Fundamental Data Structures

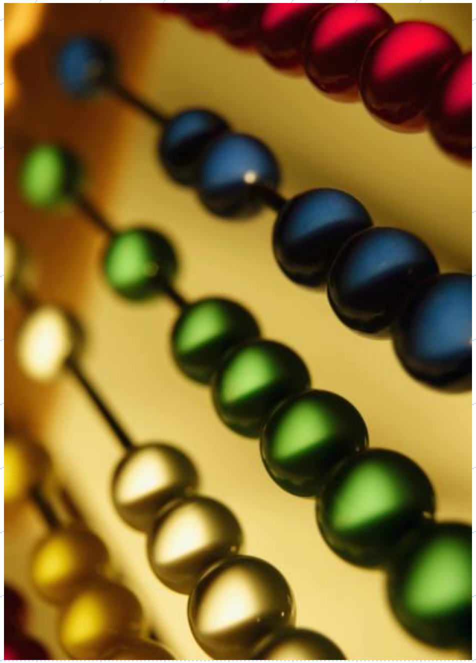
- **Arrays**
  - Add, Remove operations

- **Singly Linked Lists**
  - Inserting at the head and tail
  - Removing at the head and tail

- **Doubly Linked Lists**
  - Insertion
  - Deletion

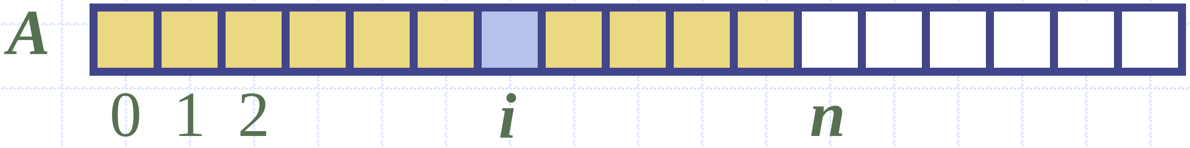
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Arrays



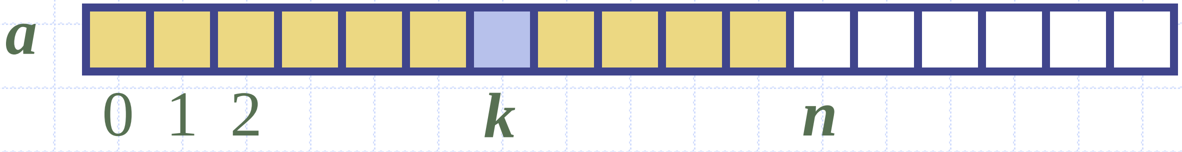
Array Definition

- 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.



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* 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)

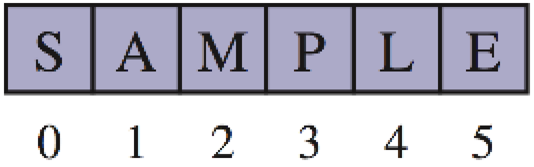
- The second way to create an array is to use the **new** operator.
  - However, because **an array is not an instance of a class**, we do not use a typical constructor. Instead we use the syntax:

new *elementType*[*length*]

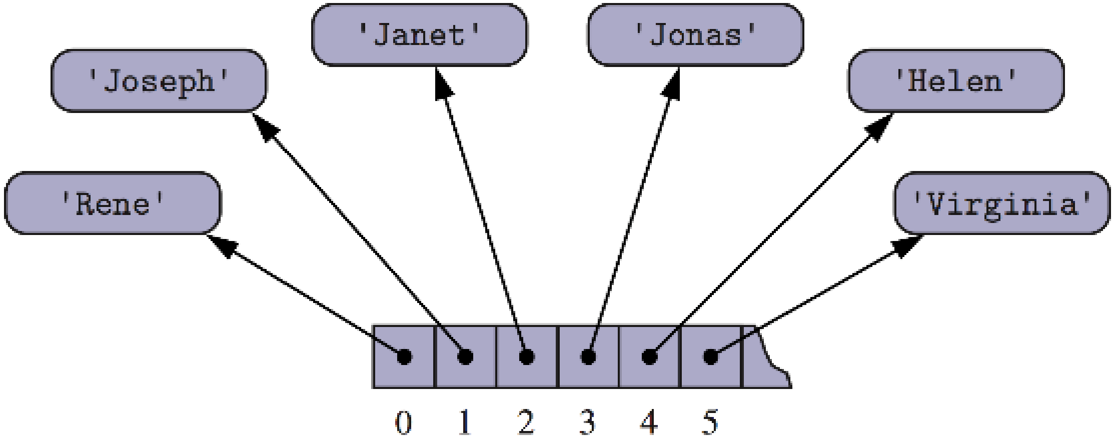
- *length* is a positive integer denoting the length of the new array.
- The **new** operator returns a reference to the new array, and typically this would be assigned to an array variable.

Arrays of Characters or Object References

- An array can store primitive elements, such as characters.



- An array can also store references to objects.



Java Example: Game Entries

- A game entry stores the **name** of a player and her **best score** so far in a game

```
public class GameEntry {
  private String name;                           // name of the person earning this score
  private int score;                             // the score value
  /** Constructs a game entry with given parameters.. */
  public GameEntry(String n, int s) {
    name = n;
    score = s;
  }
  /** Returns the name field. */
  public String getName( ) { return name; }
  /** Returns the score field. */
  public int getScore( ) { return score; }
  /** Returns a string representation of this entry. */
  public String toString( ) {
    return "(" + name + ", " + score + ")";
  }
}
```

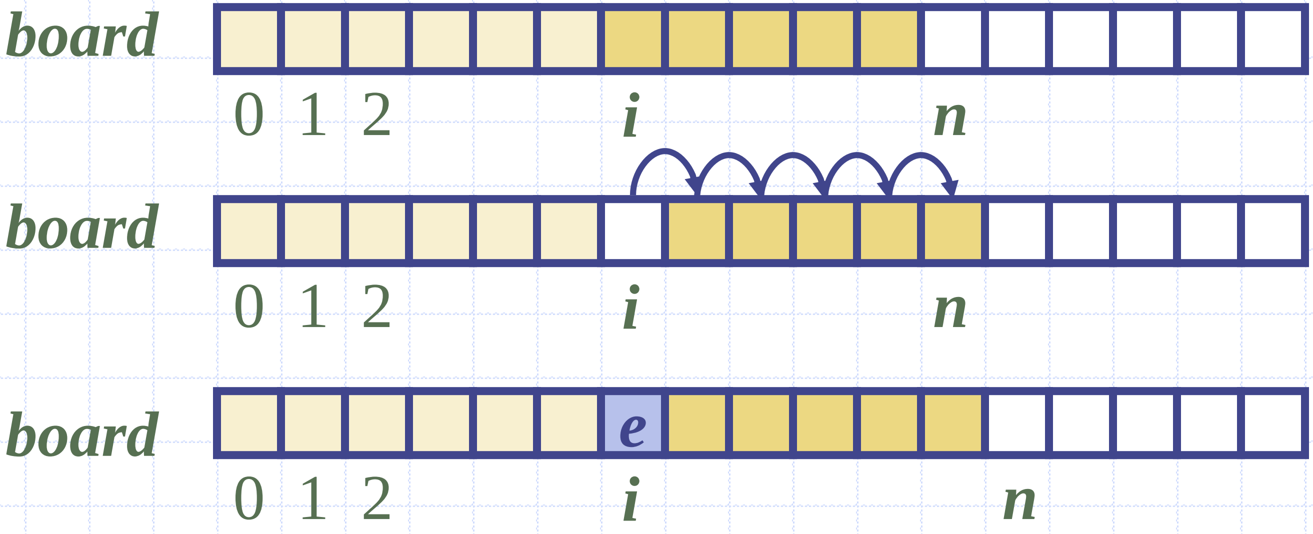
Java Example: Scoreboard

- Keep track of players and their best scores in an array, board
  - The elements of board are objects of class GameEntry
  - Array board is **sorted by score**

