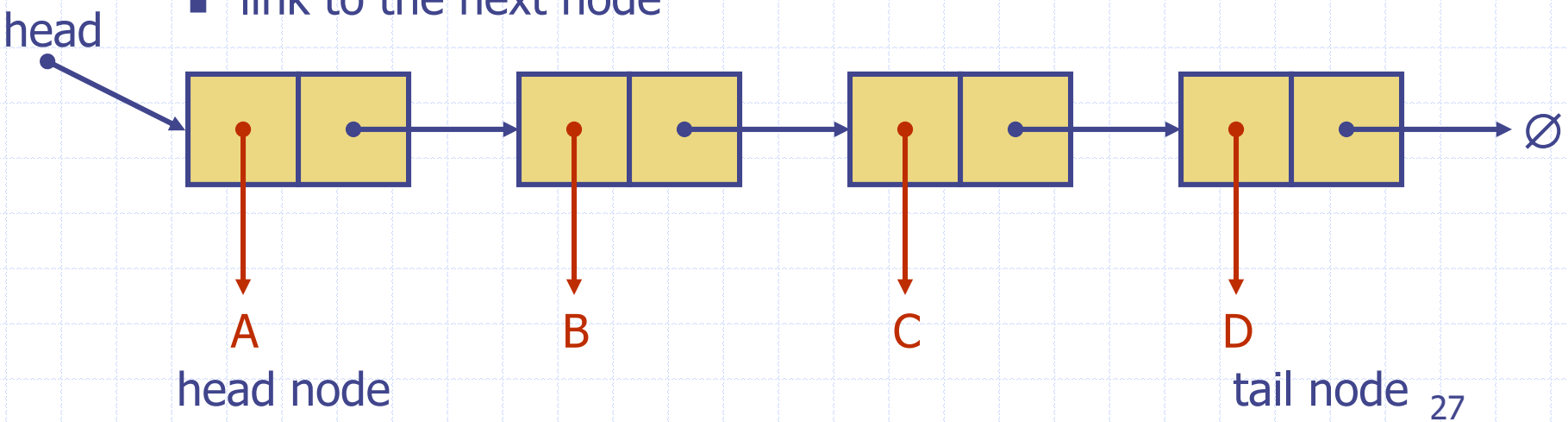
Array based List Implementation

- Space?
  - O(n)
- Getting an element?
  - O(1)
- Setting an element?
  - O(1)
- Adding an element?
  - O(n)
- Removing an element?
  - O(n)
- Growing?
  - O(1)

Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - element
  - link to the next node





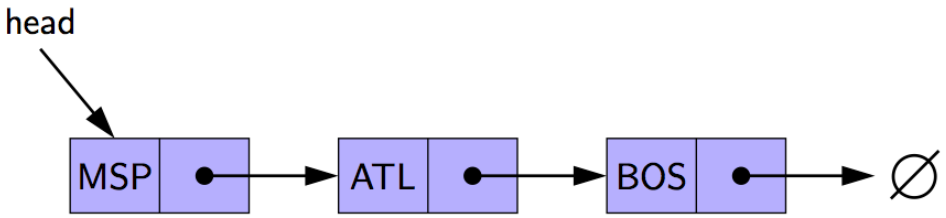
A Nested Node Class

```
1   public class SinglyLinkedList<E> {
2     //---------------- nested Node class ----------------
3     private static class Node<E> {
4       private E element;                  // reference to the element stored at this node
5       private Node<E> next;               // reference to the subsequent node in the list
6       public Node(E e, Node<E> n) {
7         element = e;
8         next = n;
9       }
10      public E getElement( ) { return element; }
11      public Node<E> getNext( ) { return next; }
12      public void setNext(Node<E> n) { next = n; }
13    } //---------- end of nested Node class ----------
      ... rest of SinglyLinkedList class will follow ...
```

Accessor Methods

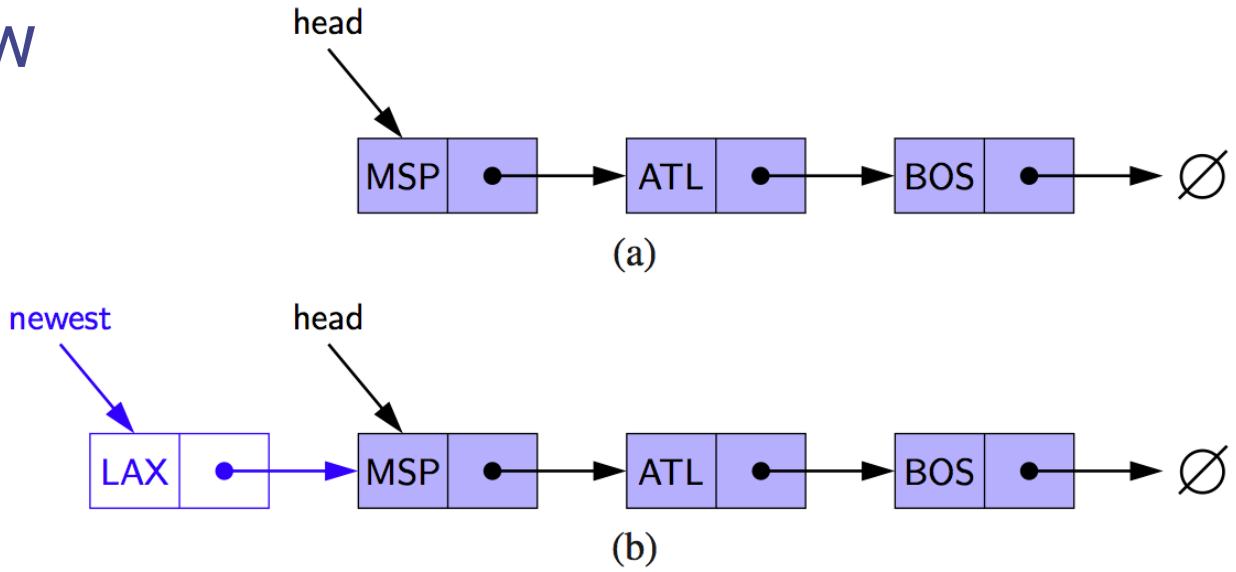
```
1   public class SinglyLinkedList<E> {
...     (nested Node class goes here)
14    // instance variables of the SinglyLinkedList
15    private Node<E> head = null;          // head node of the list (or null if empty)
16    private Node<E> tail = null;          // last node of the list (or null if empty)
17    private int size = 0;                 // number of nodes in the list
18    public SinglyLinkedList( ) { }        // constructs an initially empty list
19    // access methods
20    public int size( ) { return size; }
21    public boolean isEmpty( ) { return size == 0; }
22    public E first( ) {                   // returns (but does not remove) the first element
23      if (isEmpty( )) return null;
24      return head.getElement( );
25    }
26    public E last( ) {                    // returns (but does not remove) the last element
27      if (isEmpty( )) return null;
28      return tail.getElement( );
29    }
```

