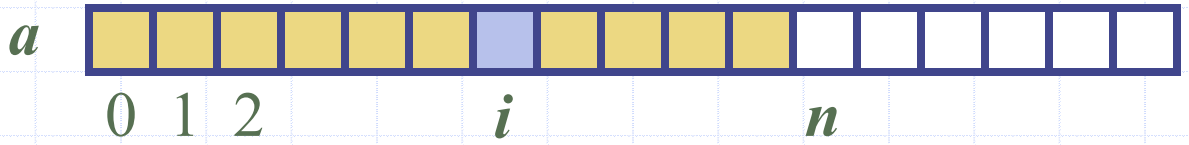
Chapter 3 Fundamental Data Structures

- Arrays
- Linked lists
  - Singly linked lists
  - Doubly linked lists
  - Circularly Linked List

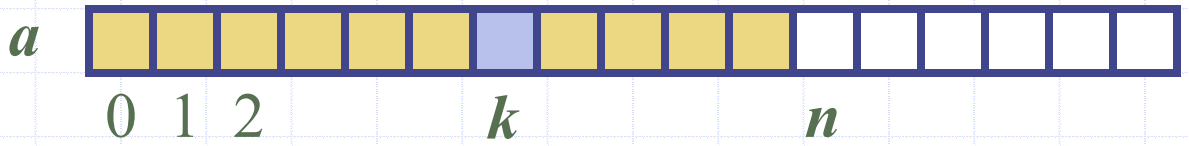
Array Definition

- 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.



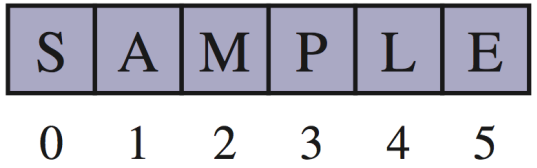
Array Length and Capacity

- 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*].

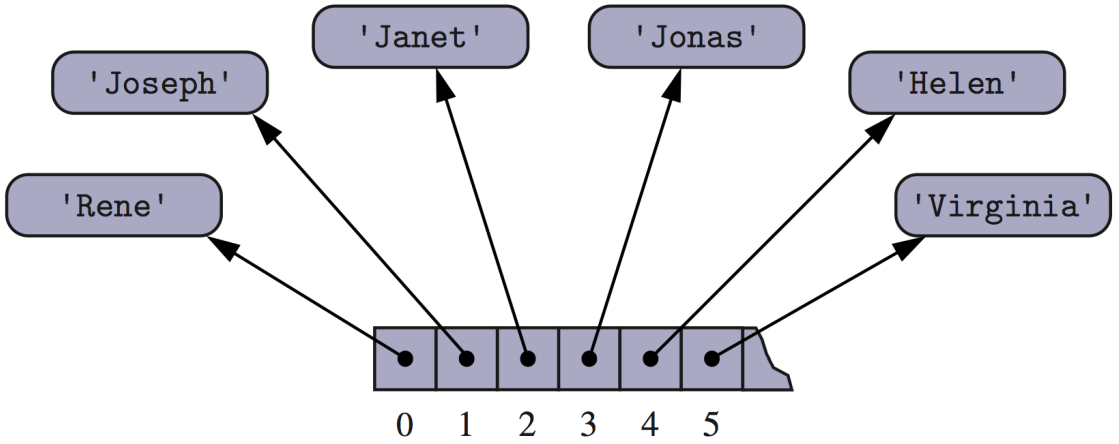


Arrays of Characters or Object References

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



- An array can also store references to objects.



Example of Scoreboard

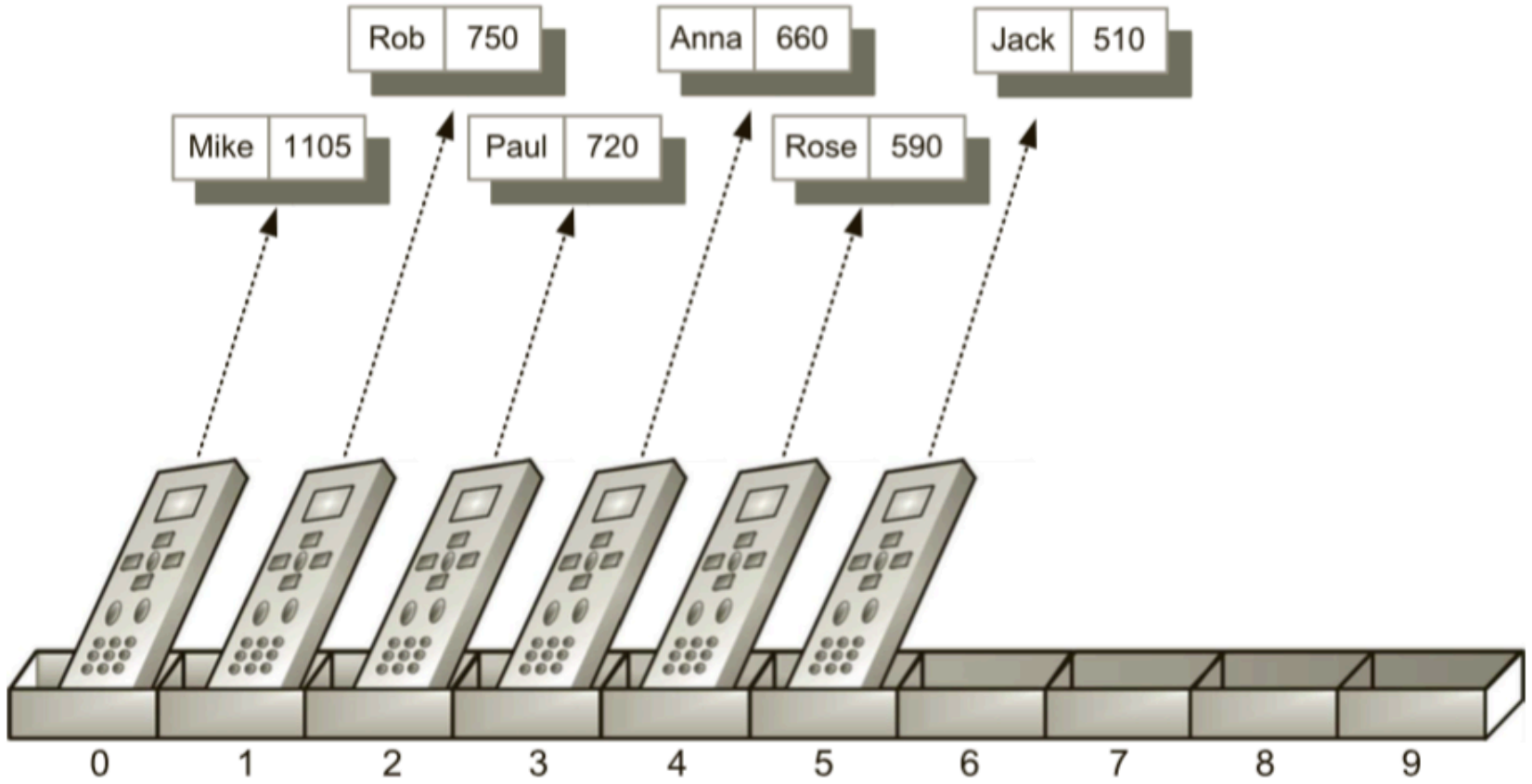


Figure 3.1: An illustration of an array of length ten storing references to six GameEntry objects in the cells with indices 0 to 5; the rest are **null** references.

Java Example: Game Entries

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

```
1  public class GameEntry {
2    private String name;                    // name of the person earning this score
3    private int score;                      // the score value
4    /** Constructs a game entry with given parameters.. */
5    public GameEntry(String n, int s) {
6      name = n;
7      score = s;
8    }
9    /** Returns the name field. */
10   public String getName( ) { return name; }
11   /** Returns the score field. */
12   public int getScore( ) { return score; }
13   /** Returns a string representation of this entry. */
14   public String toString( ) {
15     return "(" + name + ",  " + score + ")";
16   }
17 }
```

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

```
1   /** Class for storing high scores in an array in nondecreasing order. */
2   public class Scoreboard {
3     private int numEntries = 0;                  // number of actual entries
4     private GameEntry[ ] board;                  // array of game entries (names & scores)
5     /** Constructs an empty scoreboard with the given capacity for storing entries. */
6     public Scoreboard(int capacity) {
7       board = new GameEntry[capacity];
8     }
...   // more methods will go here
36  }
```

