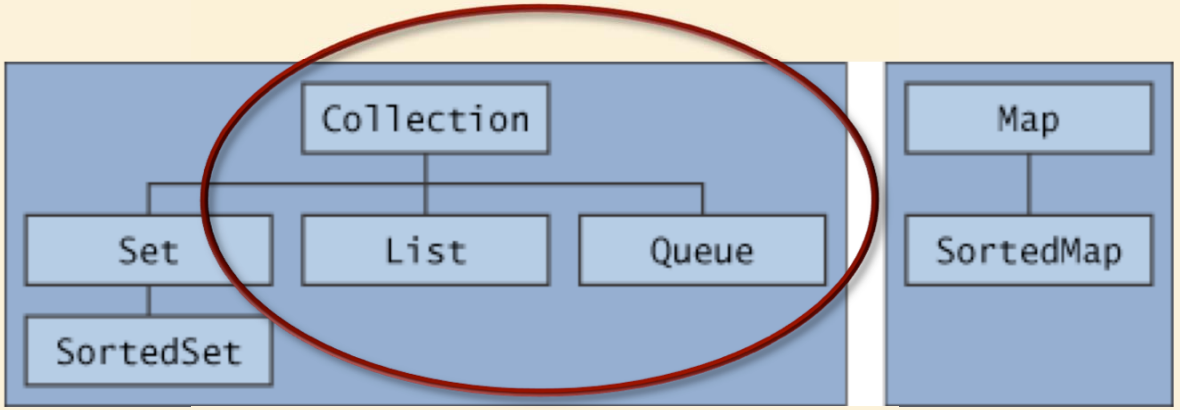
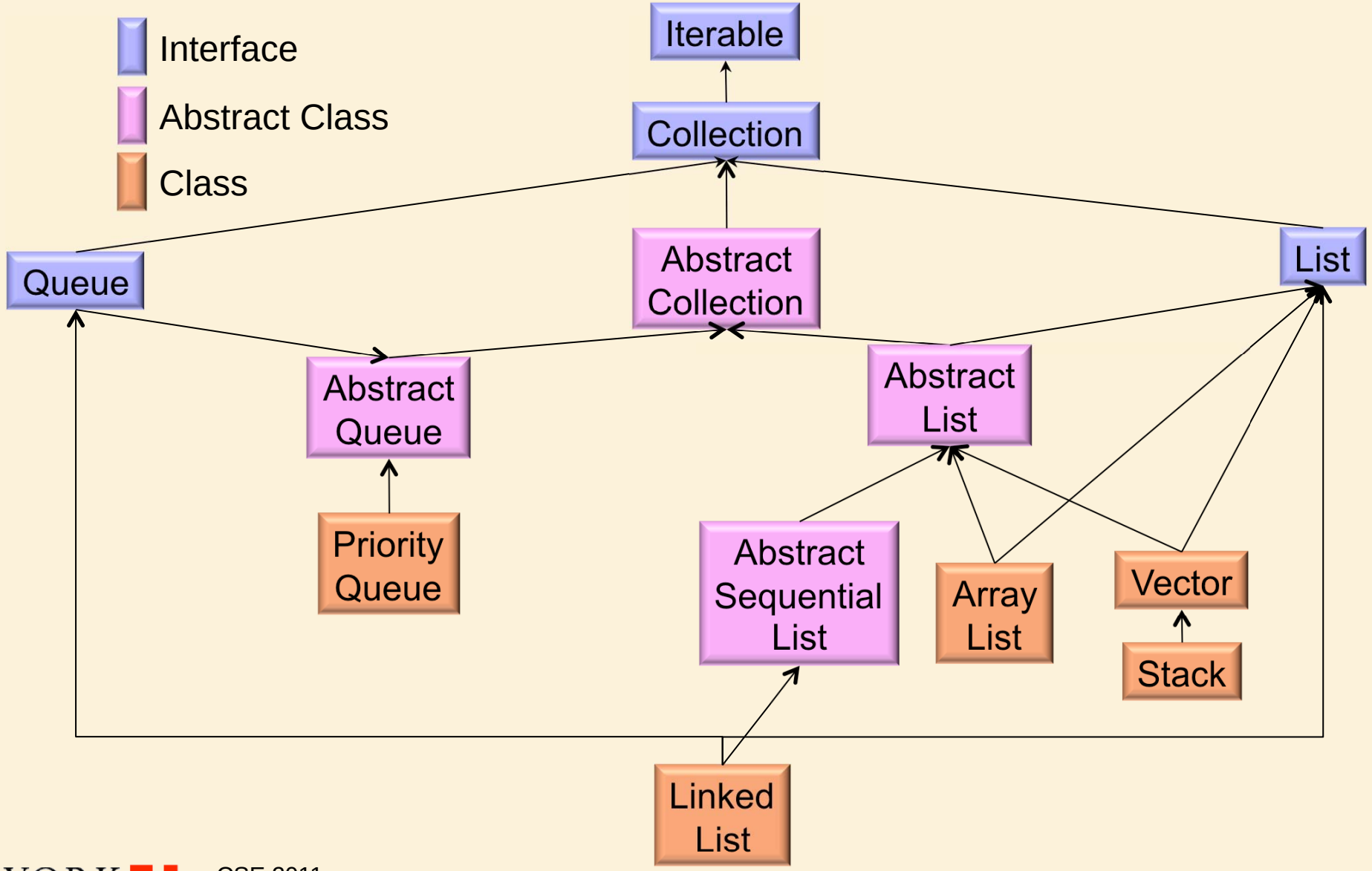
# Lecture 3
# Linear Data Structures:
# Arrays, Array Lists, Stacks, Queues and Linked Lists