Inserting at the Head



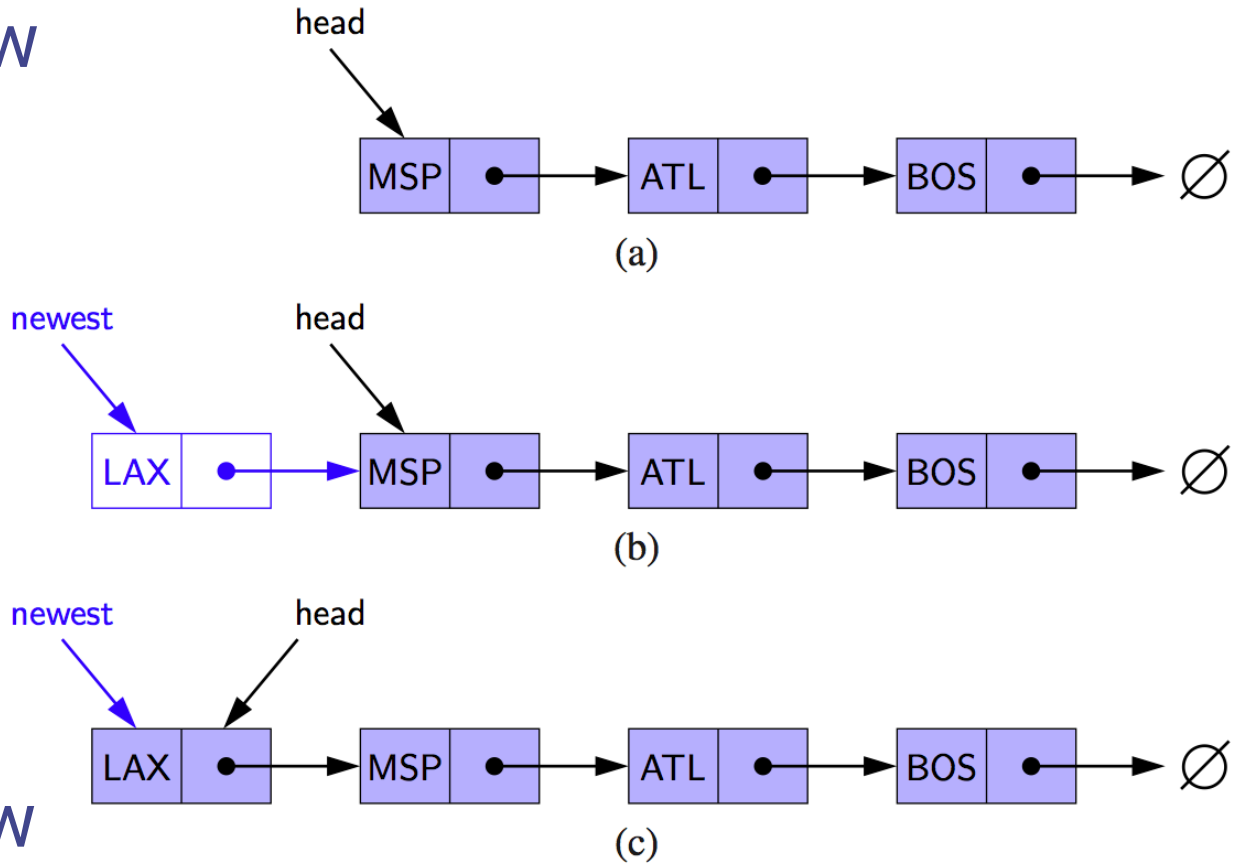
Inserting at the Head

- Allocate new node
- Insert new element
- Have new node point to old head



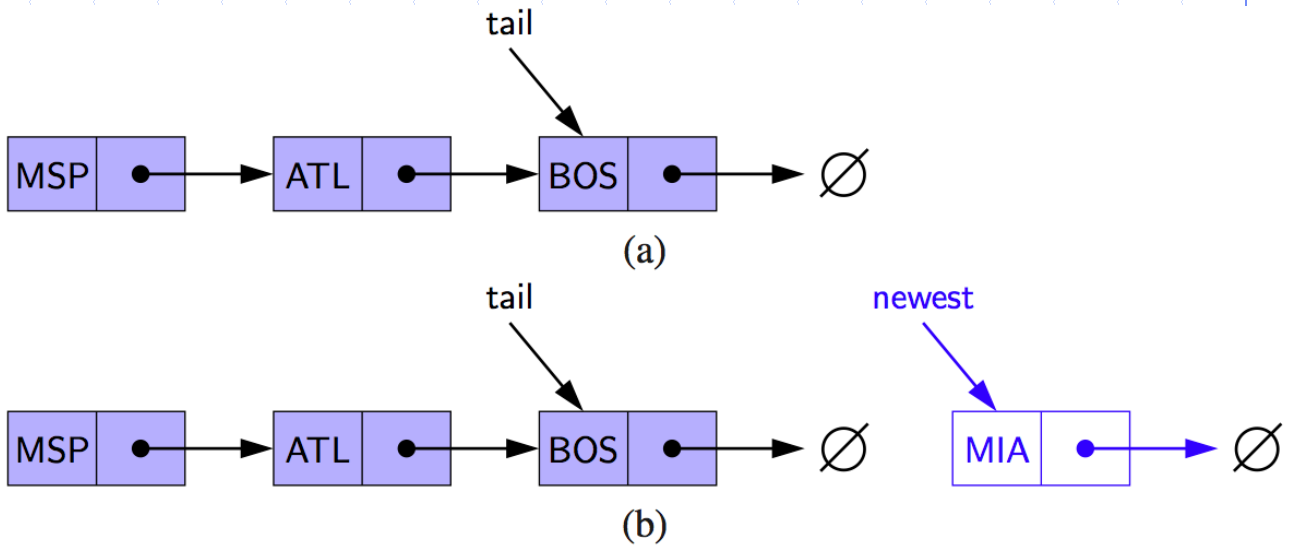
Inserting at the Head

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



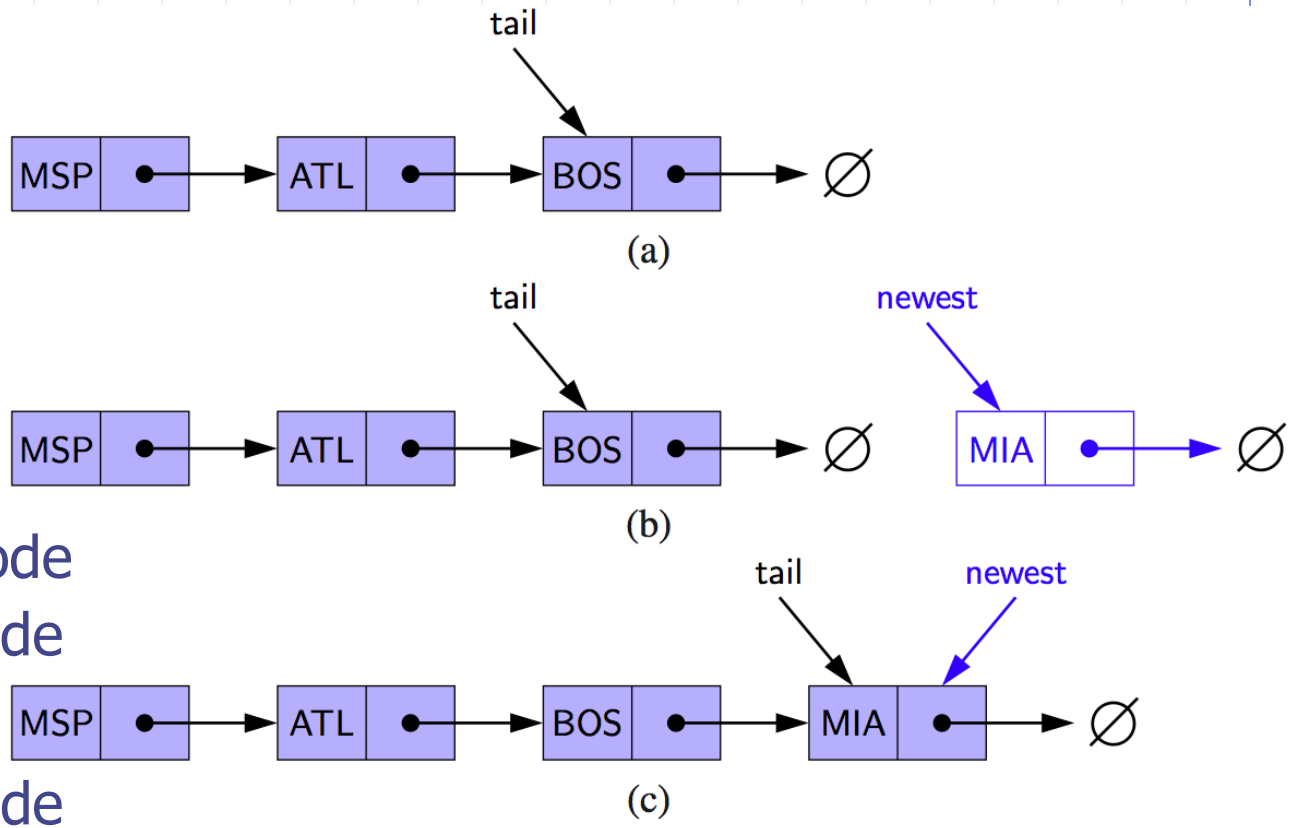
Inserting at the Tail

- Allocate a new node
- Insert new element
- Have new node point to null



Inserting at the Tail