Example of Scoreboard

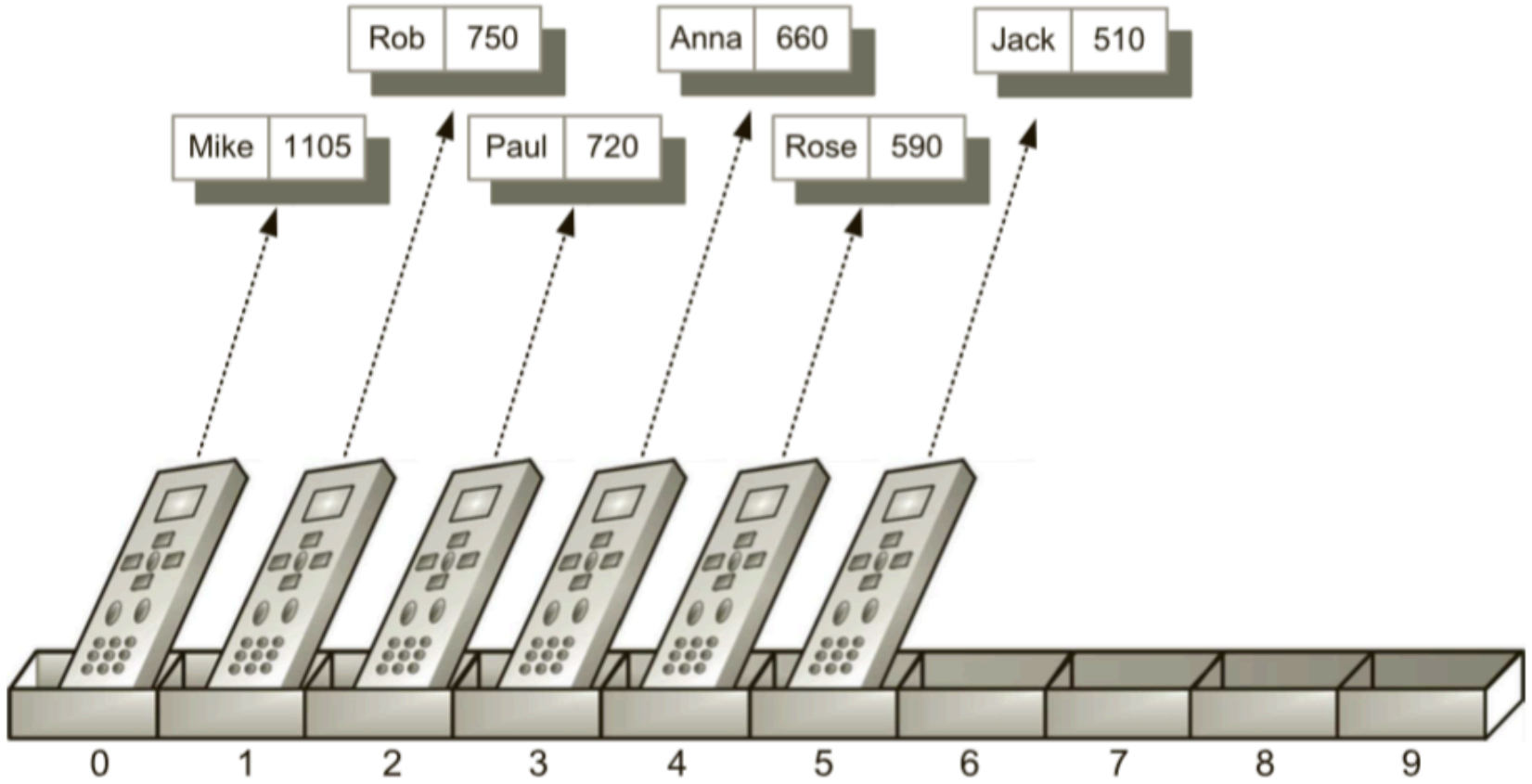
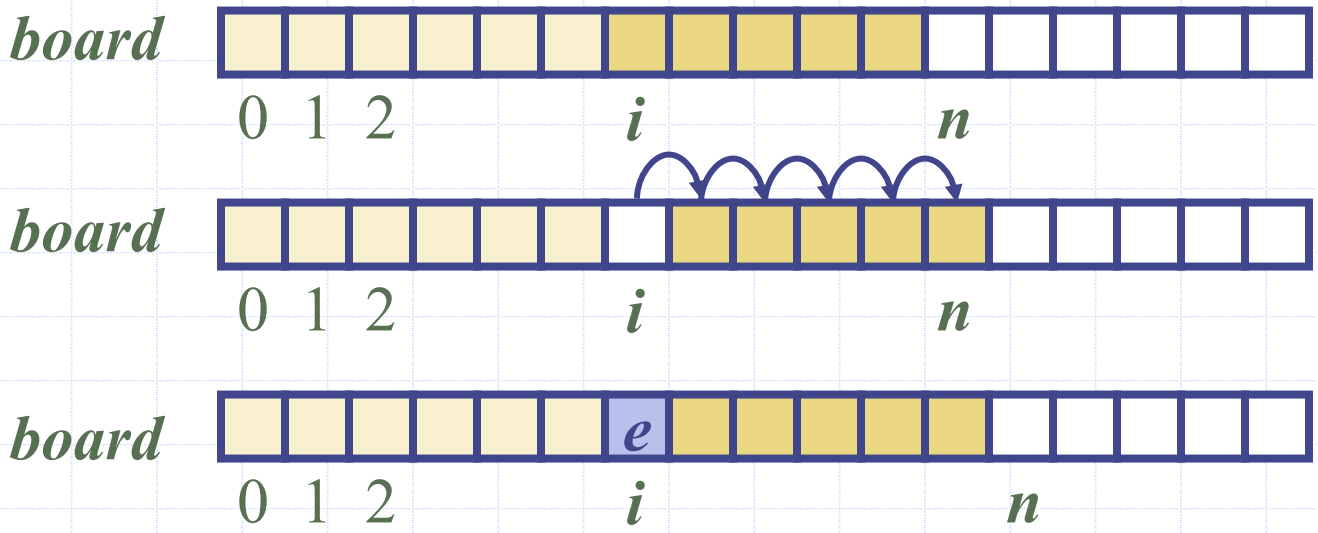


Figure 3.1: An illustration of an array of length ten storing references to six GameEntry objects in the cells with indices 0 to 5; the rest are **null** references.

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 $n - i$ entries $board[i], \ldots, board[n-1]$

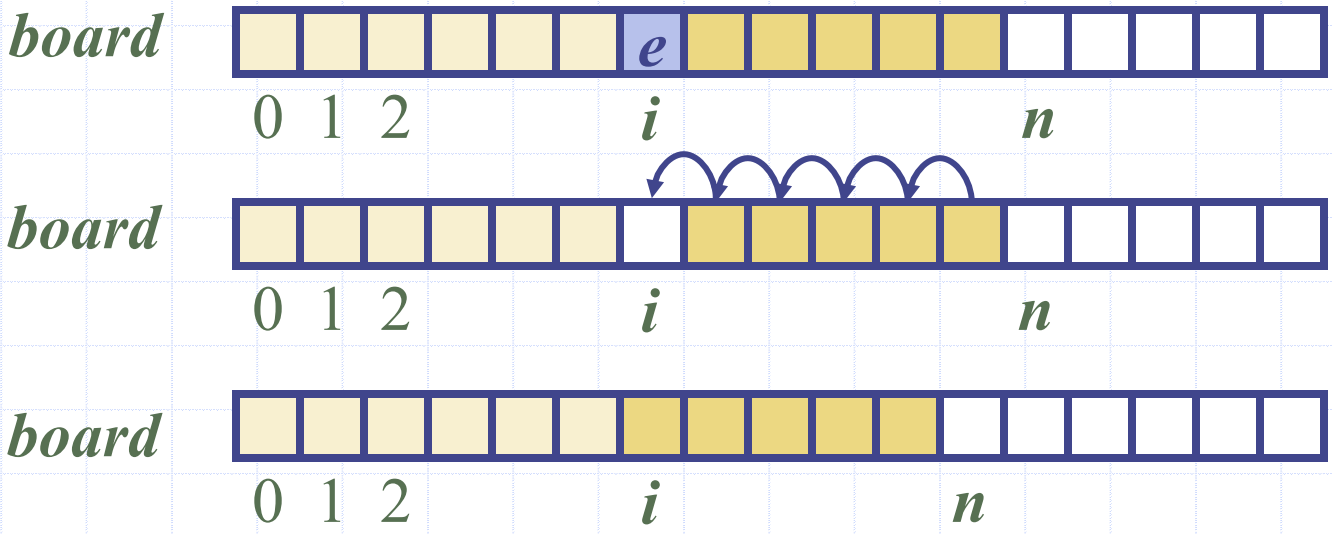


Java Example

```
 9   /** Attempt to add a new score to the collection (if it is high enough) */
10   public void add(GameEntry e) {
11     int newScore = e.getScore( );
12     // is the new entry e really a high score?
13     if (numEntries < board.length || newScore > board[numEntries−1].getScore( )) {
14       if (numEntries < board.length)                    // no score drops from the board
15         numEntries++;                                   // so overall number increases
16       // shift any lower scores rightward to make room for the new entry
17       int j = numEntries − 1;
18       while (j > 0 && board[j−1].getScore( ) < newScore) {
19         board[j] = board[j−1];                          // shift entry from j-1 to j
20         j−−;                                            // and decrement j
21       }
22       board[j] = e;                                     // when done, add new entry
23     }
24   }
```

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 $board[i + 1], \ldots, board[n - 1]$



Java Example

```
25    /** Remove and return the high score at index i. */
26    public GameEntry remove(int i) throws IndexOutOfBoundsException {
27      if (i < 0 || i >= numEntries)
28        throw new IndexOutOfBoundsException("Invalid index: " + i);
29      GameEntry temp = board[i];                    // save the object to be removed
30      for (int j = i; j < numEntries - 1; j++)      // count up from i (not down)
31        board[j] = board[j+1];                      // move one cell to the left
32      board[numEntries -1 ] = null;                 // null out the old last score
33      numEntries--;
34      return temp;                                  // return the removed object
35    }
```

Sorting an Array

- The Insertion-Sort Algorithm

Algorithm InsertionSort(A):