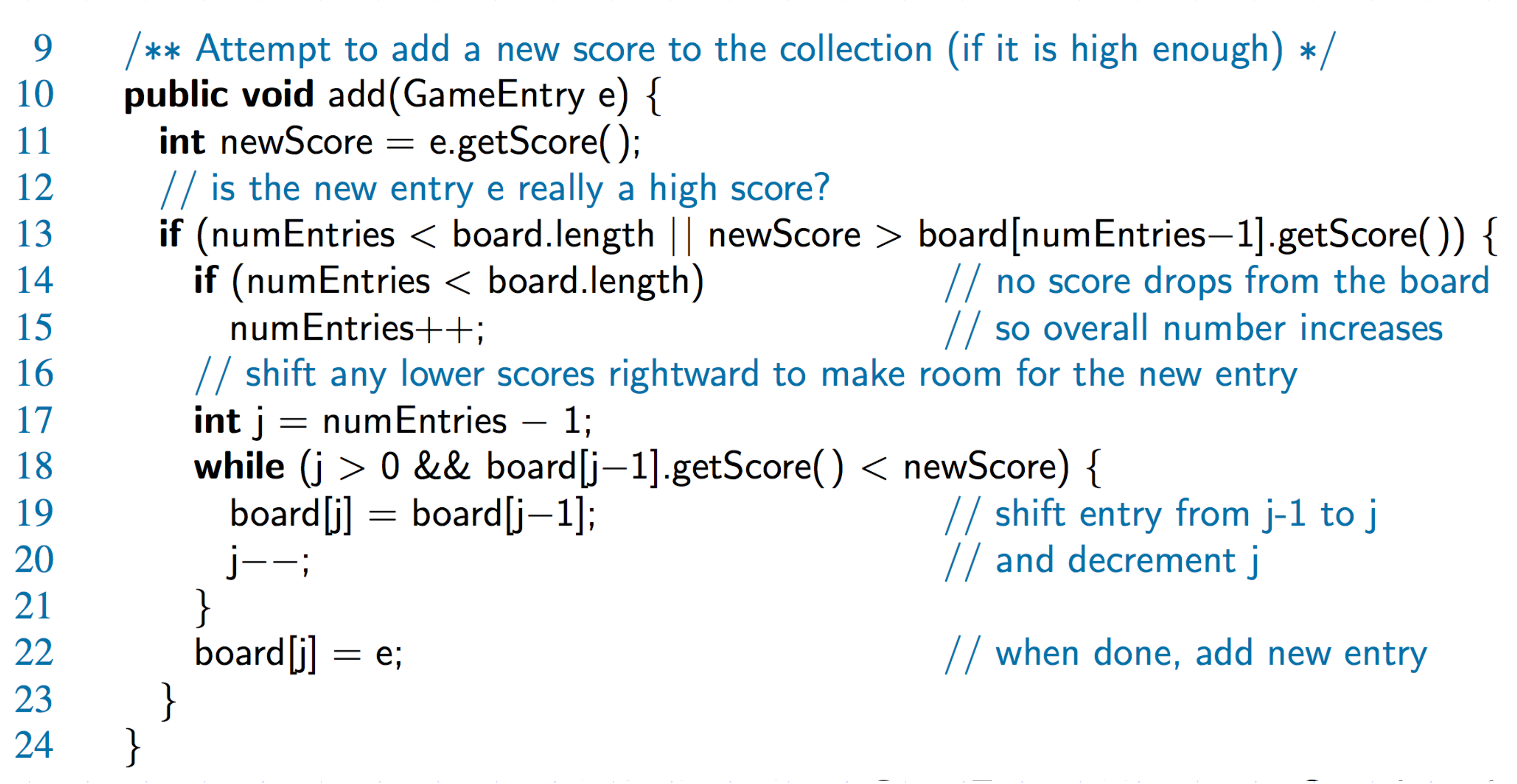
```
/** Class for storing high scores in an array in nondecreasing order. */
public class Scoreboard {
  private int numEntries = 0;                    // number of actual entries
  private GameEntry[ ] board;                    // array of game entries (names & scores)
  /** Constructs an empty scoreboard with the given capacity for storing entries. */
  public Scoreboard(int capacity) {
    board = new GameEntry[capacity];
  }
...
  // more methods will go here
}
```

Adding an Entry

- To add an entry e into array board at index i, we need to make room for it by shifting forward the $\boldsymbol{n} - \boldsymbol{i}$ entries $\boldsymbol{board}[\boldsymbol{i}], \ldots, \boldsymbol{board}[\boldsymbol{n} - 1]$



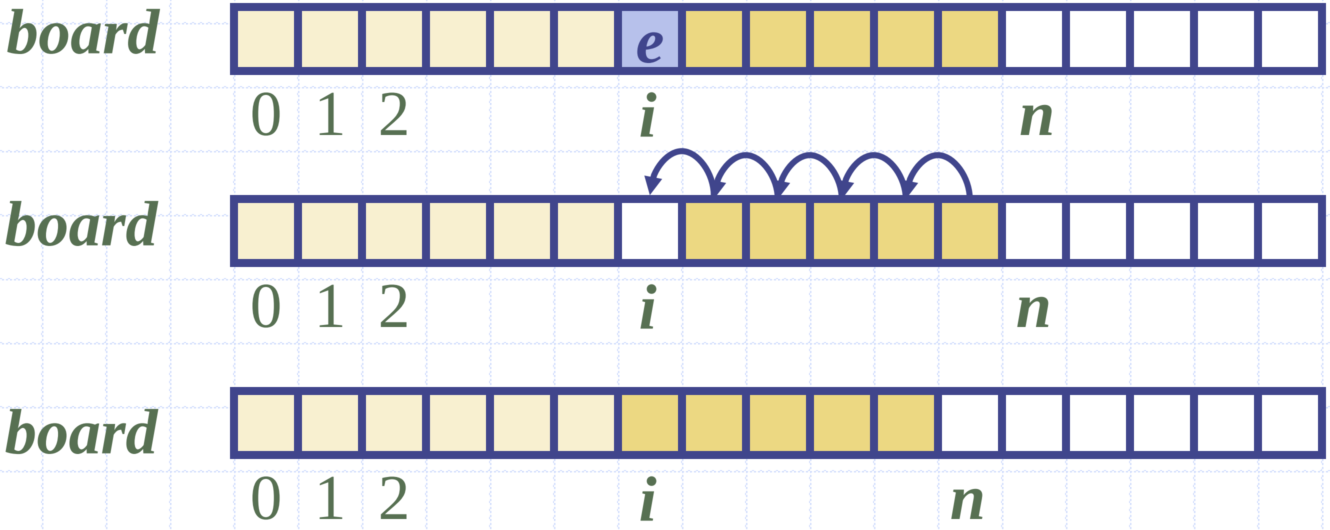
Java Example



```
/** Attempt to add a new score to the collection (if it is high enough) */
public void add(GameEntry e) {
  int newScore = e.getScore( );
  // is the new entry e really a high score?
  if (numEntries < board.length || newScore > board[numEntries-1].getScore( )) {
    if (numEntries < board.length)                    // no score drops from the board
      numEntries++;                                   // so overall number increases
    // shift any lower scores rightward to make room for the new entry
    int j = numEntries - 1;
    while (j > 0 && board[j-1].getScore( ) < newScore) {
      board[j] = board[j-1];                          // shift entry from j-1 to j
      j--;                                            // and decrement j
    }
    board[j] = e;                                     // when done, add new entry
  }
}
```

Removing an Entry

- To remove the entry e at index i, we need to fill the hole left by **e** by shifting backward the $n - i - 1$ elements ***board***[***i*** + 1], …, ***board***[***n*** □ 1]