Chapters 3.1-3.3 , 5.1-5.2, 6.1

# Core Collection Interfaces



# The Java Collections Framework



# Arrays

Chapter 3.1

# Arrays

- Array: a sequence of indexed components with the following properties:
  - **array size is fixed** at the time of array's construction
    - **int**[] numbers = **new int** [10];
  - **array elements are placed contiguously** in memory
    - address of any element can be calculated directly as its offset from the beginning of the array
  - consequently, array components **can be efficiently inspected or updated** in O(1) time, using their indices
    - **randomNumber = numbers[5];**
    - **numbers[2] = 100;**

# Arrays in Java (java.util.Arrays)

- For an array of length **n**, the index bounds are **0** to **n-1**.
- Java arrays are homogeneous
  - all array components must be of the same (object or primitive) type.
  - but, an array of an object type can contain objects of any respective subtype
- An array is itself an object.
  - it is allocated dynamically by means of **new**
  - it is automatically deallocated when no longer referred to
- When an array is first created, all values are automatically initialized with
  - 0, for an array of int[] or double[] type
  - false, for a boolean[] array
  - null, for an array of objects
- Example [ common error –uninitialized arrays]

```
int[] numbers;
numbers[2] = 100;
```

# Arrays in Java

- The length of any array object can be accessed through its instance variable '**length**'.
  - the cells of an array **A** are numbered: **0, 1, .., A.length-1**
- ArrayIndexOutOfBoundsException
  - thrown at an attempt to index into array **A** using a number larger than **A.length-1**.
  - helps Java avoid 'buffer overflow attacks'
- Example [ declaring, defining and determining the size of an array]

```
int[] A={12, 24, 37, 53, 67};
for (int i=0; i < A.length; i++) {
...}
```

# Buffer Overflows

**Windows Buffer Overflow Protection Programs: Not Much**

*<"Paul Robinson" <postmaster@paul.washington.dc.us>>*

*Tue, 10 Aug 2004 15:26:44 GMT* An 9 Aug 2004

…**there is a bug in AOL Instant Messenger allowing an attacker to send a message that can cause a buffer overflow and possibly execute code on the attacked machine**. Apparently this will only occur if the attacker sends a url - like the one in this message - as a hyperlink and the victim clicks on it, which makes the probability of attack much lower than a "standard buffer overflow attack" upon a program.

*Mon, 09 Aug 2004 17:24:19 GMT*

…a **buffer overflow exploit** is one in which someone sends too much data to a program (such as a web server application), sending far more data than the program would expect, in order to force arbitrary data into a storage area (a "buffer") so the amount of data forced into the buffer goes beyond the expected limits, causing the data to overflow the buffer and makes it possible for that data to be executed as arbitrary program code. Since the attacker forces code of his choosing into the execution stream, he now 0wns your box, because as the saying goes, if I can run code on your machine - especially if it's a Windows machine where there is not much protection - I can pretty much do anything I please there.

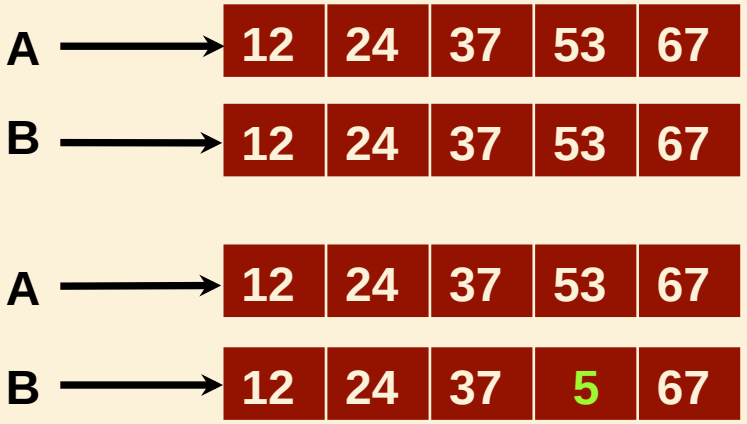
# Arrays in Java

- Since an array is an object, the name of the array is actually a **reference** (pointer) to the place in memory where the array is stored.
  - reference to an object holds the **address** of the actual object
- Example [ arrays as objects]

```
int[] A={12, 24, 37, 53, 67};
int[] B=A;
B[3]=5;
```