Input: An array A of n comparable elements

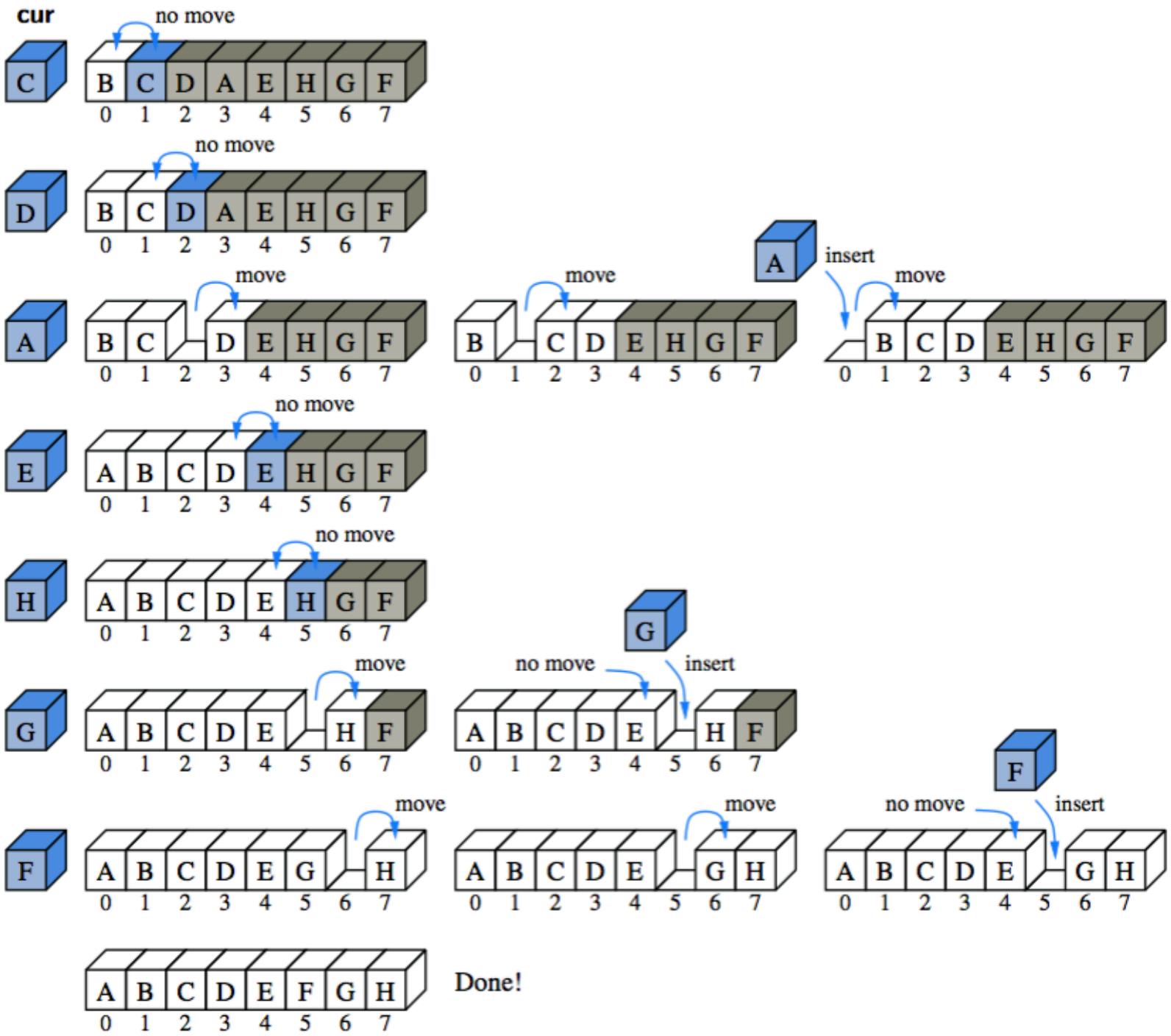
Output: The array A with elements rearranged in nondecreasing order

for k from 1 to $n-1$ **do**

Insert $A[k]$ at its proper location within $A[0], A[1], \ldots, A[k]$.

Code Fragment 3.5: High-level description of the insertion-sort algorithm.

Example


cur
no move
C
B C D A E H G F
0 1 2 3 4 5 6 7
no move
D
B C D A E H G F
0 1 2 3 4 5 6 7
move
A
B C D E H G F
0 1 2 3 4 5 6 7
move
A
insert
B C D E H G F
0 1 2 3 4 5 6 7
move
B C D E H G F
0 1 2 3 4 5 6 7
no move
E
A B C D E H G F
0 1 2 3 4 5 6 7
no move
H
A B C D E H G F
0 1 2 3 4 5 6 7
move
G
A B C D E H F
0 1 2 3 4 5 6 7
G
no move
insert
A B C D E H F
0 1 2 3 4 5 6 7
move
F
A B C D E G H
0 1 2 3 4 5 6 7
move
A B C D E G H
0 1 2 3 4 5 6 7
F
no move
insert
A B C D E G H
0 1 2 3 4 5 6 7
A B C D E F G H
0 1 2 3 4 5 6 7
Done!

Java Code

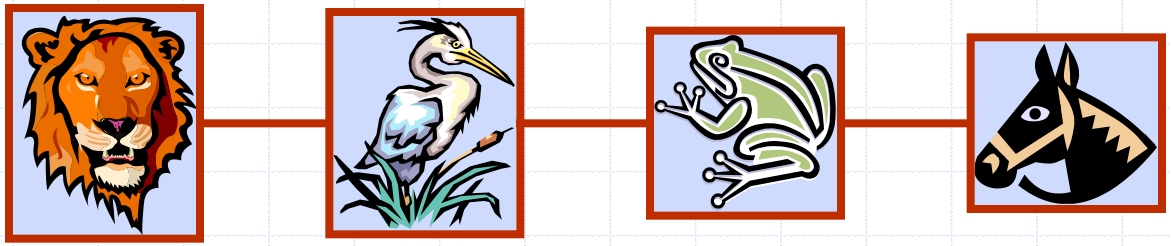
```
1   /** Insertion-sort of an array of characters into nondecreasing order */
2   public static void insertionSort(char[ ] data) {
3     int n = data.length;
4     for (int k = 1; k < n; k++) {                  // begin with second character
5       char cur = data[k];                          // time to insert cur=data[k]
6       int j = k;                                   // find correct index j for cur
7       while (j > 0 && data[j−1] > cur) {           // thus, data[j-1] must go after cur
8         data[j] = data[j−1];                       // slide data[j-1] rightward
9         j−−;                                       // and consider previous j for cur
10      }
11      data[j] = cur;                               // this is the proper place for cur
12    }
13  }
```

Code Fragment 3.6: Java code for performing insertion-sort on a character array.

Methods of java.util.Arrays

- equals(*A*, *B*)
- fill(*A*, *x*)
- copyOf(*A*, *n*)
- copyOfRange(*A*, *s*, *t*)
- toString(*A*)
- sort(*A*)
- binarySearch(*A*, *x*)

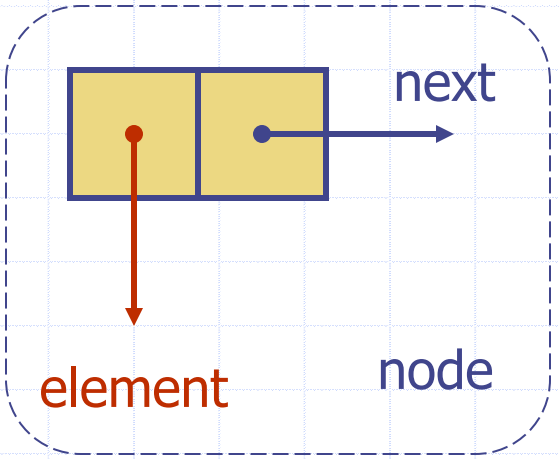
Singly Linked Lists

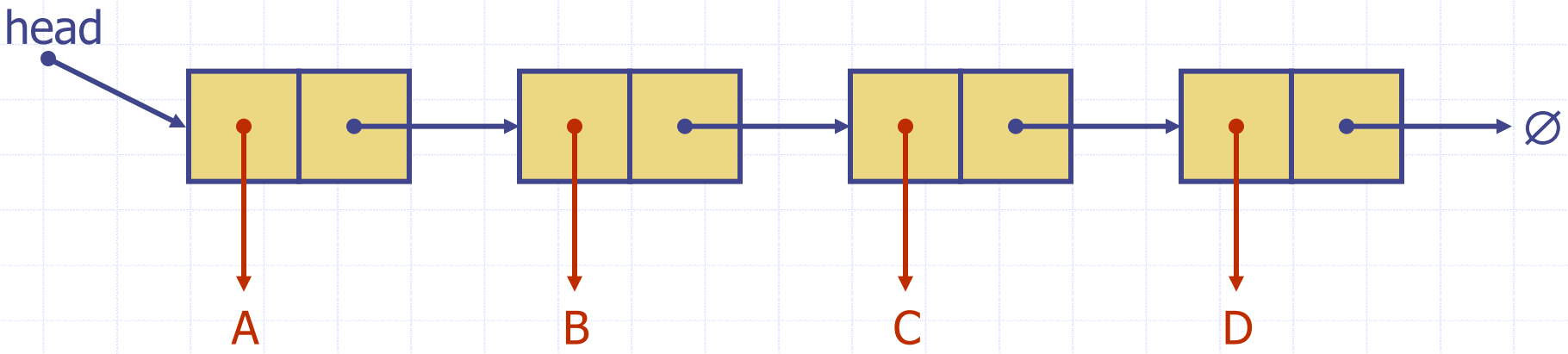


Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Singly Linked 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