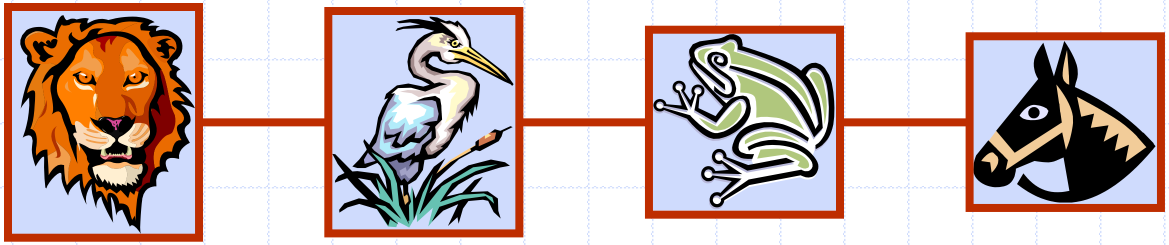


Java Example

```
/** Remove and return the high score at index i. */
public GameEntry remove(int i) throws IndexOutOfBoundsException {
  if (i < 0 || i >= numEntries)
    throw new IndexOutOfBoundsException("Invalid index: " + i);
  GameEntry temp = board[i];                    // save the object to be removed
  for (int j = i; j < numEntries - 1; j++)      // count up from i (not down)
    board[j] = board[j+1];                      // move one cell to the left
  board[numEntries -1 ] = null;                 // null out the old last score
  numEntries--;
  return temp;                                  // return the removed object
}
```

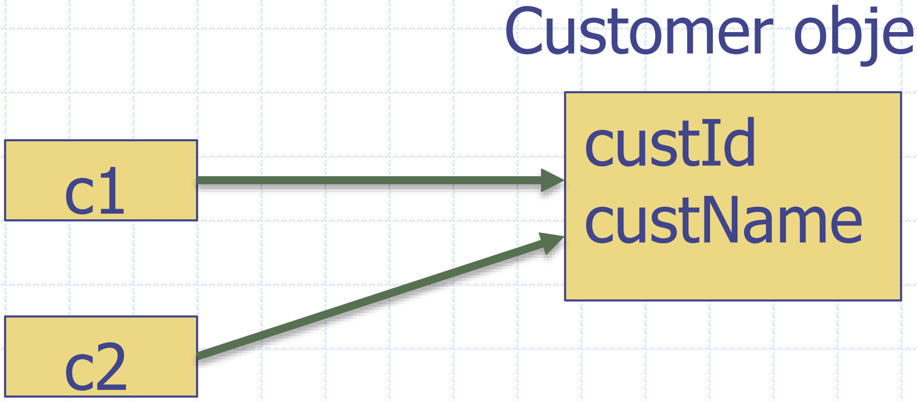
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Singly Linked Lists



Reference Variables - review

- Customer c1 = new Customer();
- c1 is a **reference variable** that contains the address of the Customer object.
- Customer c2 = c1; //set c2 to reference c1
- c2 is another reference variable that **refers to the same** Customer **object**.



Self-Referential Classes

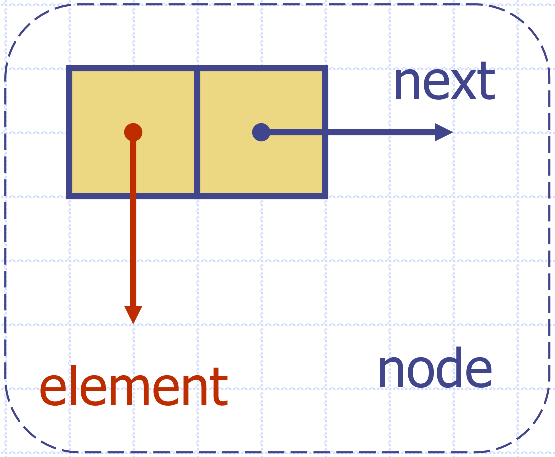
- Contains an instance variable that refers to another object of the same class type:

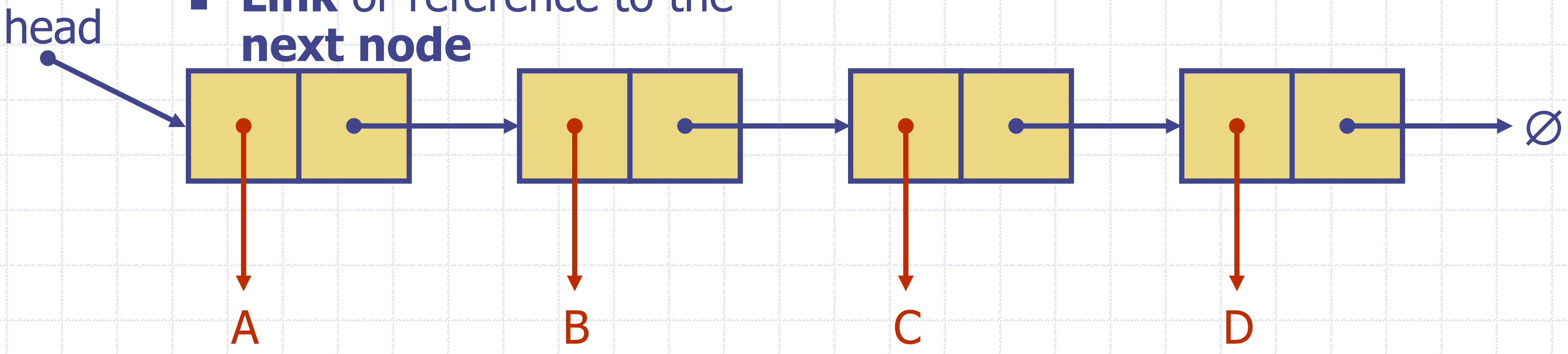
```
public class Item<E> {
   private E element;
   private Item<E> nextItem;
}
```

- The instance variable nextItem refers to an **Item** object.
  - is called a **link** because usually is used to link or reference the **next item in a sequence**.

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** or reference to the **next node**





Singly Linked List

- The **tail** of a list can be found by ***traversing*** the linked list starting at the ***head*** and moving from one node to another by following each node's **next** reference.
- Storing an explicit reference to the ***tail*** node is a common efficiency to avoid such a traversal:

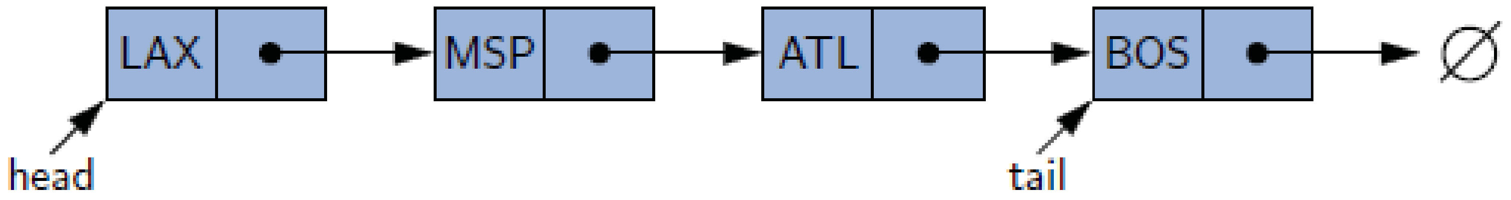
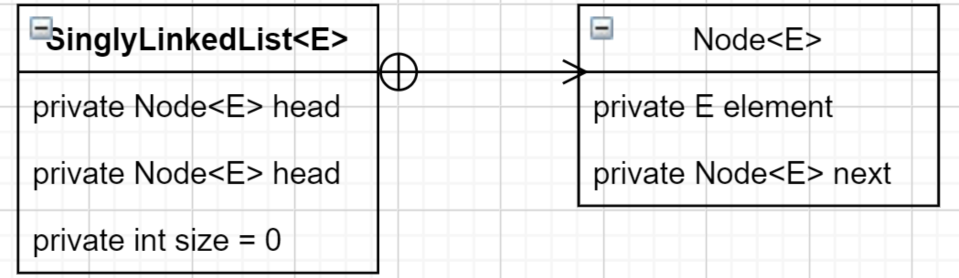


Figure 3.11: Example of a singly linked list whose elements are strings indicating airport codes. The list instance maintains a member named head that refers to the first node of the list, and another member named tail that refers to the last node of the list. The **null** value is denoted as Ø.