- Example [ cloning an array]

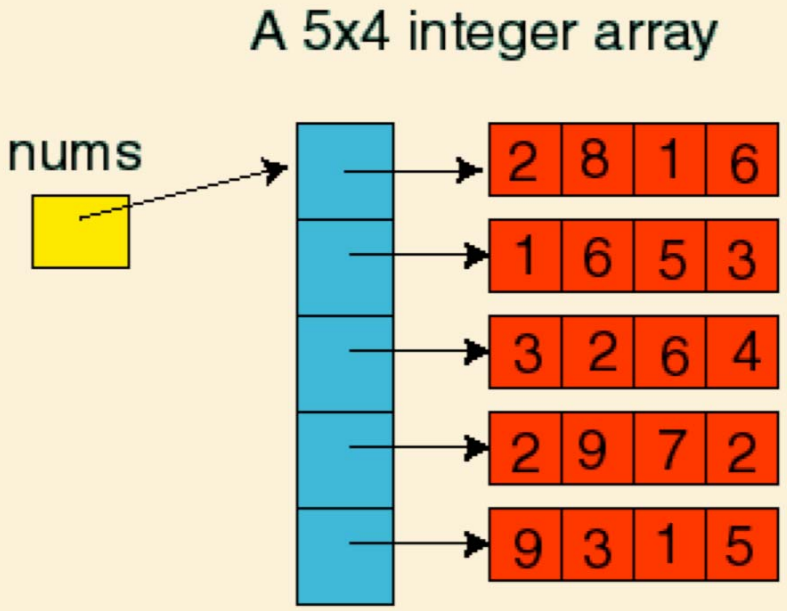
```
int[] A={12, 24, 37, 53, 67};
int[] B=A.clone();
B[3]=5;
```



# Example

Example [ 2D array in Java = array of arrays]

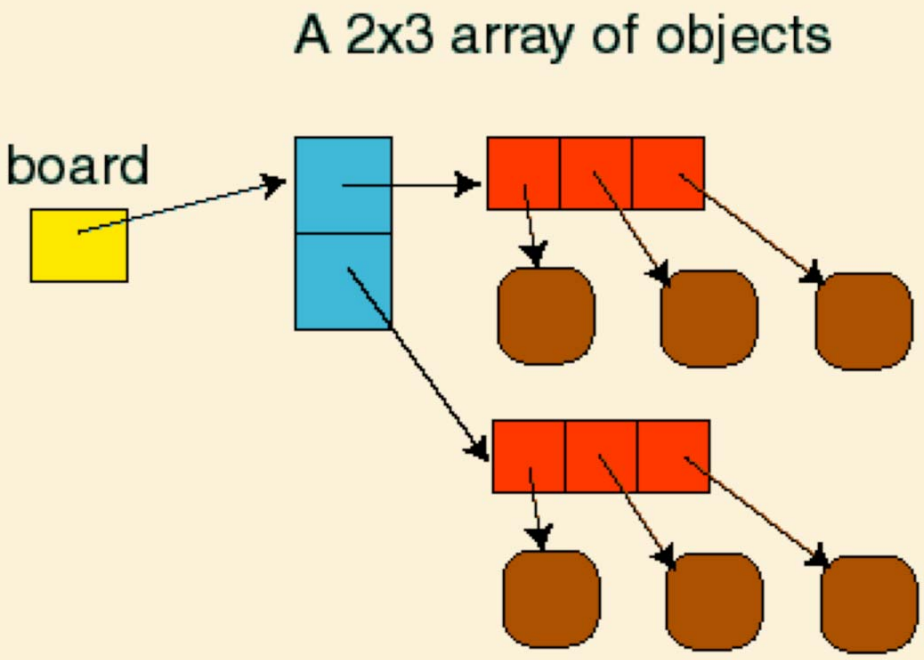
```
int[][] nums = new int[5][4];

int[][] nums;

nums = new int[5][];

for (int i=0; i<5; i++) {

    nums[i] = new int[4];

}
```



# Example

Example [ 2D array of objects in Java = an array of arrays of references]

Square[][] board = **new** Square[2][3];



# The **Java.util.Arrays** Class

- Useful Built-In Methods in **Java.util.Arrays**
  - **equals**(A,B)
    - returns true if A and B have an equal number of elements and every corresponding pair of elements in the two arrays are equal
  - **fill**(A,x)
    - store element x into every cell of array A
  - **sort**(A)
    - sort the array A in the natural ordering of its elements
  - **binarySearch**([int] A, int key)
    - search the specified array of ints for the specified value using the binary search algorithm