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

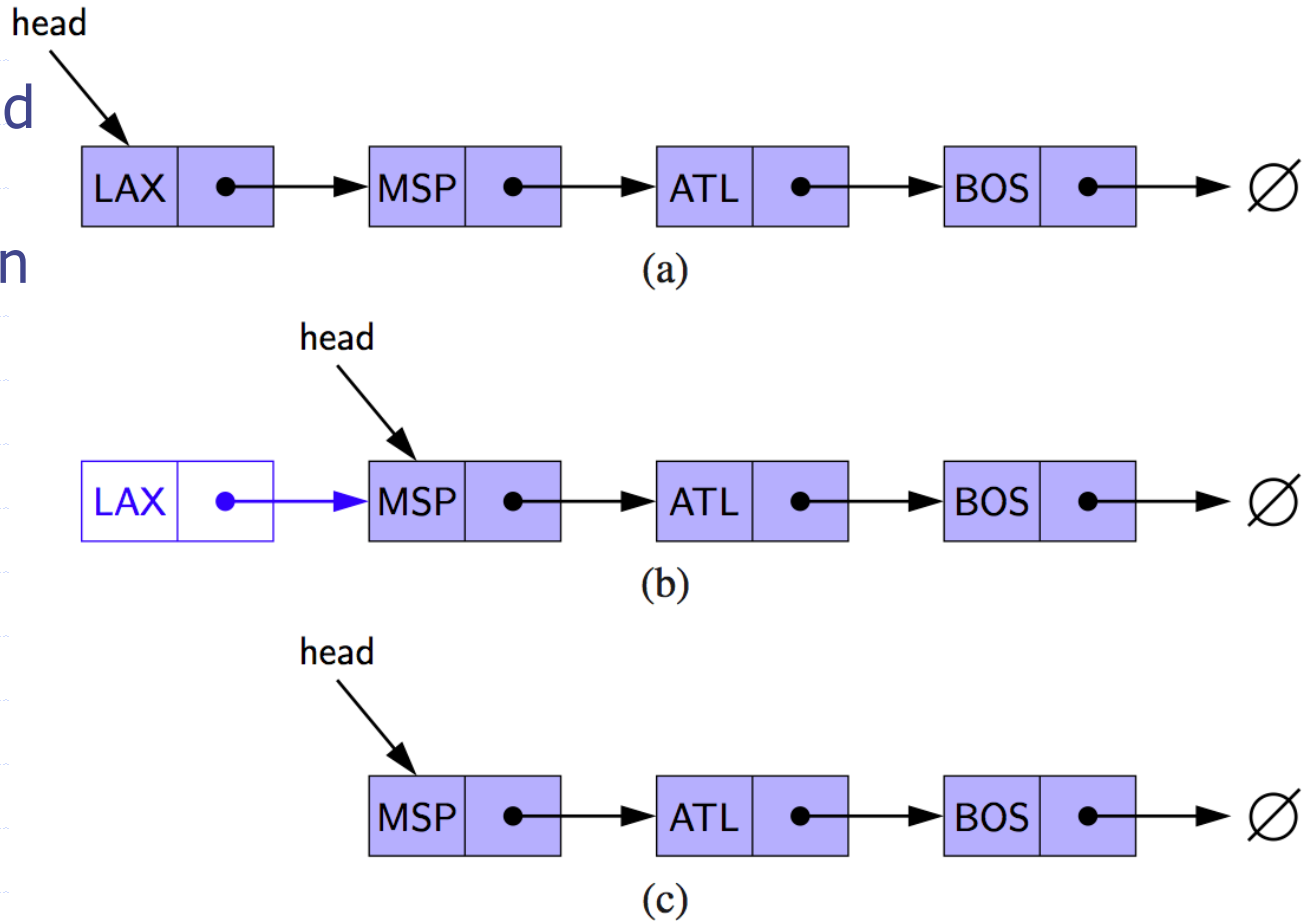


Java Methods

```
31    public void addFirst(E e) {                  // adds element e to the front of the list
32      head = new Node<>(e, head);                // create and link a new node
33      if (size == 0)
34        tail = head;                             // special case: new node becomes tail also
35      size++;
36    }
37    public void addLast(E e) {                   // adds element e to the end of the list
38      Node<E> newest = new Node<>(e, null);      // node will eventually be the tail
39      if (isEmpty( ))
40        head = newest;                           // special case: previously empty list
41      else
42        tail.setNext(newest);                    // new node after existing tail
43      tail = newest;                             // new node becomes the tail
44      size++;
45    }
```