Singly Linked List - design

- A class to **represent a node** in the list – needs two instance variables to store:
  - the **element**
  - the **link** or reference to the **next node**
  - a **constructor** with two arguments to initialize the node.
  - **getter** and **setter** methods
- Another class to **represent the list of nodes –** needs three instance variables to store:
  - the **head**
  - **tail**
  - the list **size**
  - **accessor methods** and **list operations** (add, remove, etc.)



SinglyLinkedList<E>

private Node<E> head

private Node<E> head

private int size = 0

Node<E>

private E element

private Node<E> next

A Nested Node Class

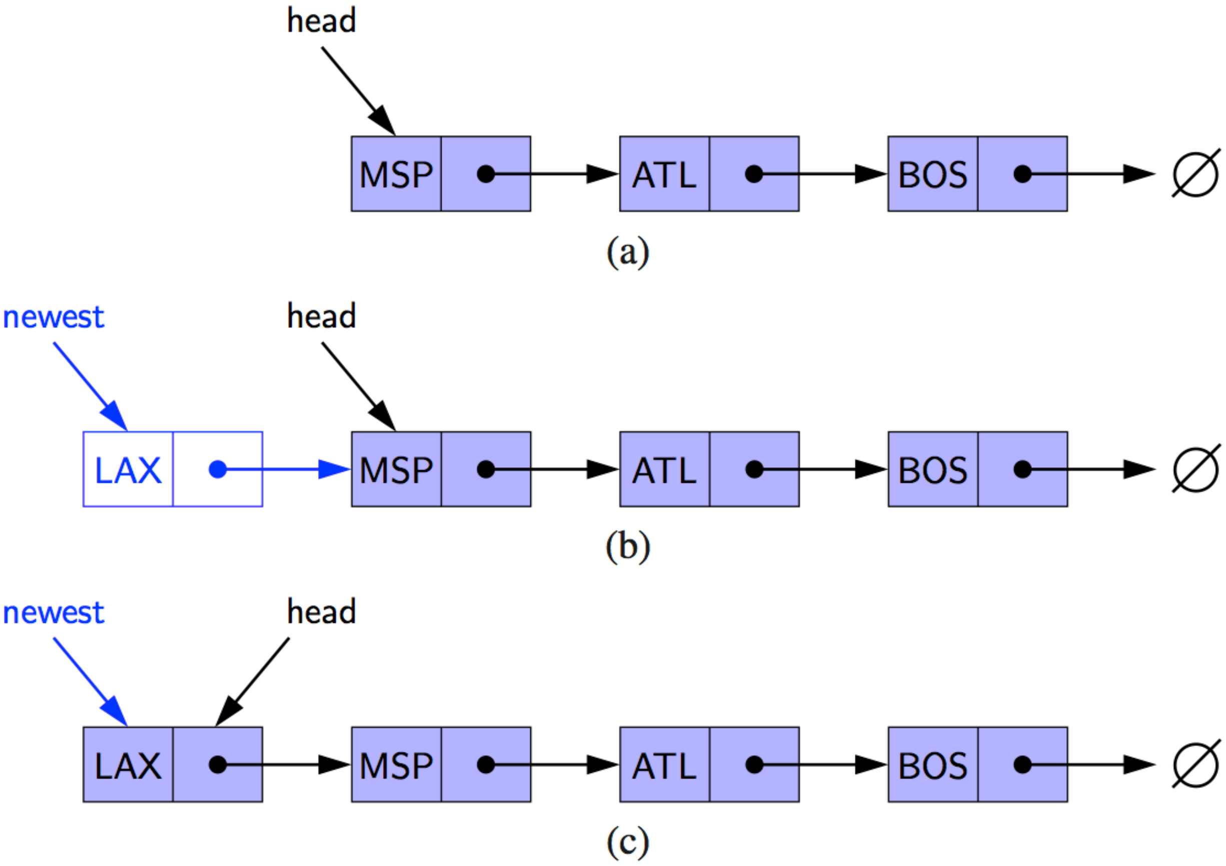
```
public class SinglyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> {
    private E element;                 // reference to the element stored at this node
    private Node<E> next;              // reference to the subsequent node in the list
    public Node(E e, Node<E> n) {
      element = e;
      next = n;
    }
    public E getElement( ) { return element; }
    public Node<E> getNext( ) { return next; }
    public void setNext(Node<E> n) { next = n; }
  } //----------- end of nested Node class -----------
  ... rest of SinglyLinkedList class will follow ...
```

Accessor Methods

```
public class SinglyLinkedList<E> {
  (nested Node class goes here)
  // instance variables of the SinglyLinkedList
  private Node<E> head = null;          // head node of the list (or null if empty)
  private Node<E> tail = null;          // last node of the list (or null if empty)
  private int size = 0;                 // number of nodes in the list
  public SinglyLinkedList( ) { }        // constructs an initially empty list
  // access methods
  public int size( ) { return size; }
  public boolean isEmpty( ) { return size == 0; }
  public E first( ) {                 // returns (but does not remove) the first element
    if (isEmpty( )) return null;
    return head.getElement( );
  }
  public E last( ) {                  // returns (but does not remove) the last element
    if (isEmpty( )) return null;
    return tail.getElement( );
  }
```