# Example

## What is printed?

```
int[] A={12, 24, 37, 53, 67};
int[] B=A.clone();
if (A==B)  System.out.println(“ Superman ”);
if (A.equals(B))  System.out.println(“ Snow White ”);
```

# Limitations of Arrays

- Static data structure
  - size must be fixed at the time the program creates the array
  - once set, array size cannot be changed
  - if number of entered items > declared array size ⇒ out of memory
    - fix 1: use array size > number of expected items ⇒ waste of memory
    - fix 2: increase array size to fit the number of items ⇒ extra time
- Insertion / deletion in an array is time consuming
  - all the elements following the inserted element must be shifted appropriately
- Example [ time complexity of “growing”an array]

```
if (numberOfItems > numbers.length) {
    int[] newNumbers = new int[2*numbers.length];
    System.arraycopy(numbers, 0, newNumbers, 0, numbers.length);
    numbers = newNumbers;
}
```

# Array Lists

Chapter 6.1

# The Array List ADT (§6.1)

- The Array List ADT extends the notion of array by storing a sequence of arbitrary objects
- An element can be accessed, inserted or removed by specifying its rank (number of elements preceding it)
- An exception is thrown if an incorrect rank is specified (e.g., a negative rank)

# The Array List ADT

```
public interface IndexList<E> {
/** Returns the number of elements in this list */
public int size();
/** Returns whether the list is empty. */
public boolean isEmpty();
/** Inserts an element e to be at index I, shifting all elements after this. */
public void add(int I, E e) throws IndexOutOfBoundsException;
/** Returns the element at index I, without removing it. */
public E get(int i) throws IndexOutOfBoundsException;
/** Removes and returns the element at index I, shifting the elements after this. */
public E remove(int i) throws IndexOutOfBoundsException;
/** Replaces the element at index I with e, returning the previous element at i. */
public E set(int I, E e) throws IndexOutOfBoundsException;
}
```

# Applications of Array Lists

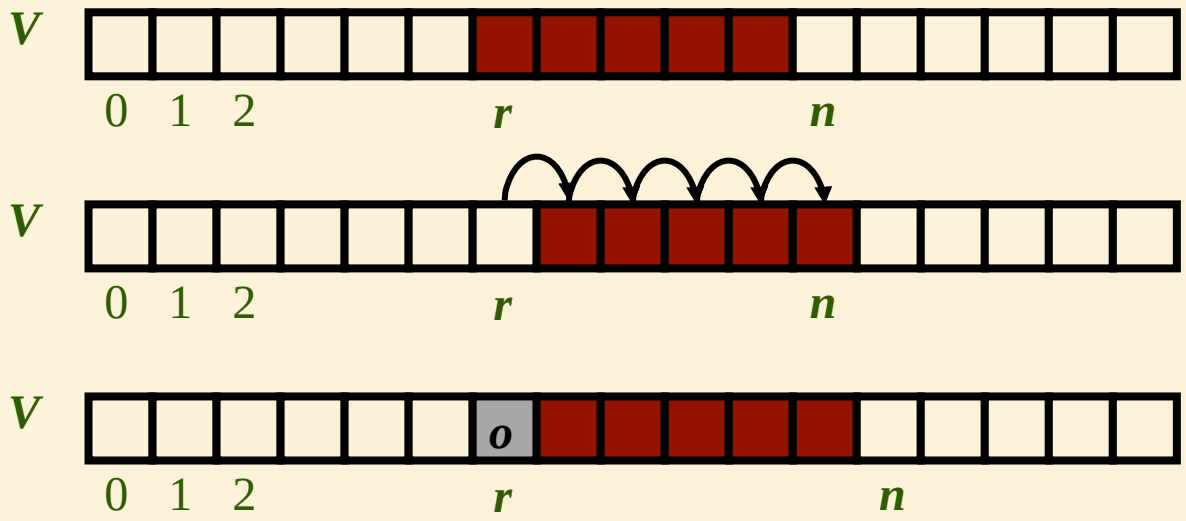
- Direct applications
  - Sorted collection of objects (elementary database)
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

# A Simple Array-based Implementation

- Use an array $V$ of size $N$
- A variable $n$ keeps track of the size of the array list (number of elements stored)
- Operation ***get***($r$) is implemented in $O(1)$ time by returning $V[r]$