Removing at the Head

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

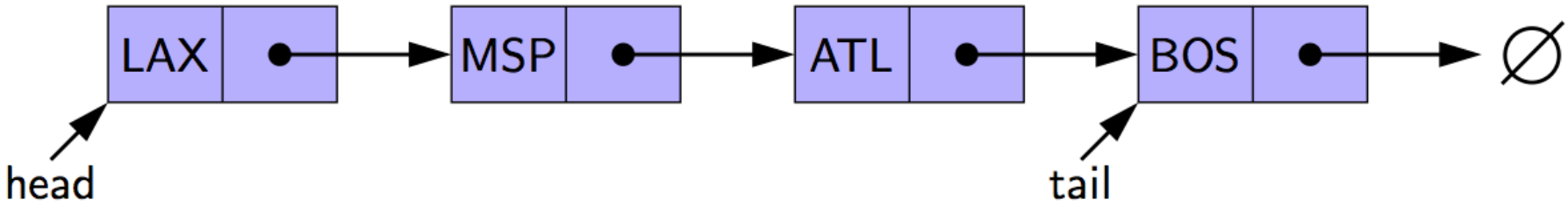


Java Method

```
46    public E removeFirst( ) {                  // removes and returns the first element
47      if (isEmpty( )) return null;             // nothing to remove
48      E answer = head.getElement( );
49      head = head.getNext( );                  // will become null if list had only one node
50      size−−;
51      if (size == 0)
52        tail = null;                           // special case as list is now empty
53      return answer;
54    }
55  }
```

Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node

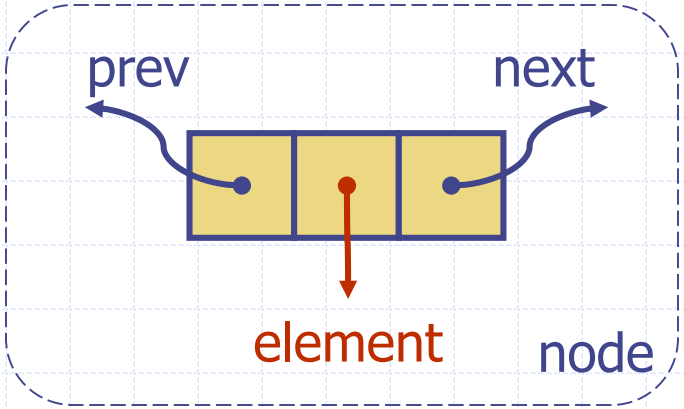


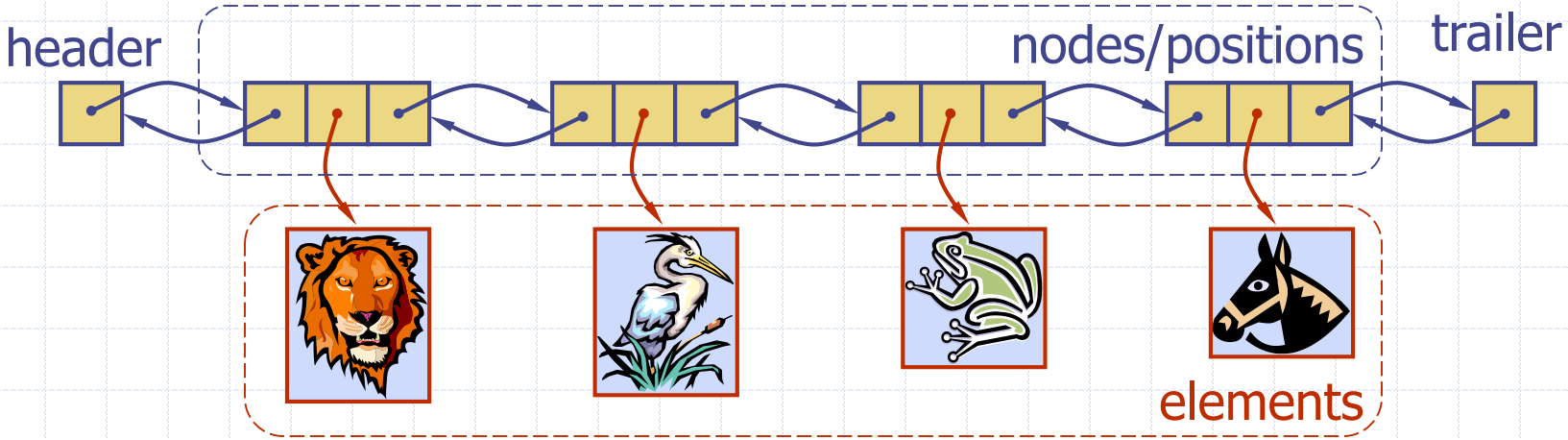
Home Exercise

- Think about how to write the removeLast() as a Java method
- Don’t worry if it is O(n)!