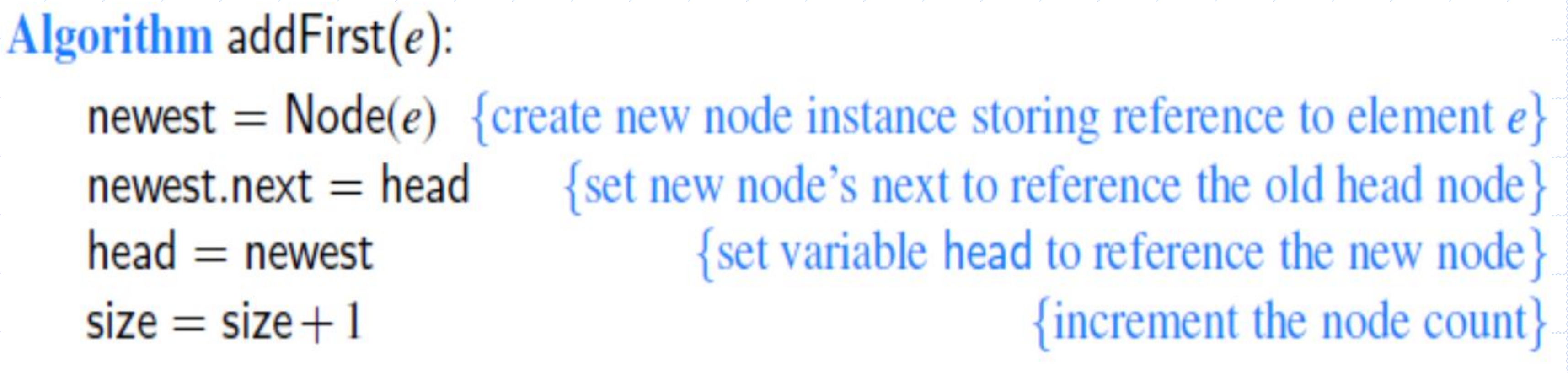
Inserting at the Head

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



Inserting at the Head

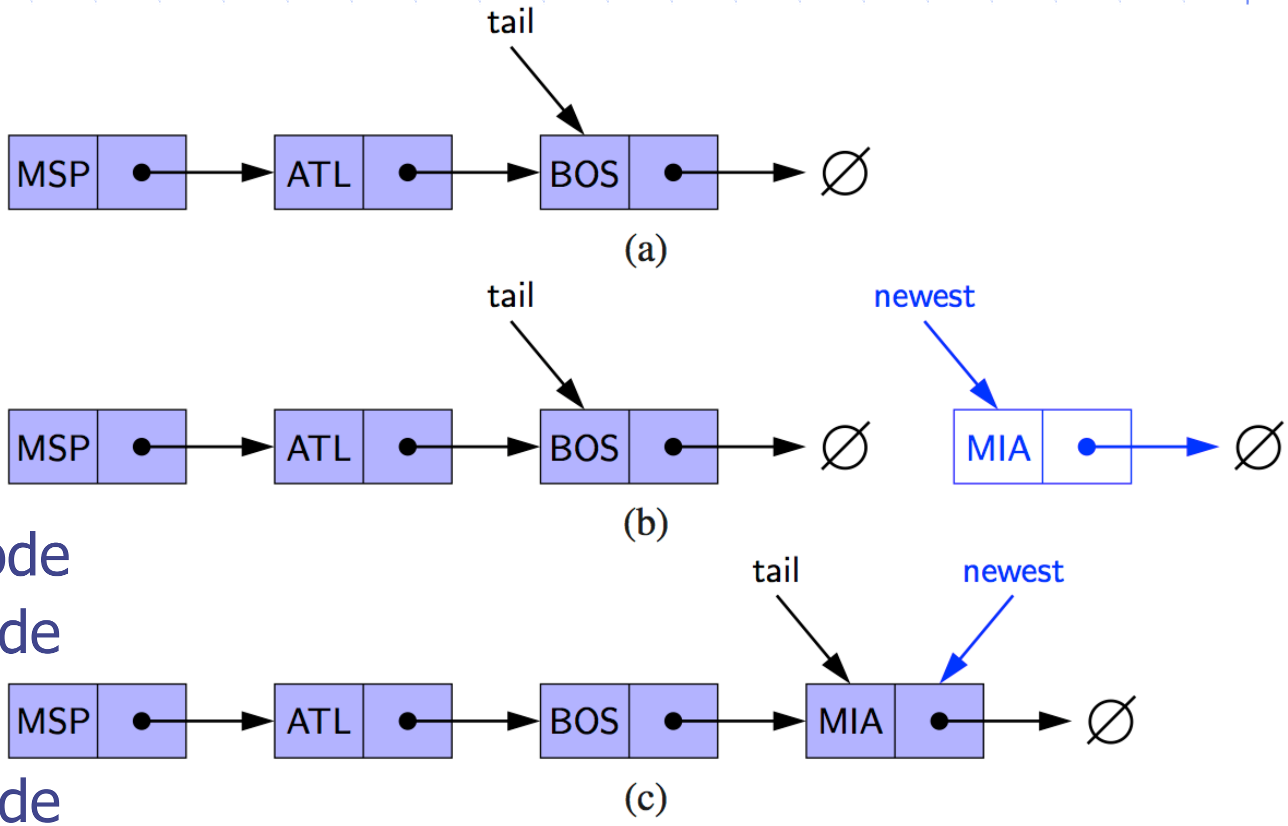
- Allocate new node
- Insert new element
- Have new node point to old head
- Update head to point to new node



```
Algorithm addFirst($e$):
    newest = Node($e$)   {create new node instance storing reference to element $e$}
    newest.next = head   {set new node's next to reference the old head node}
    head = newest        {set variable head to reference the new node}
    size = size + 1      {increment the node count}
```