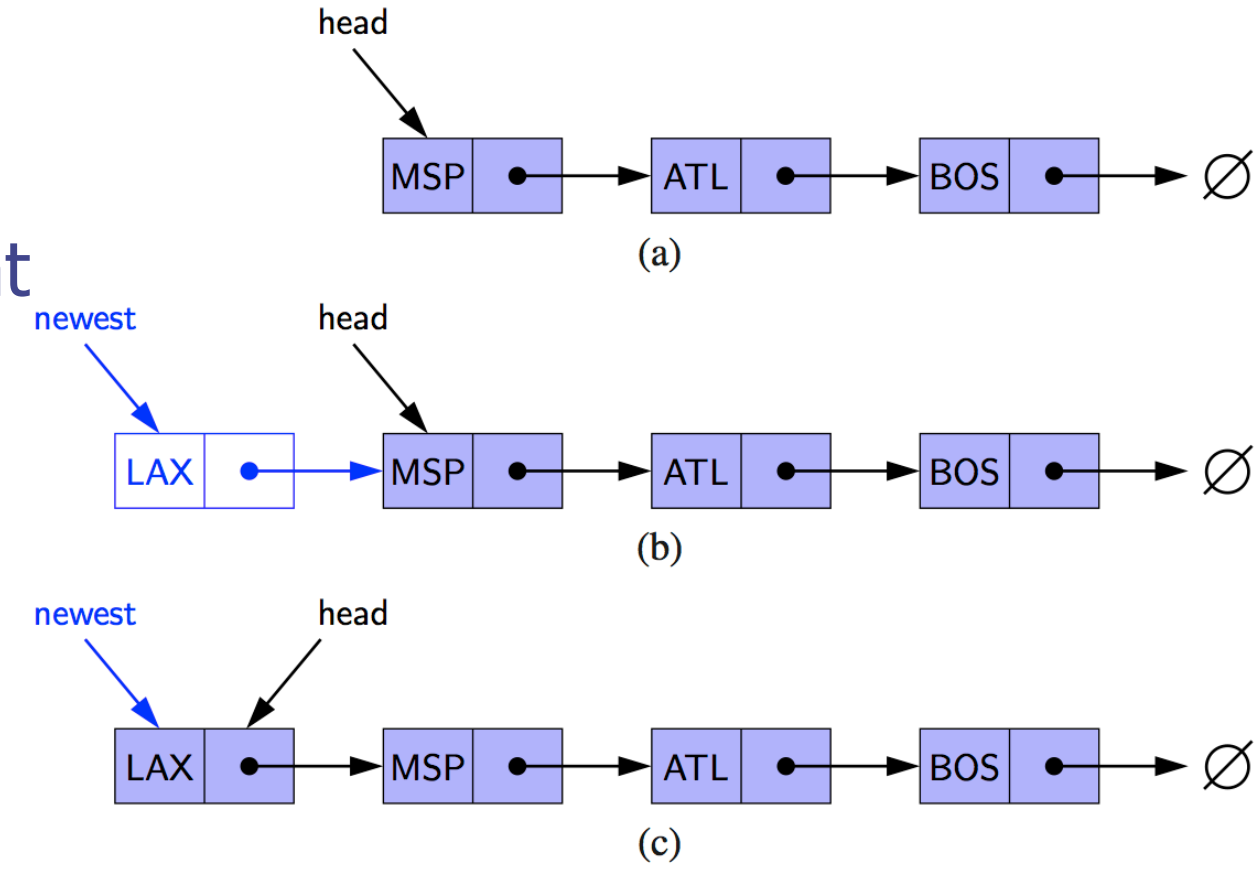
```
1   public class SinglyLinkedList<E> {
2     //---------------- nested Node class ----------------
3     private static class Node<E> {
4       private E element;                  // reference to the element stored at this node
5       private Node<E> next;               // reference to the subsequent 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    }
```

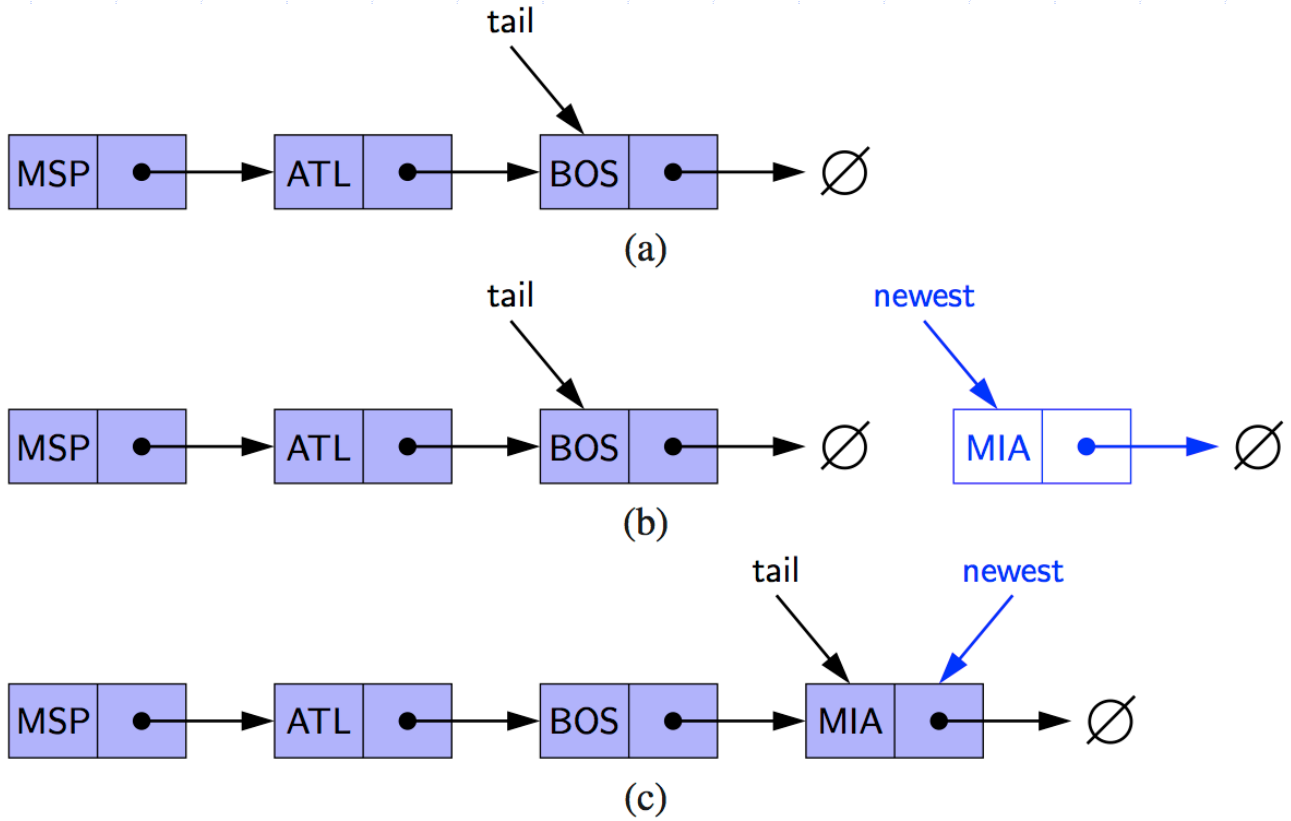
Inserting at the Head

- 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
- Have old last node point to new node
- Update tail to point to new node