Doubly Linked List

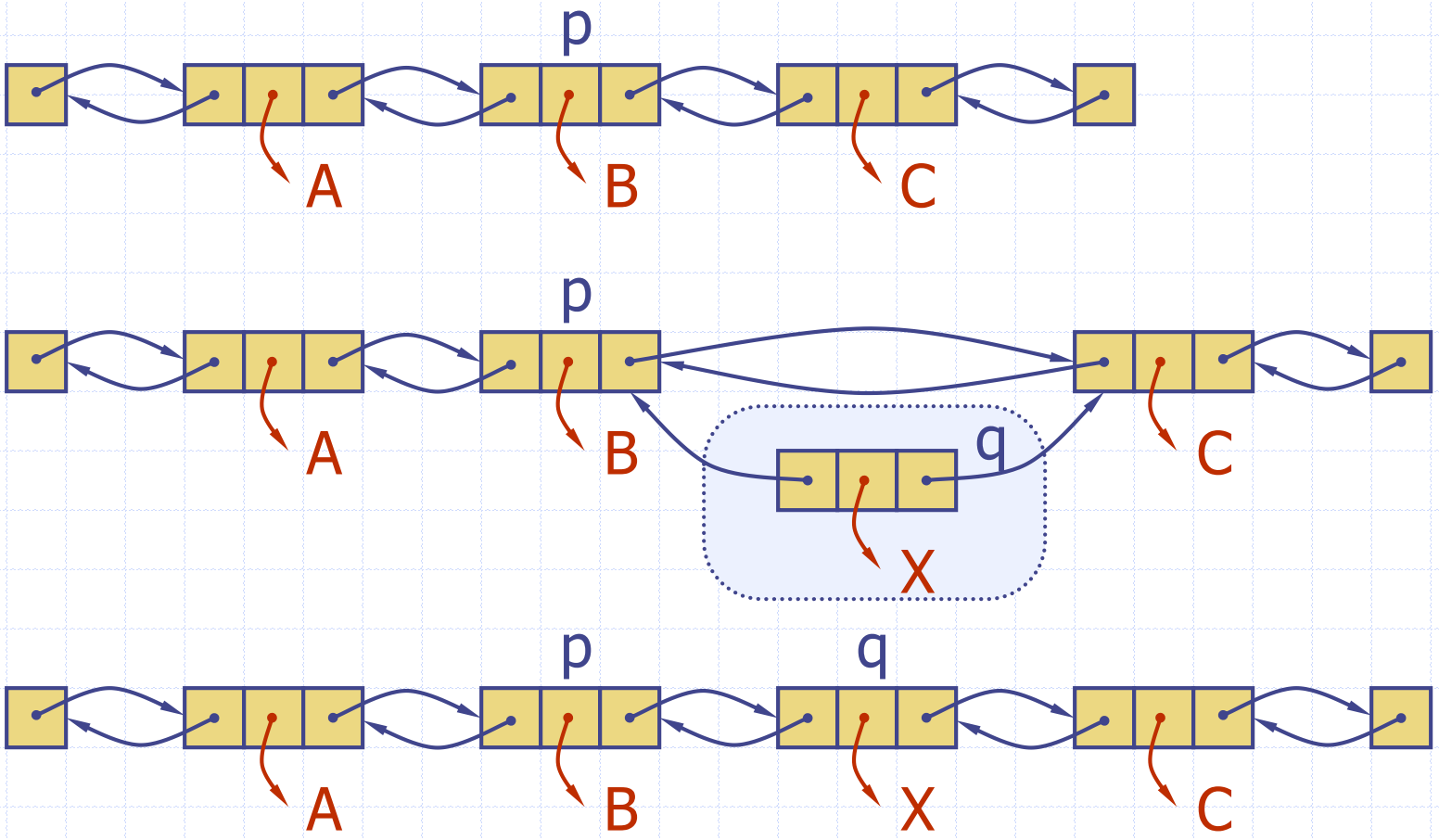
- A doubly linked list can be traversed forward and backward
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