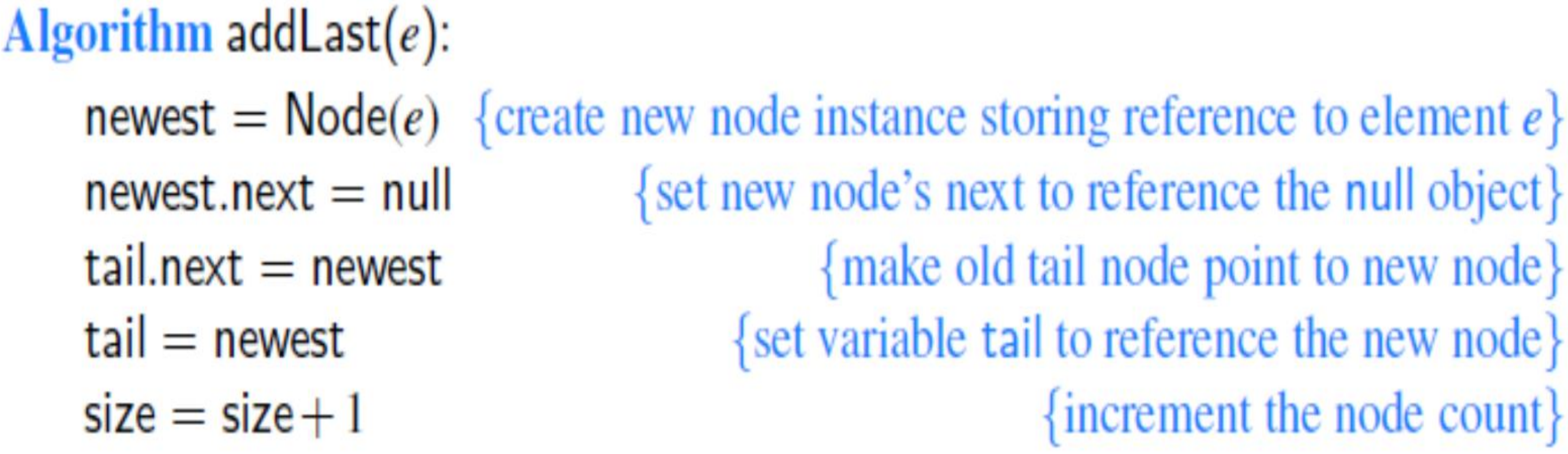
Inserting at the Tail

- 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



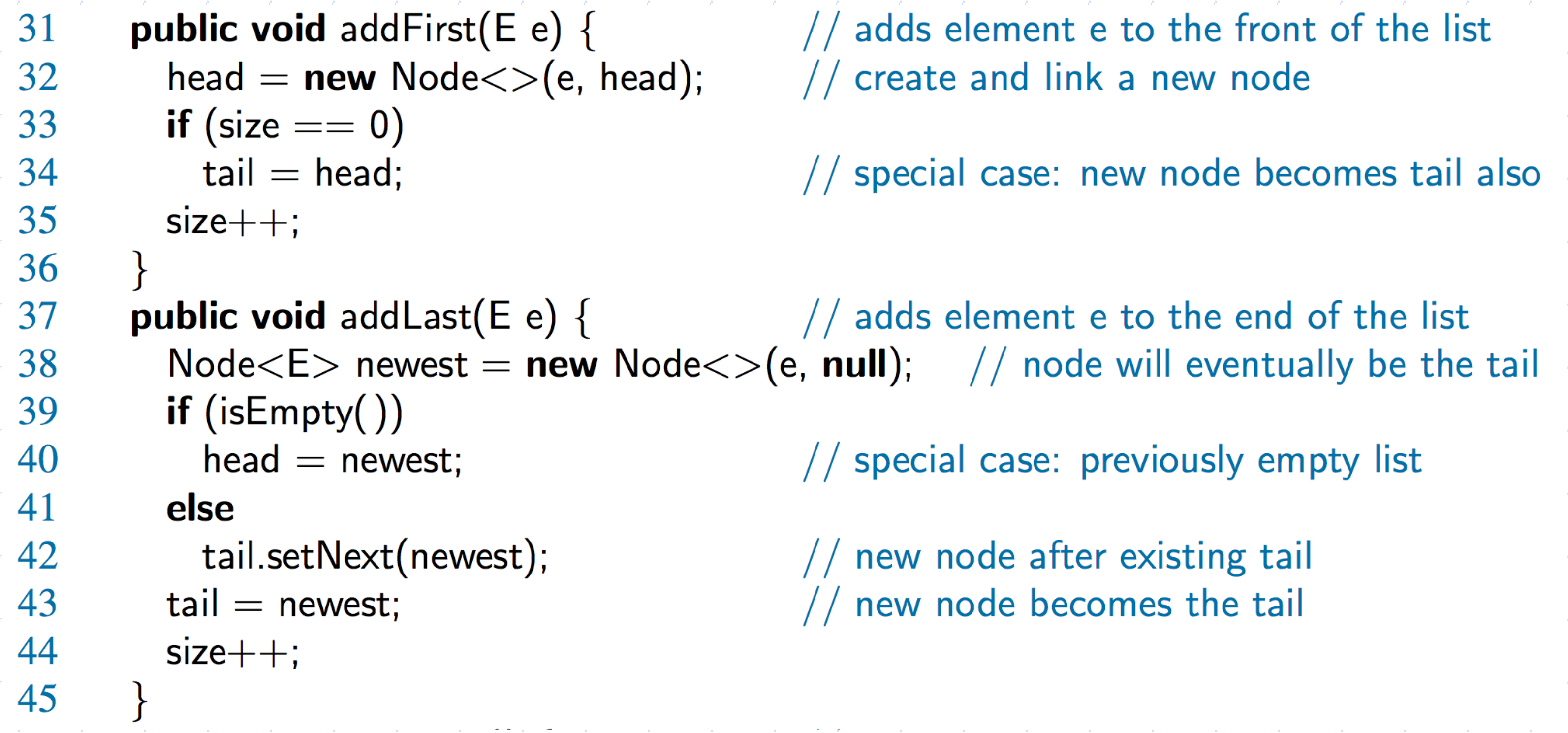
Inserting at the Tail

- 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



```
Algorithm addLast($e$):
    newest = Node($e$)      {create new node instance storing reference to element $e$}
    newest.next = null      {set new node's next to reference the null object}
    tail.next = newest      {make old tail node point to new node}
    tail = newest           {set variable tail to reference the new node}
    size = size + 1         {increment the node count}
```

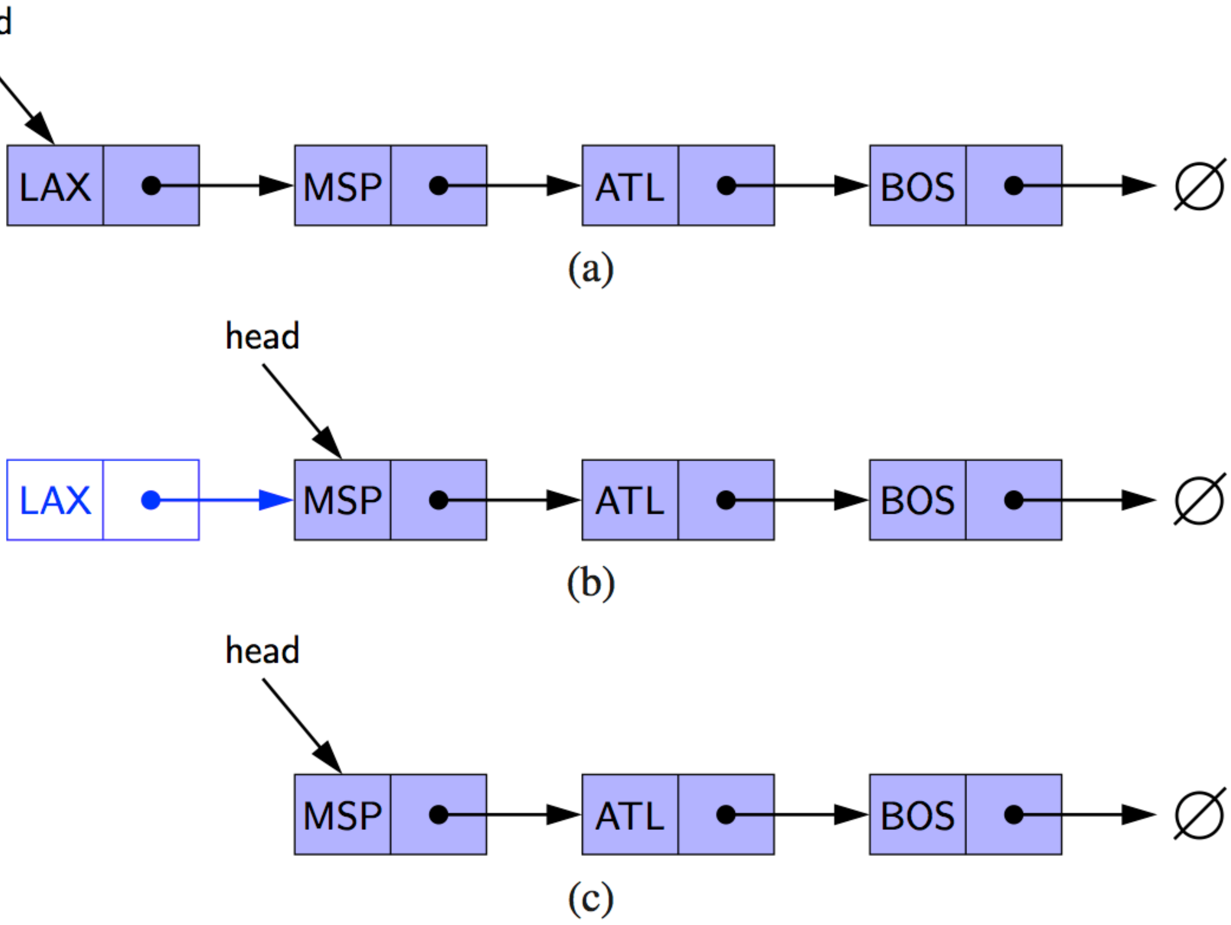
Java Methods



```
public void addFirst(E e) {                    // adds element e to the front of the list
  head = new Node<>(e, head);                  // create and link a new node
  if (size == 0)
    tail = head;                               // special case: new node becomes tail also
  size++;
}
public void addLast(E e) {                     // adds element e to the end of the list
  Node<E> newest = new Node<>(e, null);    // node will eventually be the tail
  if (isEmpty())
    head = newest;                             // special case: previously empty list
  else
    tail.setNext(newest);                      // new node after existing tail
  tail = newest;                               // new node becomes the tail
  size++;
}
```

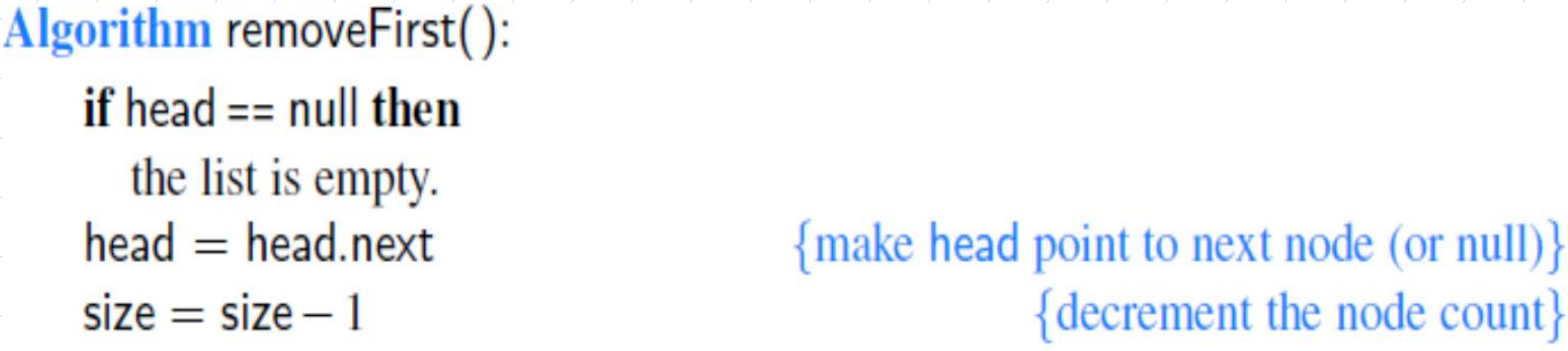
Removing at the Head

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



Removing at the Head

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



```
Algorithm removeFirst( ):
    if head == null then
        the list is empty.
    head = head.next                    {make head point to next node (or null)}
    size = size − 1                     {decrement the node count}
```

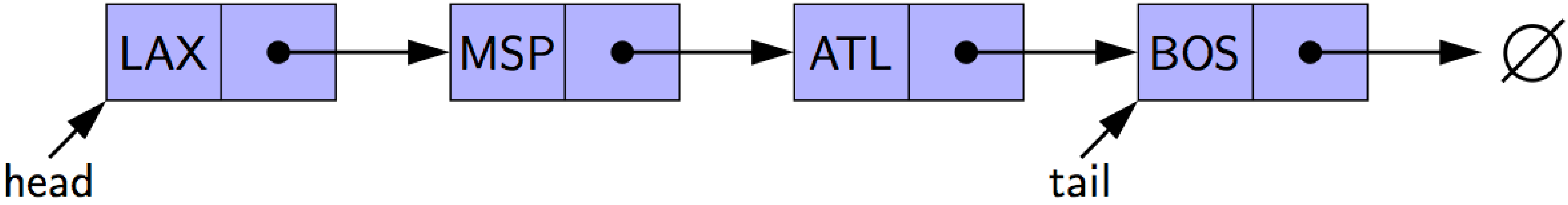
Code Fragment 3.13: Removing the node at the beginning of a singly linked list.