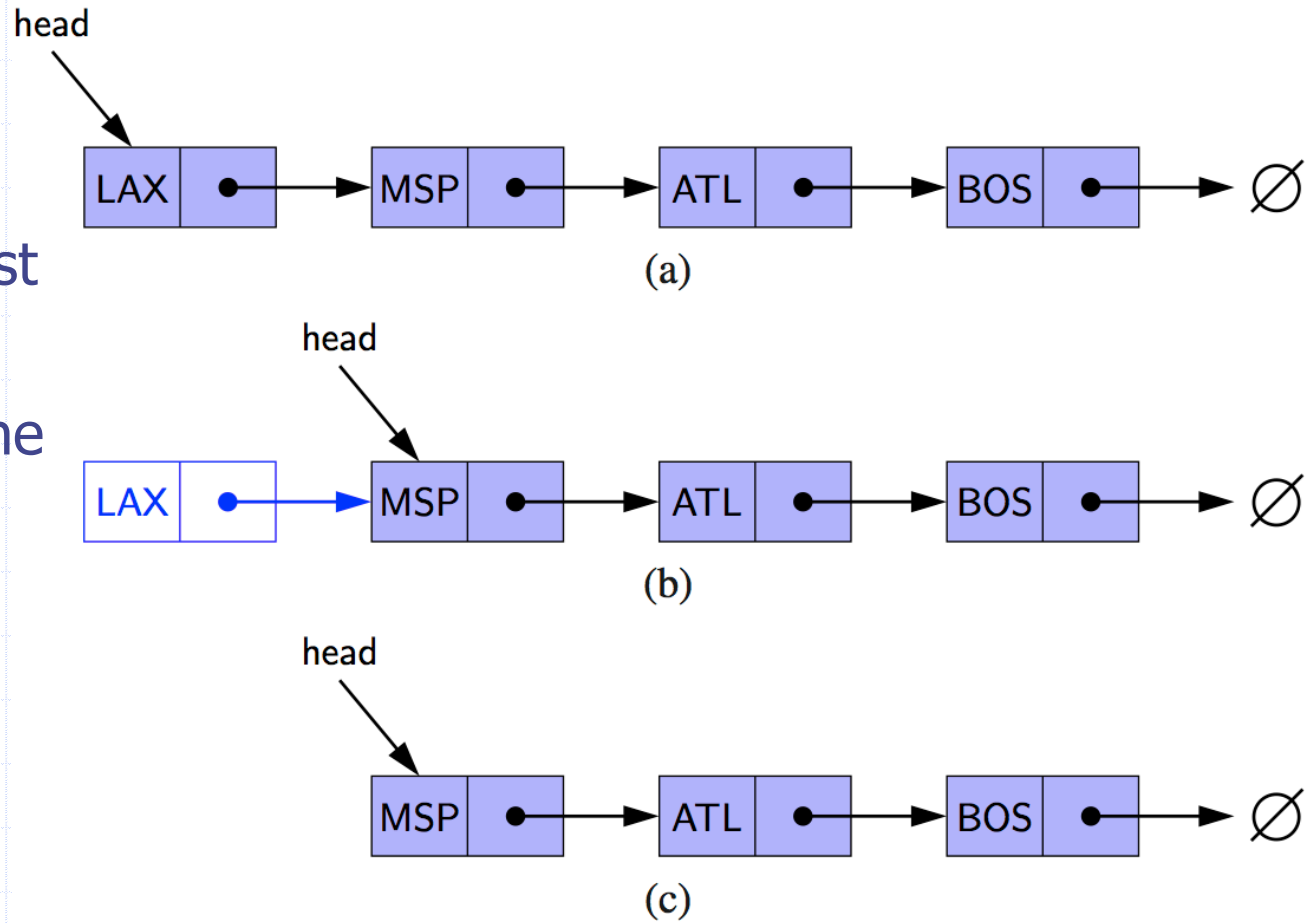


Java Methods

```
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

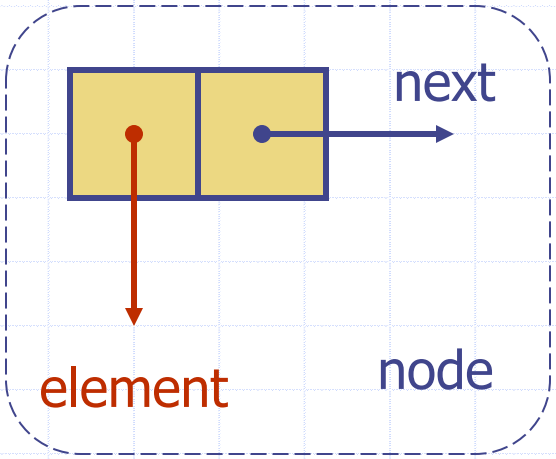


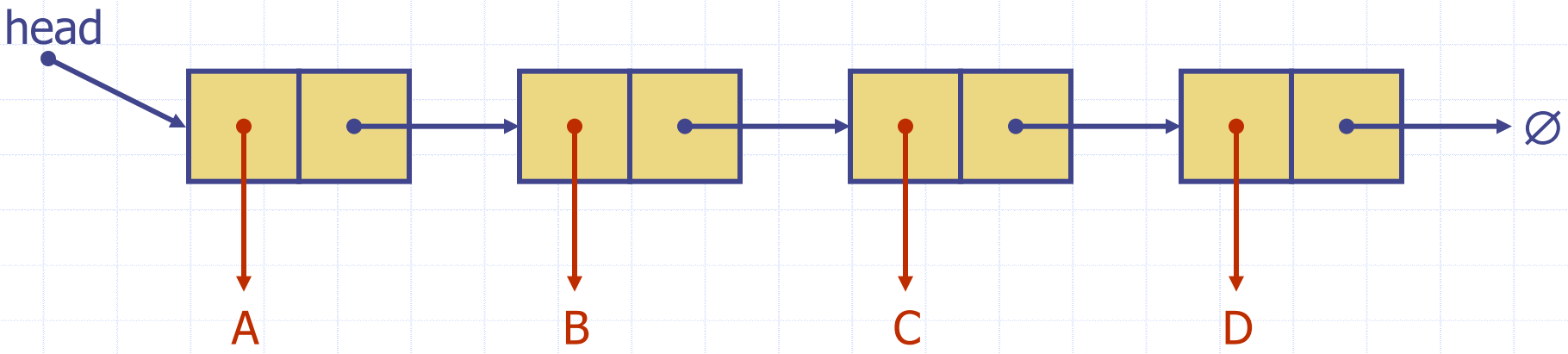
Java Method

```
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  }
```

Recaps of Singly Linked List

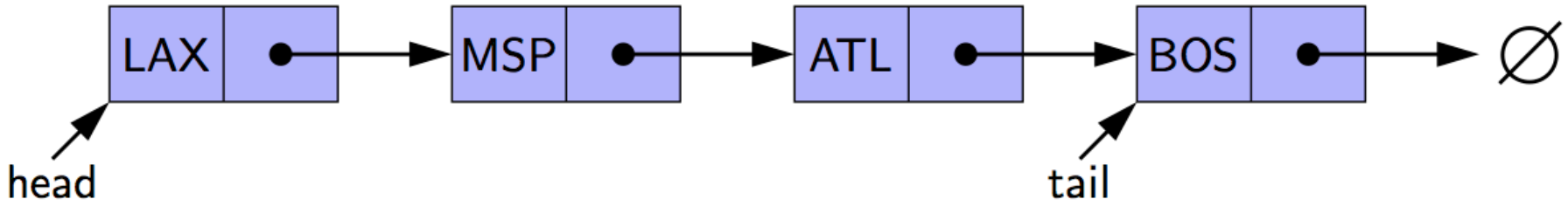
- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - element
  - link to the next node





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



A Solution

```
Algorithm removeLast (list)
    If list.size == 0
        return;
    If list.size == 1
        list.head = null; list.tail = null; list.size = 0