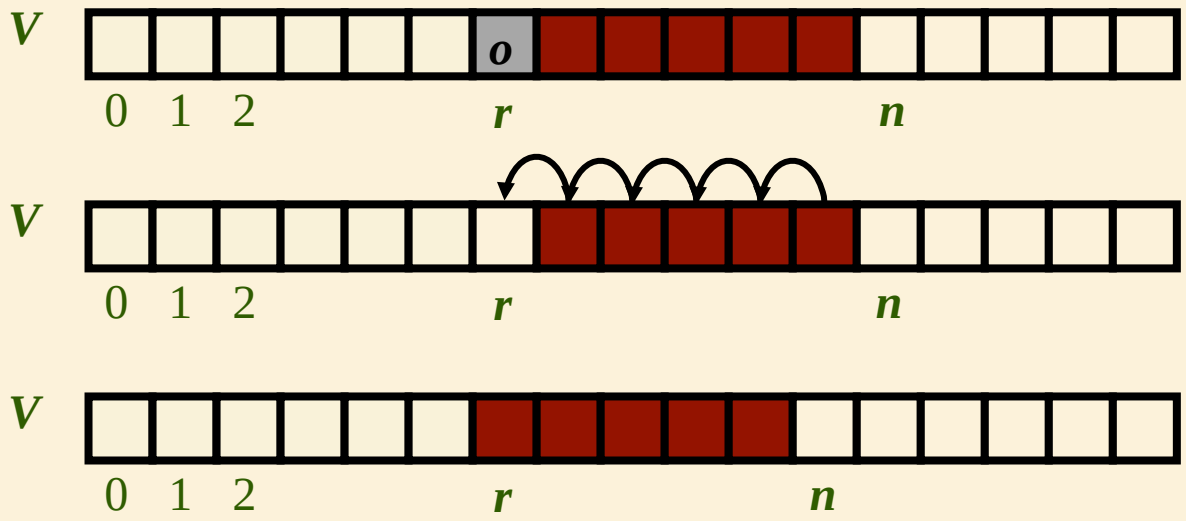
# Insertion

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



# Deletion

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



# Performance

- In the array based implementation
  - The space used by the data structure is $O(n)$
  - ***size***, ***isEmpty***, ***get*** and ***set*** run in $O(1)$ time
  - ***add*** and ***remove*** run in $O(n)$ time
- In an ***add*** operation, when the array is full, instead of throwing an exception, we could replace the array with a larger one.
- In fact **java.util.ArrayList** implements this ADT using **extendable arrays** that do just this.

# Implementing Array Lists using Extendable Arrays

- In an **add** operation, when the array is full, instead of throwing an exception, we can 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 size(S) = N
    then
        A ← new array of size ???
        for i ← 0 to N-1 do
            A[i] ← S[i]
            S ← A
    size(S) ← size(S) + 1
    S[N] ← o
```

# Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of $n$ **add** operations
- We simplify the analysis by assuming **add(o)** operations that append the object to the end of the list.
- We assume that we start with an empty array list represented by an array of size $1$
- The amortized time of an **add(o)** operation is the average time taken 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(o)** operations is proportional to

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

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

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

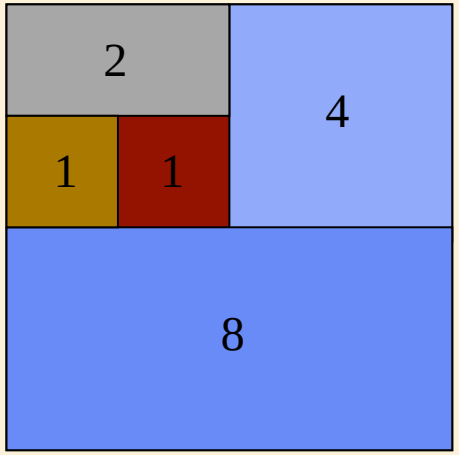
- Since $c$ is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an **add(o)** operation is $O(n)$

# Doubling Strategy Analysis

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

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

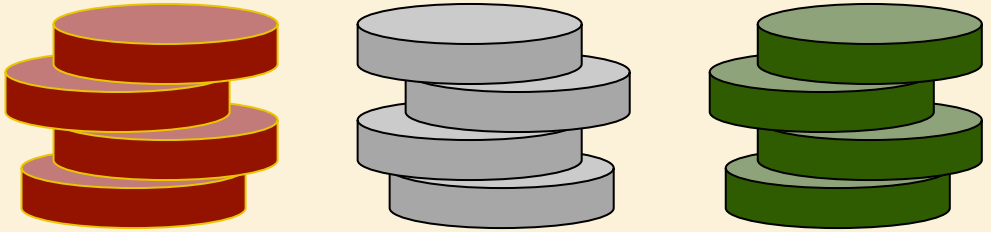
- Thus $T(n)$ is $O(n)$
- **The amortized time of an add operation is $O(1)$!**



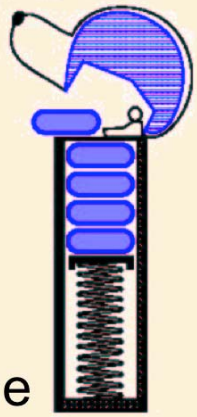

$$\left( \text{Recall: } \sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r} \right)$$

# Stacks

Chapter 5.1



# The Stack ADT



- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored

# Stack Interface in Java

- Example java interface