Java Method

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

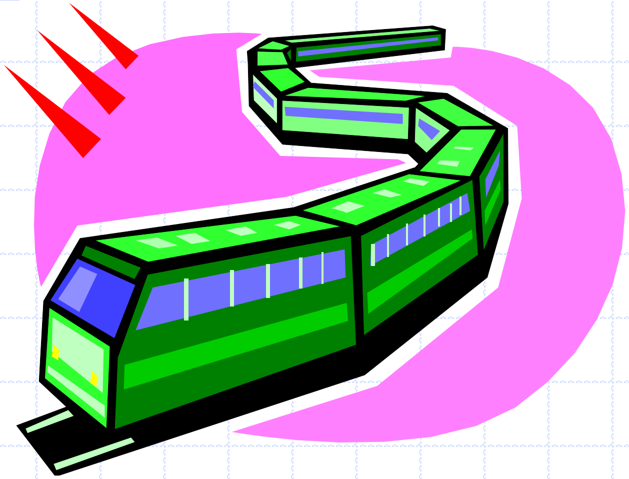
Removing at the Tail

- 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



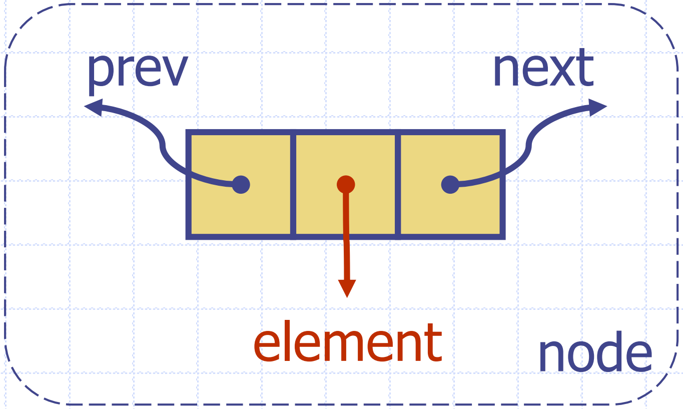
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

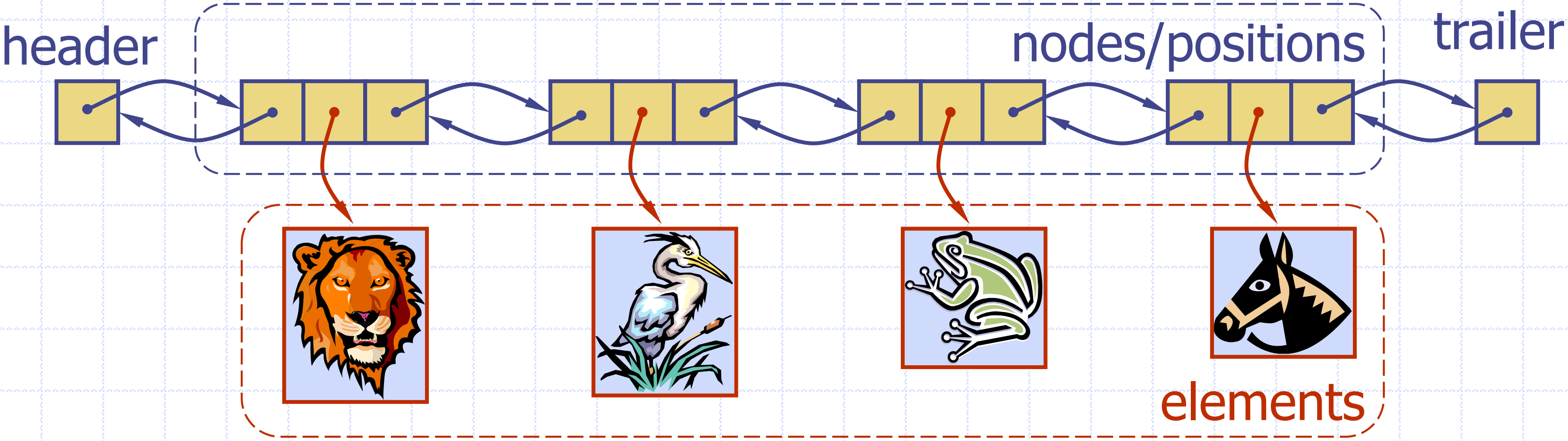
Doubly Linked Lists



Doubly Linked List

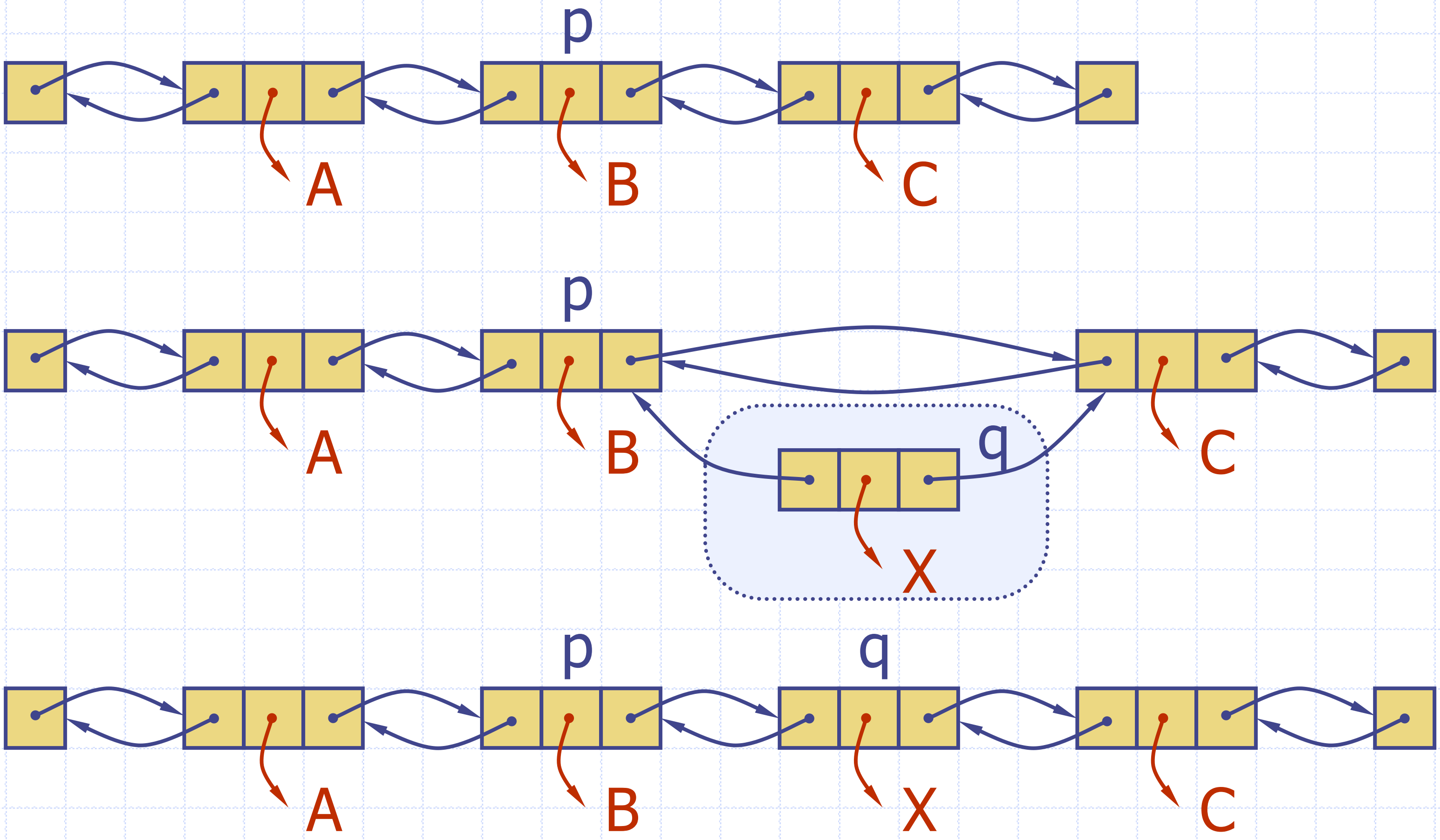
- A doubly linked list can be traversed forward and backward
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special ***trailer*** and ***header*** nodes