    prev = list.head;
    curr = list.head.next;
    while(curr.next != null)
        prev = curr;
        curr = curr.next;
    prev.next = null;
    list.tail = prev;
```

Circularly Linked Lists

- Round-Robin Scheduling
  - Each active process is given its own time slice
  - A process is interrupted when the slice ends
  - Processes take turns in a cyclic order

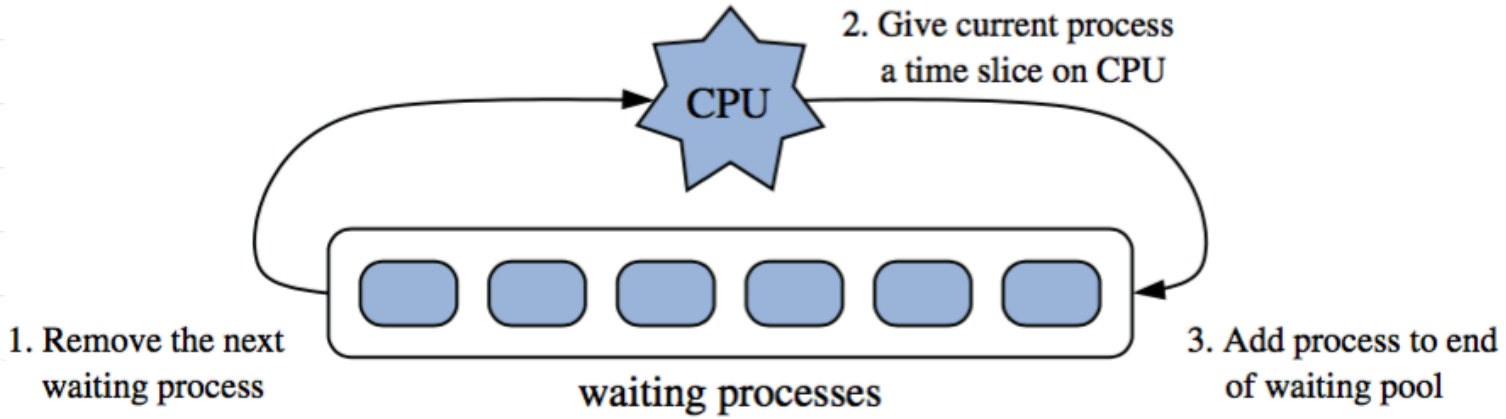


Figure 3.15: The three iterative steps for round-robin scheduling.

Circularly Linked List

- One additional update method:
  - rotate(): Moves the first element to the end of the list.

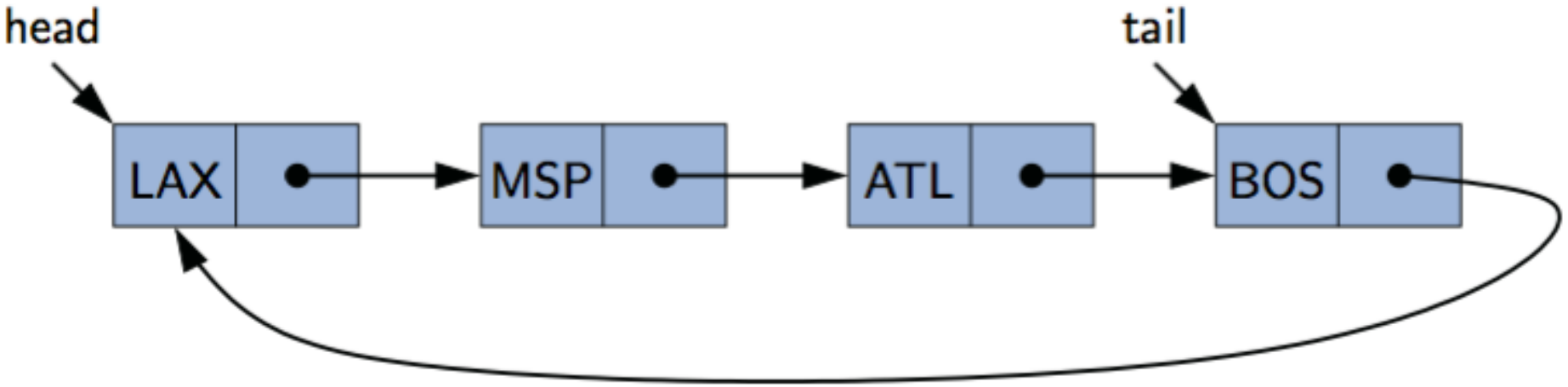


Figure 3.16: Example of a singly linked list with circular structure.

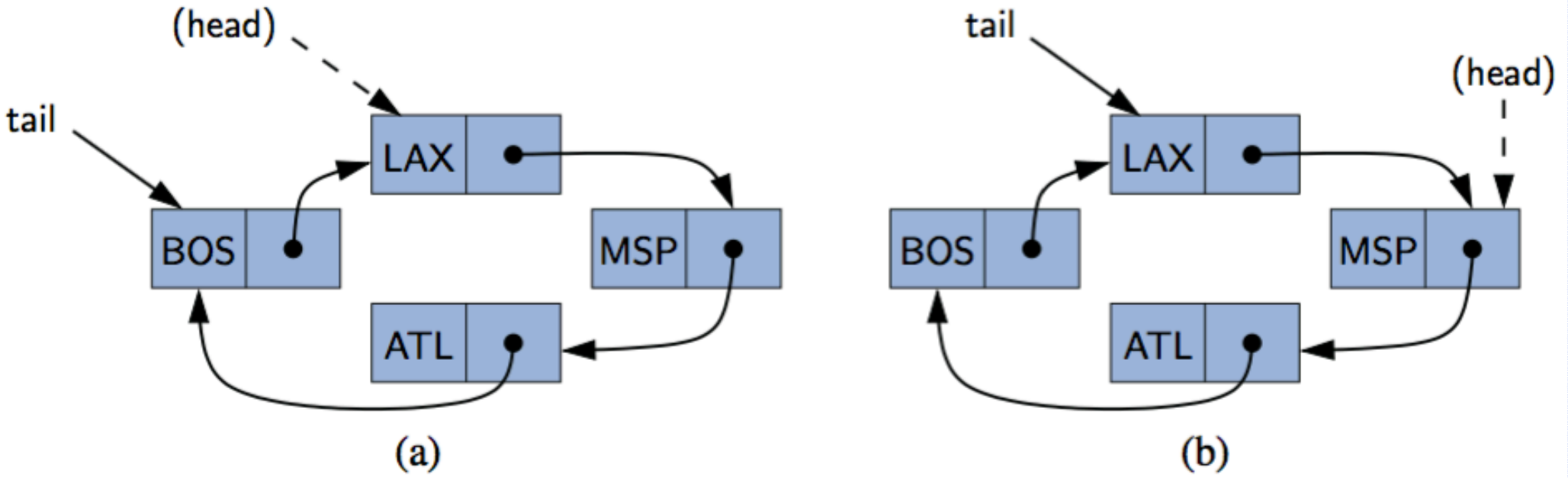
Implementing Round-Robin

- Let C be a circularly linked list. Round-Robin can be implemented by repeatedly performing the following two steps:
  - Give a time slice to process *C*.first()
  - *C*.rotate()

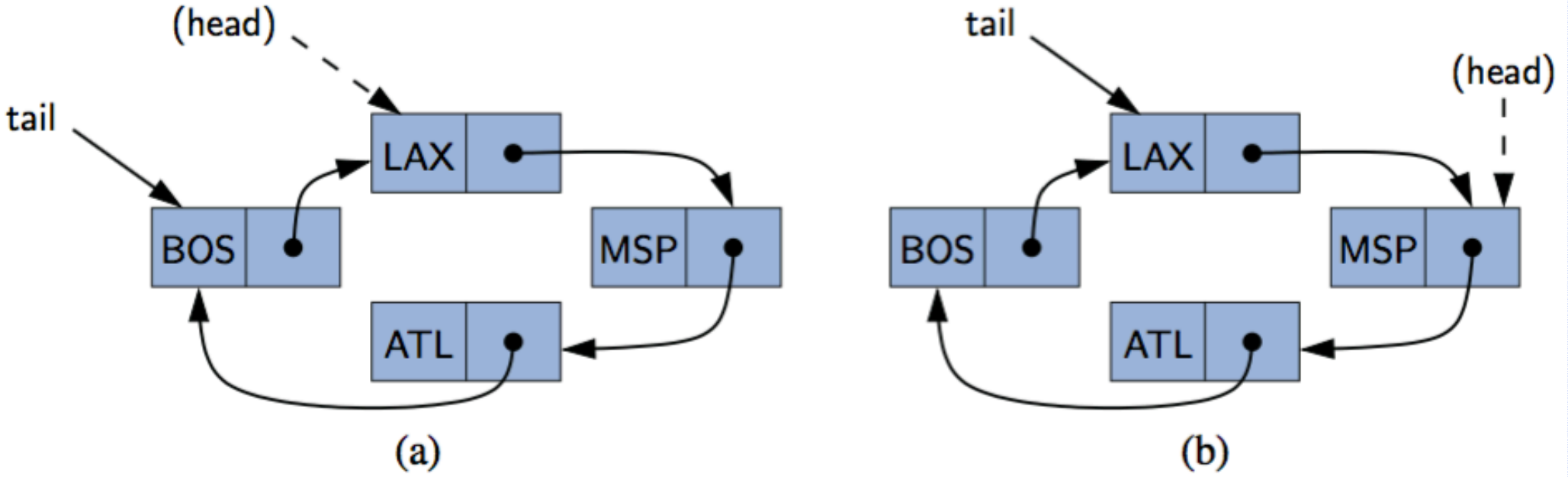
Design Issues

- We maintain a reference to the tail
- We can locate the head as tail.getNext()
- No longer need maintain the head reference.

How to Rotate?

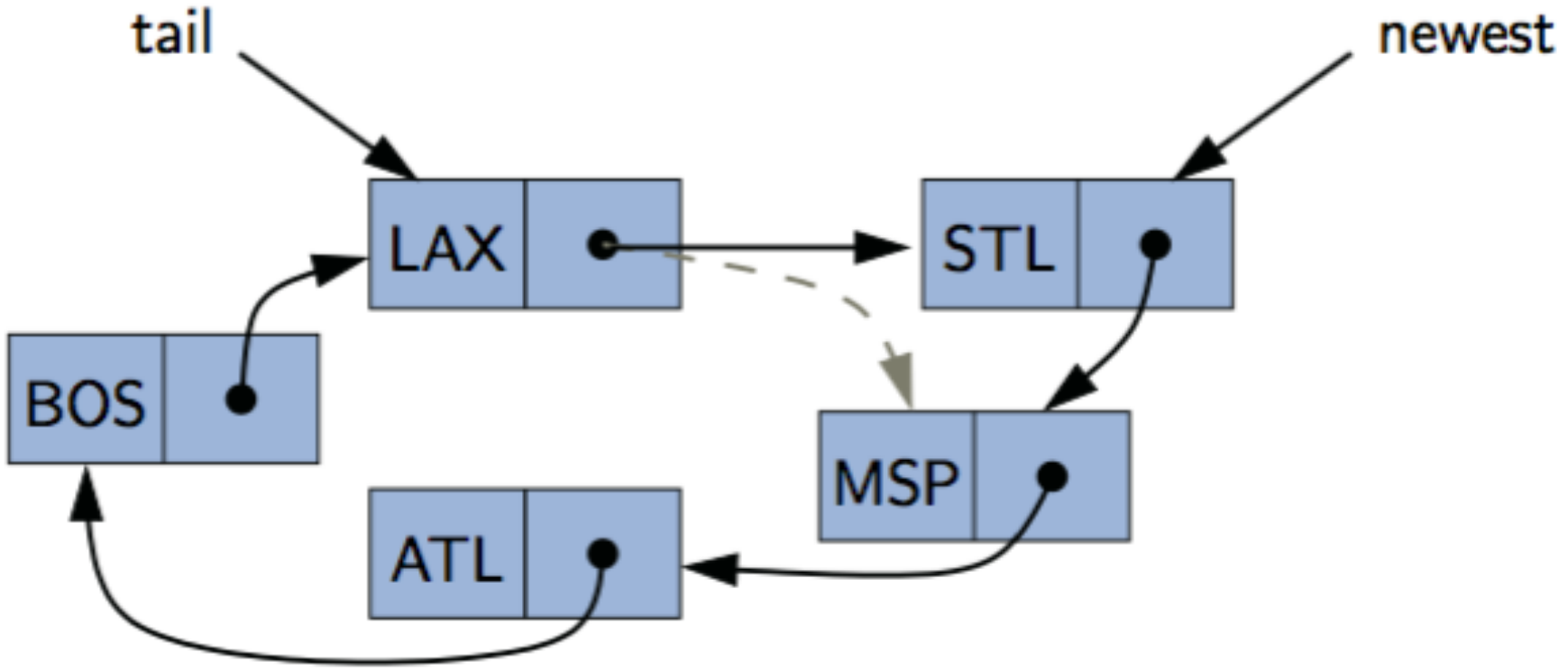


How to Rotate?

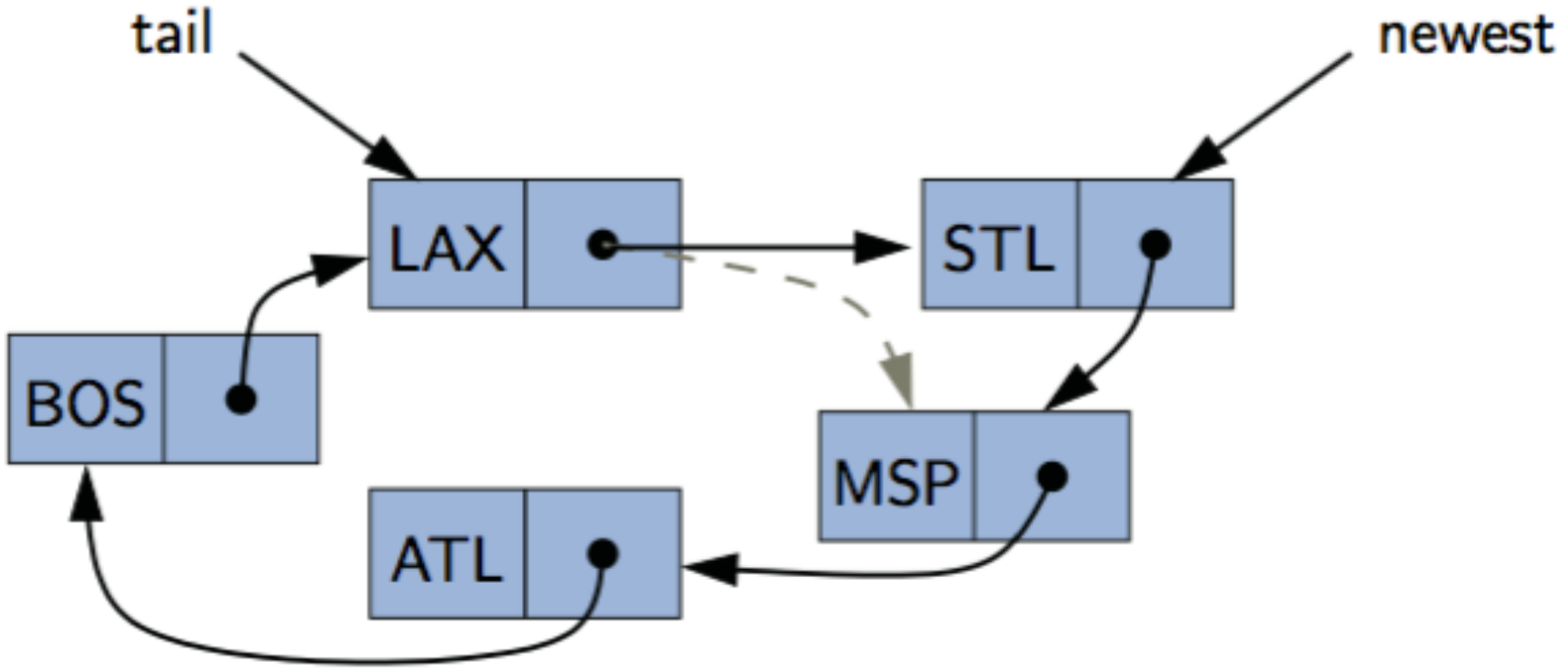