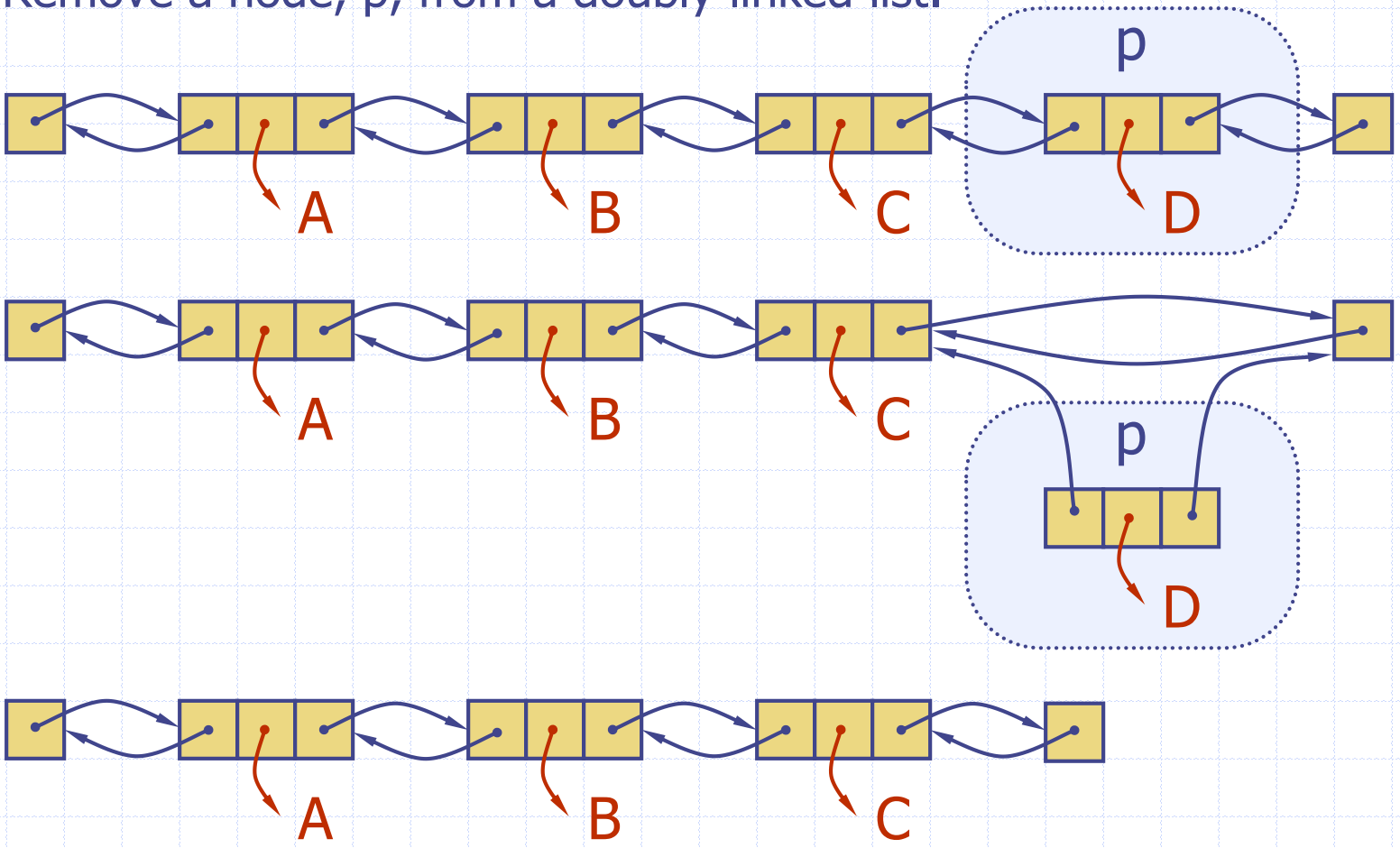
Insertion

- Insert a new node, q, between p and its successor.



Deletion

- Remove a node, p, from a doubly linked list.



Doubly-Linked List in Java

```
1  /** A basic doubly linked list implementation. */
2  public class DoublyLinkedList<E> {
3    //---------------- nested Node class ----------------
4    private static class Node<E> {
5      private E element;                  // reference to the element stored at this node
6      private Node<E> prev;               // reference to the previous node in the list
7      private Node<E> next;               // reference to the subsequent node in the list
8      public Node(E e, Node<E> p, Node<E> n) {
9        element = e;
10       prev = p;
11       next = n;
12     }
13     public E getElement( ) { return element; }
14     public Node<E> getPrev( ) { return prev; }
15     public Node<E> getNext( ) { return next; }
16     public void setPrev(Node<E> p) { prev = p; }
17     public void setNext(Node<E> n) { next = n; }
18   } //---------- end of nested Node class ----------
19
```

Doubly-Linked List in Java, 2

```
21    private Node<E> header;                          // header sentinel
22    private Node<E> trailer;                         // trailer sentinel
23    private int size = 0;                            // number of elements in the list
24    /** Constructs a new empty list. */
25    public DoublyLinkedList( ) {
26      header = new Node<>(null, null, null);         // create header
27      trailer = new Node<>(null, header, null);      // trailer is preceded by header
28      header.setNext(trailer);                       // header is followed by trailer
29    }
30    /** Returns the number of elements in the linked list. */
31    public int size( ) { return size; }
32    /** Tests whether the linked list is empty. */
33    public boolean isEmpty( ) { return size == 0; }
34    /** Returns (but does not remove) the first element of the list. */
35    public E first( ) {
36      if (isEmpty( )) return null;
37      return header.getNext( ).getElement( );        // first element is beyond header
38    }
39    /** Returns (but does not remove) the last element of the list. */
40    public E last( ) {
41      if (isEmpty( )) return null;
42      return trailer.getPrev( ).getElement( );       // last element is before trailer
43    }
```

Doubly-Linked List in Java, 3

```
44    // public update methods
45    /** Adds element e to the front of the list. */
46    public void addFirst(E e) {
47      addBetween(e, header, header.getNext( ));          // place just after the header
48    }
49    /** Adds element e to the end of the list. */
50    public void addLast(E e) {
51      addBetween(e, trailer.getPrev( ), trailer);        // place just before the trailer
52    }
53    /** Removes and returns the first element of the list. */
54    public E removeFirst( ) {
55      if (isEmpty( )) return null;                       // nothing to remove
56      return remove(header.getNext( ));                  // first element is beyond header
57    }
58    /** Removes and returns the last element of the list. */
59    public E removeLast( ) {
60      if (isEmpty( )) return null;                       // nothing to remove
61      return remove(trailer.getPrev( ));                 // last element is before trailer
62    }
```

Doubly-Linked List in Java, 4

```
64    // private update methods
65    /** Adds element e to the linked list in between the given nodes. */
66    private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
67      // create and link a new node
68      Node<E> newest = new Node<>(e, predecessor, successor);
69      predecessor.setNext(newest);
70      successor.setPrev(newest);
71      size++;
72    }
73    /** Removes the given node from the list and returns its element. */
74    private E remove(Node<E> node) {
75      Node<E> predecessor = node.getPrev( );
76      Node<E> successor = node.getNext( );
77      predecessor.setNext(successor);
78      successor.setPrev(predecessor);
79      size--;
80      return node.getElement( );
81    }
82  } //---------- end of DoublyLinkedList class ----------
```

List Implementations

- Array Lists
- **Linked List**

Linked List Based List Implementation

- We used explicit nodes to add/remove elements in the doubly linked list implementations
- What happens when we only have indices?

- Set/Get element at an index i:
  - Traverse the list starting from the head (or tail) i times (n-i)
- Add/Remove Element
  - Traverse to the correct location then add/remove
- Performance?
  - O(n)

- Home Exercise: Think about how to implement the List ADT without positions

Positional Lists

- Want to perform arbitrary insertion/deletions efficiently and capture the "locality" of the elements
- **Positional list** ADT: A general abstraction of a sequence of elements with the ability to identify the location of an element
- A **position** acts as a marker or token within the broader positional list.
- A position p is unaffected by changes elsewhere in a list
- A position instance is a simple object, supporting only the following method:
  - p.getElement( ): Return the element stored at position p.