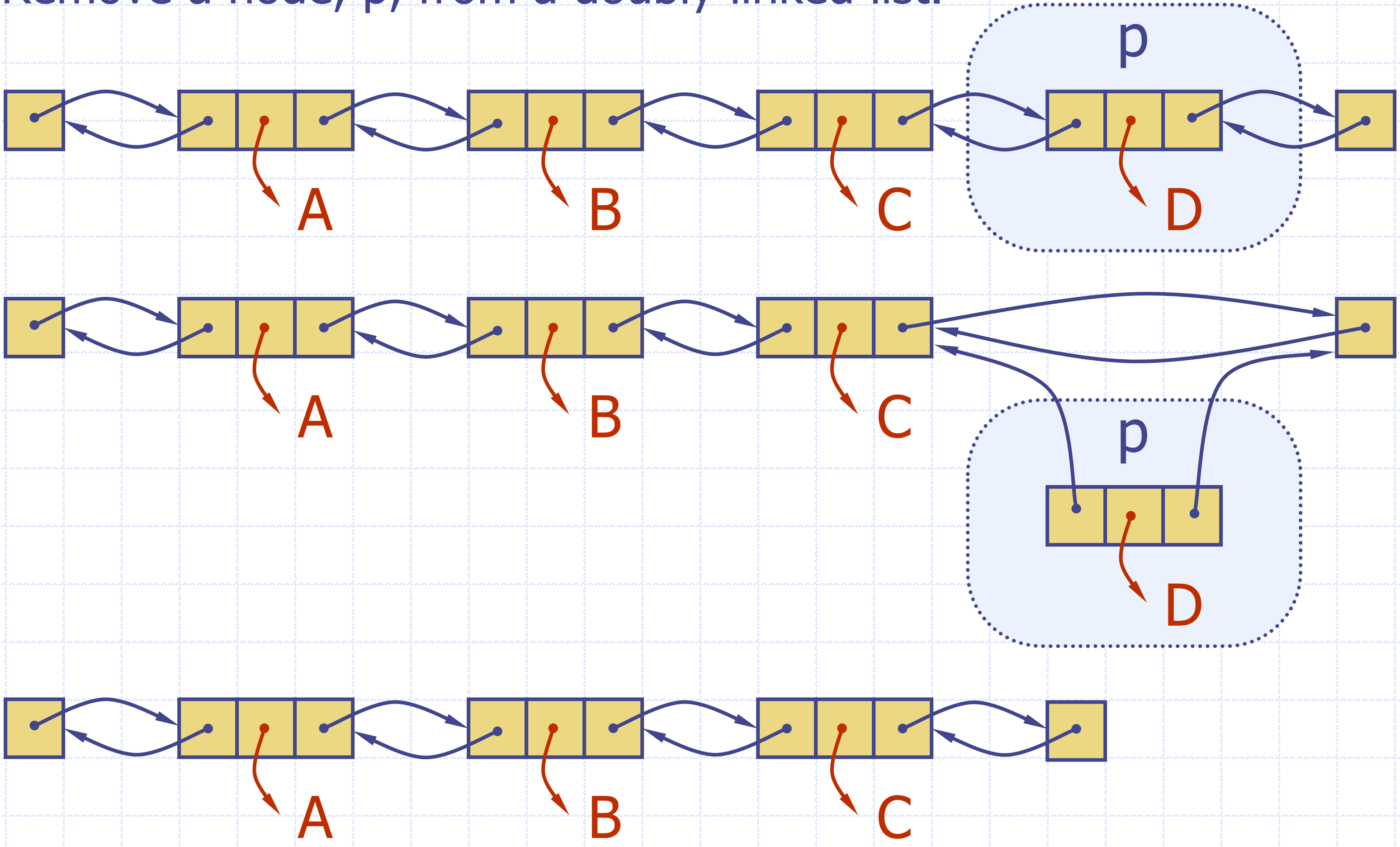
Insertion

- 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

```
/** A basic doubly linked list implementation. */
public class DoublyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> {
    private E element;                  // reference to the element stored at this node
    private Node<E> prev;               // reference to the previous node in the list
    private Node<E> next;               // reference to the subsequent node in the list
    public Node(E e, Node<E> p, Node<E> n) {
      element = e;
      prev = p;
      next = n;
    }
    public E getElement( ) { return element; }
    public Node<E> getPrev( ) { return prev; }
    public Node<E> getNext( ) { return next; }
    public void setPrev(Node<E> p) { prev = p; }
    public void setNext(Node<E> n) { next = n; }
  } //----------- end of nested Node class -----------

```

Doubly-Linked List in Java, 2

```
private Node<E> header;                          // header sentinel
private Node<E> trailer;                         // trailer sentinel
private int size = 0;                            // number of elements in the list
/** Constructs a new empty list. */
public DoublyLinkedList( ) {
  header = new Node<>(null, null, null);         // create header
  trailer = new Node<>(null, header, null);      // trailer is preceded by header
  header.setNext(trailer);                       // header is followed by trailer
}
/** Returns the number of elements in the linked list. */
public int size( ) { return size; }
/** Tests whether the linked list is empty. */
public boolean isEmpty( ) { return size == 0; }
/** Returns (but does not remove) the first element of the list. */
public E first( ) {
  if (isEmpty( )) return null;
  return header.getNext( ).getElement( );        // first element is beyond header
}
/** Returns (but does not remove) the last element of the list. */
public E last( ) {
  if (isEmpty( )) return null;
  return trailer.getPrev( ).getElement( );       // last element is before trailer
}
```

Doubly-Linked List in Java, 3

```
// public update methods
/** Adds element e to the front of the list. */
public void addFirst(E e) {
  addBetween(e, header, header.getNext());       // place just after the header
}
/** Adds element e to the end of the list. */
public void addLast(E e) {
  addBetween(e, trailer.getPrev(), trailer);     // place just before the trailer
}
/** Removes and returns the first element of the list. */
public E removeFirst() {
  if (isEmpty()) return null;                   // nothing to remove
  return remove(header.getNext());              // first element is beyond header
}
/** Removes and returns the last element of the list. */
public E removeLast() {
  if (isEmpty()) return null;                   // nothing to remove
  return remove(trailer.getPrev());             // last element is before trailer
}
```

Doubly-Linked List in Java, 4

```
  // private update methods
  /∗∗ Adds element e to the linked list in between the given nodes. ∗/
  private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
    // create and link a new node
    Node<E> newest = new Node<>(e, predecessor, successor);
    predecessor.setNext(newest);
    successor.setPrev(newest);
    size++;
  }
  /∗∗ Removes the given node from the list and returns its element. ∗/
  private E remove(Node<E> node) {
    Node<E> predecessor = node.getPrev( );
    Node<E> successor = node.getNext( );
    predecessor.setNext(successor);
    successor.setPrev(predecessor);
    size−−;
    return node.getElement( );
  }
} //----------- end of DoublyLinkedList class -----------
```