```
public void rotate( ) {                  // rotate the first element to the back of the list
  if (tail != null)                      // if empty, do nothing
    tail = tail.getNext( );              // the old head becomes the new tail
}
```

How to Add a New Node at the Front?



How to Add a New Node at the Front?



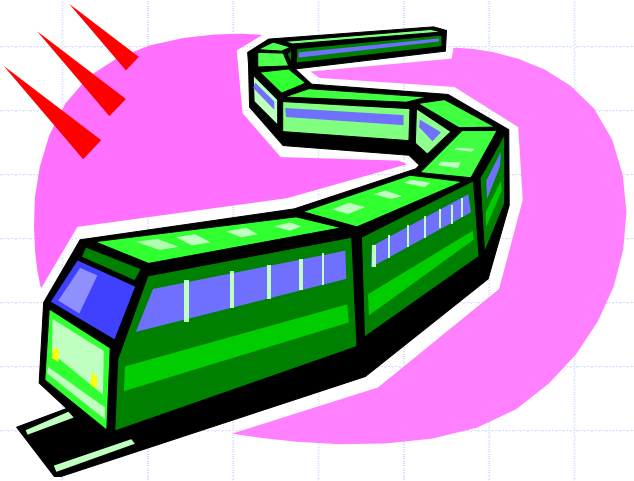

```
Node<E> newest = new Node<>(e, tail.getNext( ));
tail.setNext(newest);
```

How to Remove the First Node?