```
public interface Stack {

    public int size();

    public boolean isEmpty();

    public Object top()
            throws EmptyStackException;

    public void push(Object o);

    public Object pop()
            throws EmptyStackException;
}
```

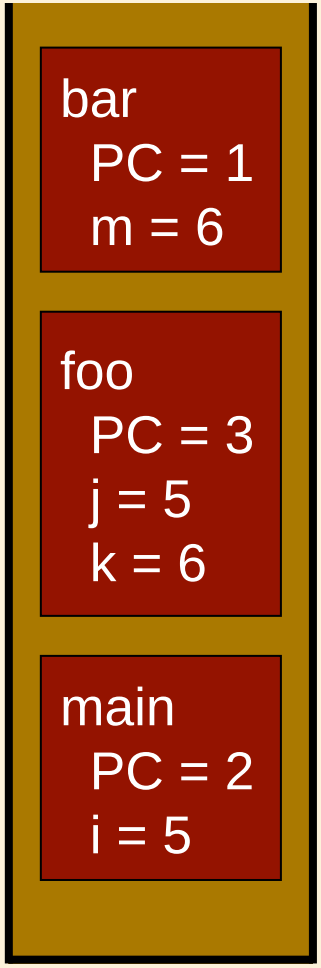
# Applications of Stacks

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Chain of method calls in the Java Virtual Machine

# Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack
- Allows for **recursion**

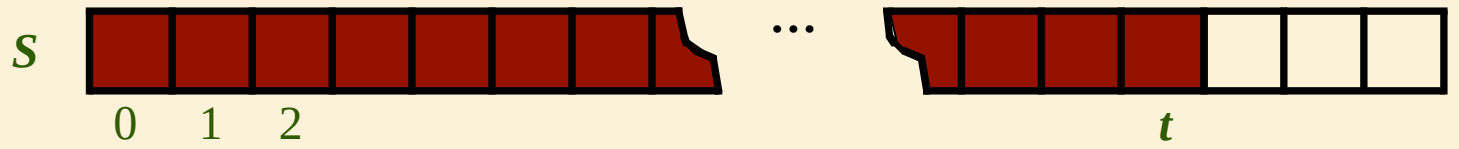
```
main() {
    int i = 5;
    foo(i);
    }

foo(int j) {
    int k;
    k = j+1;
    bar(k);
    }

bar(int m) {
    …
    }
```



# Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

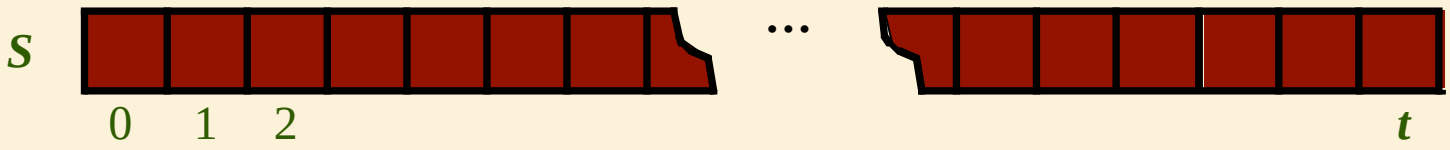
```
Algorithm size()
    return t + 1

Algorithm pop()
    if isEmpty() then
        throw EmptyStackException
    else
        t ← t - 1
        return S[t + 1]
```



# Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

```
Algorithm push(o)
    if t = S.length - 1 then
        throw FullStackException
    else
        t ← t + 1
        S[t] ← o
```



# Performance and Limitations

- Performance
  - Let $n$ be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

# Example array-based stack in Java

```
public class ArrayStack
      implements Stack {

   // holds the stack elements
   private Object S[ ];

   // index to top element
   private int top = -1;

   // constructor
   public ArrayStack(int capacity) {
       S = new Object[capacity]);
    }
```