Positional List ADT

- Accessor methods:

first( ): Returns the position of the first element of L (or null if empty).

last( ): Returns the position of the last element of L (or null if empty).

before(p): Returns the position of L immediately before position p (or null if p is the first position).

after(p): Returns the position of L immediately after position p (or null if p is the last position).

isEmpty( ): Returns true if list L does not contain any elements.

size( ): Returns the number of elements in list L.

Positional List ADT, 2

- Update methods:

addFirst(e): Inserts a new element e at the front of the list, returning the position of the new element.

addLast(e): Inserts a new element e at the back of the list, returning the position of the new element.

addBefore(p, e): Inserts a new element e in the list, just before position p, returning the position of the new element.

addAfter(p, e): Inserts a new element e in the list, just after position p, returning the position of the new element.

set(p, e): Replaces the element at position p with element e, returning the element formerly at position p.

remove(p): Removes and returns the element at position p in the list, invalidating the position.

Example

- A sequence of Positional List operations:

Method	Return Value	List Contents
addLast(8)	p	(8_p)
first( )		
addAfter(p, 5)		
before(q)		
addBefore(q, 3)		
r.getElement( )		
after(p)		
before(p)		
addFirst(9)		
remove(last( ))		
set(p, 7)		
remove(q)		

Example

- A sequence of Positional List operations:

Method	Return Value	List Contents
addLast(8)	p	(8_p)
first( )	p	(8_p)
addAfter(p, 5)	q	$(8_p, 5_q)$
before(q)	p	$(8_p, 5_q)$
addBefore(q, 3)	r	$(8_p, 3_r, 5_q)$
r.getElement( )		
after(p)		
before(p)		
addFirst(9)		
remove(last( ))		
set(p, 7)		
remove(q)		

Example

- A sequence of Positional List operations:

Method	Return Value	List Contents
addLast(8)	p	(8_p)
first( )	p	(8_p)
addAfter(p, 5)	q	$(8_p, 5_q)$
before(q)	p	$(8_p, 5_q)$
addBefore(q, 3)	r	$(8_p, 3_r, 5_q)$
r.getElement( )	3	$(8_p, 3_r, 5_q)$
after(p)	r	$(8_p, 3_r, 5_q)$
before(p)	null	$(8_p, 3_r, 5_q)$
addFirst(9)	s	$(9_s, 8_p, 3_r, 5_q)$
remove(last( ))	5	$(9_s, 8_p, 3_r)$
set(p, 7)	8	$(9_s, 7_p, 3_r)$
remove(q)	"error"	$(9_s, 7_p, 3_r)$

Positional Lists

- **Positional list** ADT: A general abstraction of a sequence of elements with the ability to identify the location of an element

- Most natural way to implement a positional list is to use a **doubly linked list**

  (Arrays and other data structures we are going to see can be used to represent these as well)

Linked List based Positional List

- Space? Entire list/Single Position
  - O(n) / O(1)
- All position based operations (including adding and removing):
  - O(1)

- If we do not use positions?
  - Inserting and removing: O(n)

Arrays vs Linked Lists

Operation	Array	List
size, isEmpty	1	1
atIndex, indexOf, get	1	n
first, last, prev, next	1	1
set(p,e)	1	1
set(i,e)	1	n
add(i,e), remove(i)	n	n
addFirst, addLast	n,1	1
addAfter, addBefore	n	1
remove(p)	n	1

Array implementations tend to have lower constants!

Arrays vs Linked Lists

- Most obvious advantages:
  - Arrays: Fast Random Access
  - Linked Lists: Fast Add/Remove From Tail or Head(we will see why we need this soon), or when you know the position

There are other advantages/disadvantages that you do not need to worry about right now

Notes

- Java has ADT List (as an interface)
  - It also has AbstractList abstract class!
- Java has a concrete ArrayList class and a LinkedList (doubly linked) class
  - They implement List and extend AbstractList
- It is important for you to understand how they are implemented!
  - Use ArrayList in your own programs (we might force you to implement your own)

Home Exercise

- Study all the java implementations you have seen
- Think about linked lists without explicit positions

- For the ambitious student: Implement all the data structures you have seen so far from scratch by yourself!

Iterators

- An iterator is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a time.

hasNext( ):	Returns true if there is at least one additional element in the sequence, and false otherwise.
next( ):	Returns the next element in the sequence.

- Extends the concepts of position by adding traversal capability

The Iterable Interface

- Java defines a parameterized interface, named Iterable, that includes the following single method:
  - iterator( ): Returns an iterator of the elements in the collection.
- An instance of a typical collection class in Java, such as an ArrayList, is iterable (but not itself an iterator); it produces an iterator for its collection as the return value of the iterator( ) method.
- Each call to iterator( ) returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection.

Iterators

- Two notions of iterator:
  - snapshot: freezes the contents of the data structure at a given time
  - dynamic: follows changes to the data structure
  - In Java: an iterator will fail (and throw an exception) if the underlying collection changes unexpectedly (thus snapshot type)

The for-each Loop

- Java's Iterable class also plays a fundamental role in support of the "for-each" loop syntax:

```
for (ElementType variable : collection) {
  loopBody                                   // may refer to "variable"
}
```

is equivalent to:

```
Iterator<ElementType> iter = collection.iterator();
while (iter.hasNext()) {
  ElementType variable = iter.next();
  loopBody                                   // may refer to "variable"
}
```