```
public E removeFirst( ) {                          // removes and returns the first element
   if (isEmpty( )) return null;                    // nothing to remove
   Node<E> head = tail.getNext( );
   if (head == tail) tail = null;                  // must be the only node left
   else tail.setNext(head.getNext( ));             // removes "head" from the list
   size−−;
   return head.getElement( );
}
```

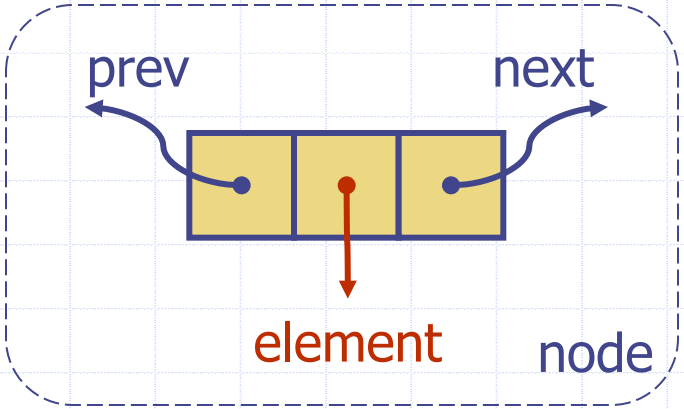
Doubly Linked Lists

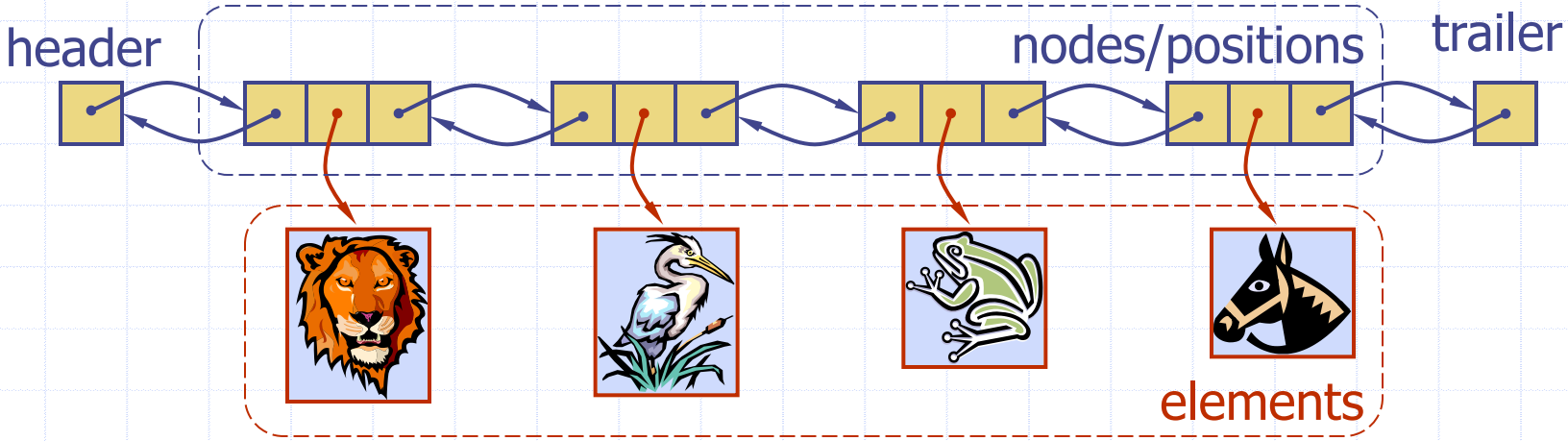
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014



Doubly Linked List

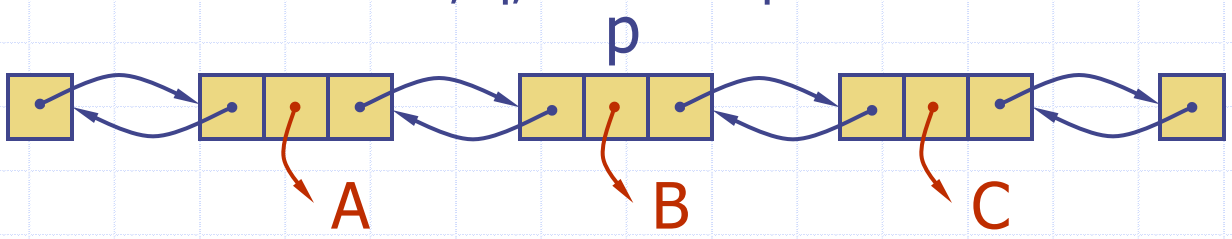
- A doubly linked list can be traversed forward and backward
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





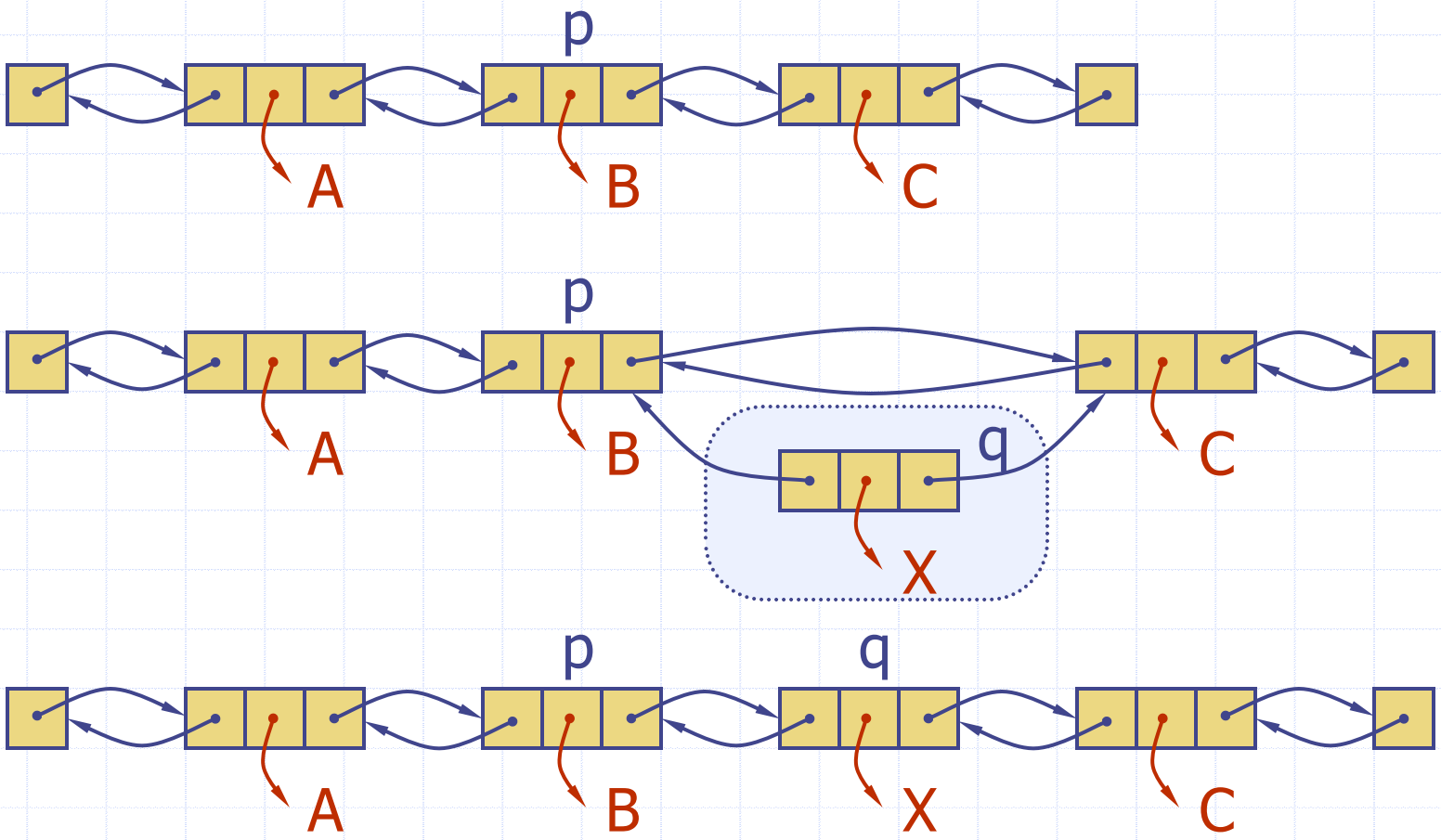
Insertion

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



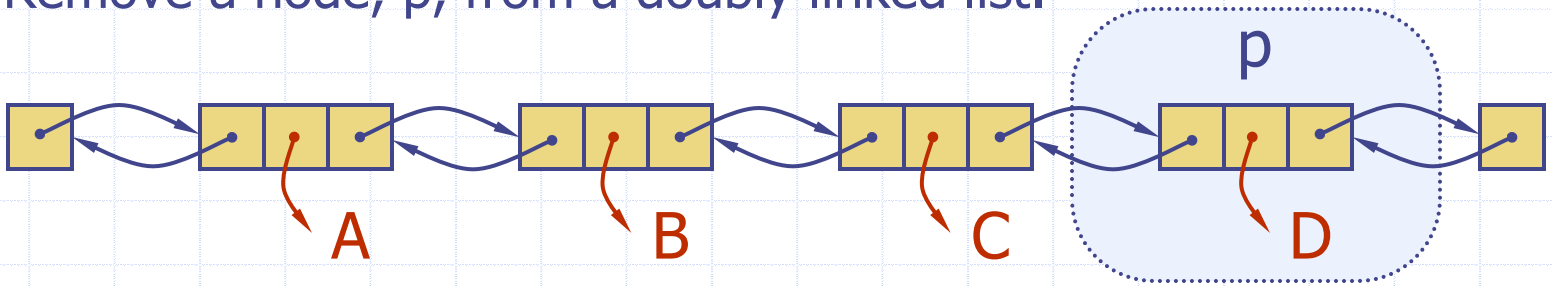
Insertion

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



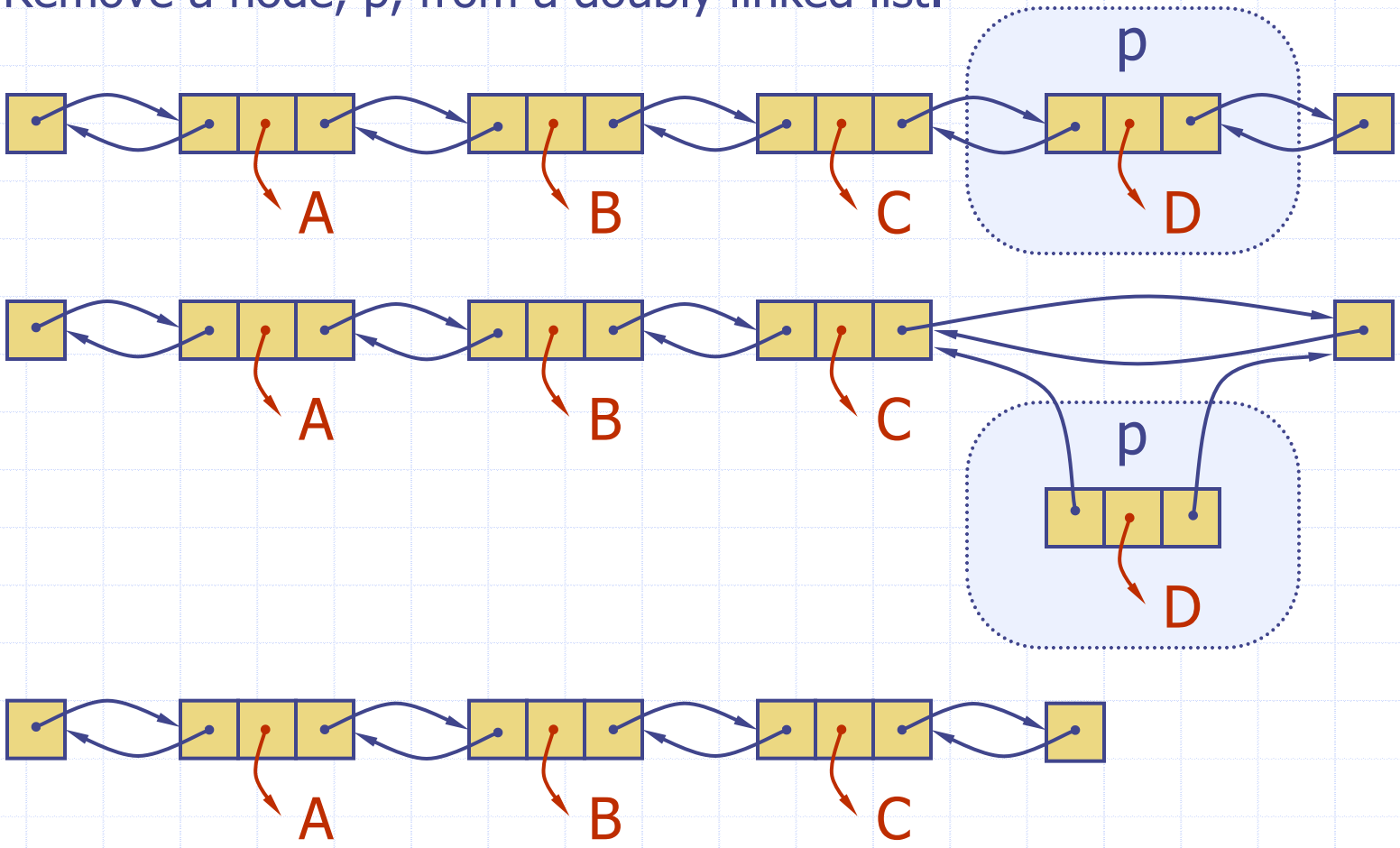
Deletion

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



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
70
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
78
79
80      return node.getElement( );
81    }
82  } //---------- end of DoublyLinkedList class ----------
```

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
78
79
80      return node.getElement( );
81    }
82  } //---------- end of DoublyLinkedList class ----------
```

Comparing Arrays with Linked Lists

- Searching
- Insertion
- Deletion

Exercise C-3.27

- Describe in detail how to swap two nodes x and y (and not just their contents) in a singly linked list L given references only to x and y.