```
   public Object pop()
         throws EmptyStackException {
      if isEmpty()
         throw new EmptyStackException
            ("Empty stack: cannot pop");
      Object temp = S[top];
      // facilitates garbage collection
      S[top] = null;
      top = top – 1;
      return temp;
   }
```

# Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect: )(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

# Parentheses Matching Algorithm

**Algorithm** ParenMatch($X$,$n$):

***Input:*** An array $X$ of $n$ tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

***Output:*** **true** if and only if all the grouping symbols in $X$ match

Let $S$ be an empty stack

**for** $i$=0 to $n$-1 **do**

    **if** $X[i]$ is an opening grouping symbol **then**

        $S$.push($X[i]$)

    **else if** $X[i]$ is a closing grouping symbol **then**

        **if** $S$.isEmpty() **then**

            **return false** *{nothing to match with}*

        **if** $S$.pop() does not match the type of $X[i]$ **then**

            **return false** *{wrong type}*

**if** $S$.isEmpty() **then**

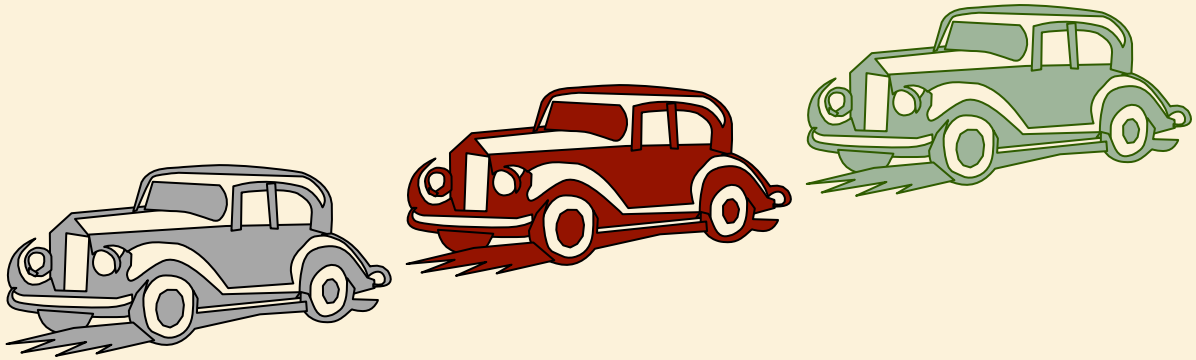
    **return true** *{every symbol matched}*

**else**

    **return false** *{some symbols were never matched}*

# Queues

Chapters 5.2-5.3



# The Queue ADT

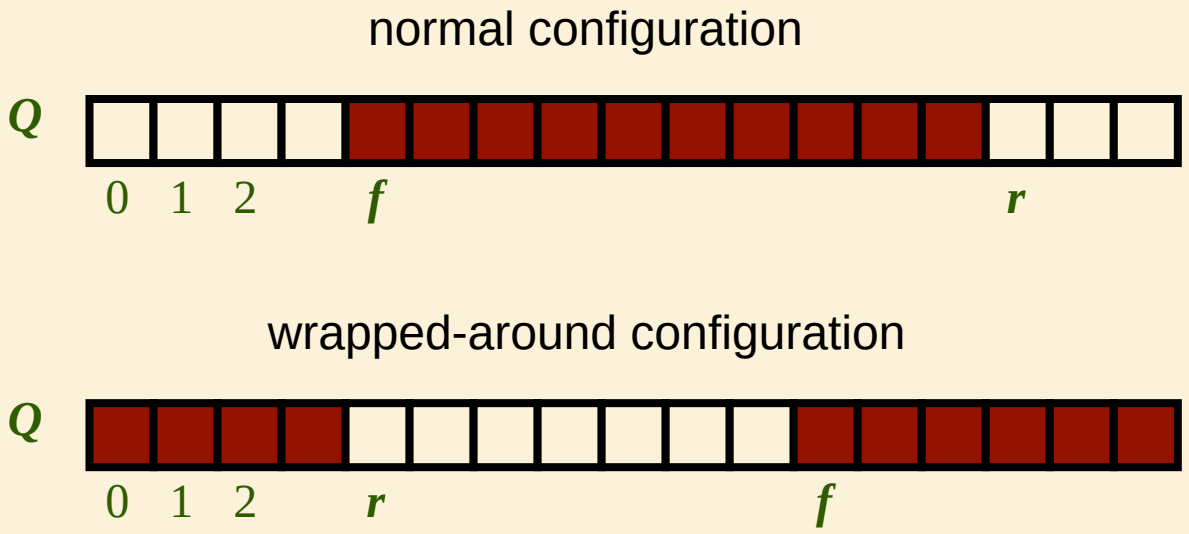
- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object front(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException
  - Attempting to enqueue an element on a queue that is full ***can*** be signaled with a FullQueueException.

# Queue Example

| *Operation* | *Output* | *Q* |
|---|---|---|
| enqueue(5) | – | (5) |
| enqueue(3) | – | (5, 3) |
| dequeue() | *5* | (3) |
| enqueue(7) | – | (3, 7) |
| dequeue() | *3* | (7) |
| front() | *7* | (7) |
| dequeue() | *7* | () |
| dequeue() | *"error"* | () |
| isEmpty() | *true* | () |
| enqueue(9) | – | (9) |
| enqueue(7) | – | (9, 7) |
| size() | *2* | (9, 7) |
| enqueue(3) | – | (9, 7, 3) |
| enqueue(5) | – | (9, 7, 3, 5) |
| dequeue() | *9* | (7, 3, 5) |

# Array-Based Queue

- Use an array of size $N$ in a circular fashion
- Two variables keep track of the front and rear
  - $f$ index of the front element
  - $r$ index immediately past the rear element
- Array location $r$ is kept empty



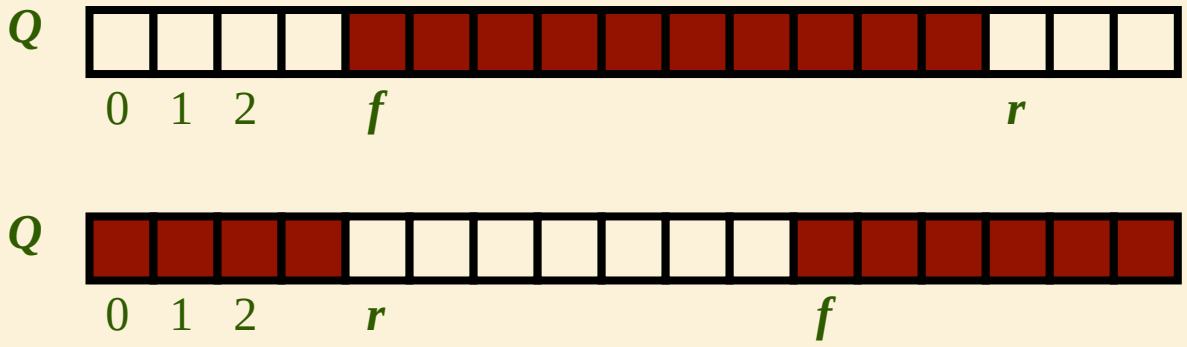
# Queue Operations

- We use the modulo operator (remainder of division)

**Algorithm** ***size***()
  **return** $(N - f + r) \bmod N$

**Algorithm** ***isEmpty***()
  **return** $(f = r)$

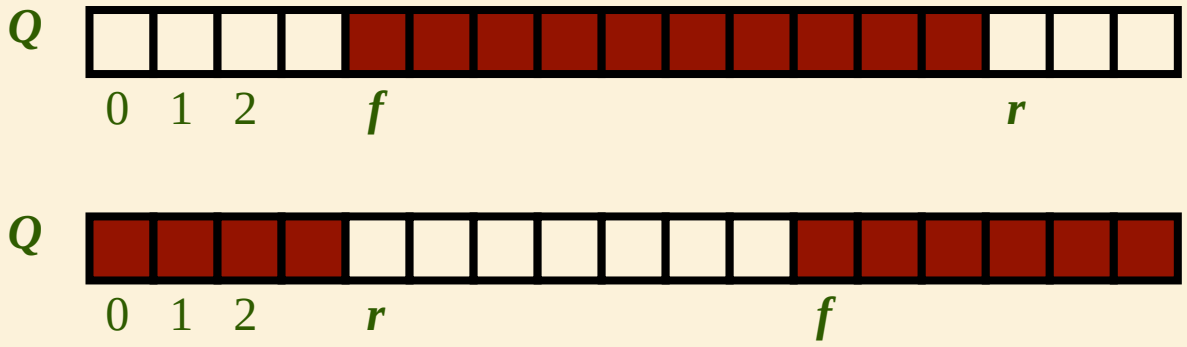
Note: $N - f + r = (r + N) - f$



# Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full

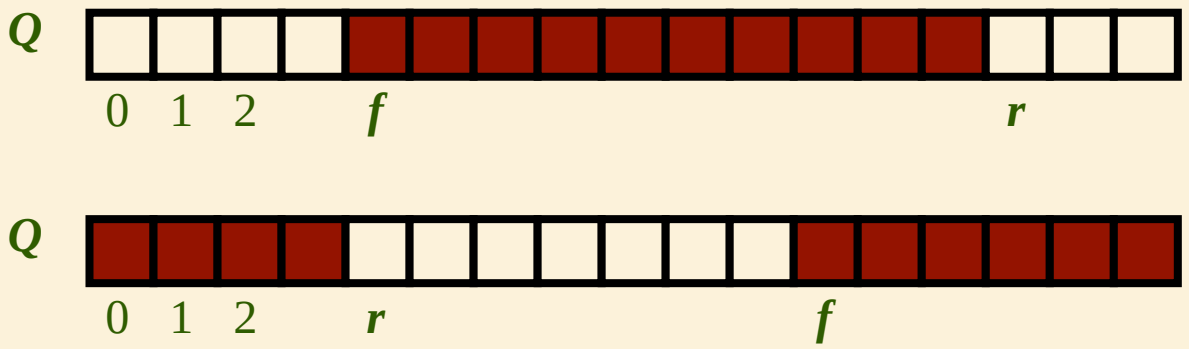
```
Algorithm enqueue(o)
    if size() = N - 1 then
        throw FullQueueException
    else
        Q[r] ← o
        r ← (r + 1) mod N
```



# Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty

```
Algorithm dequeue()
    if isEmpty() then
        throw EmptyQueueException
    else
        o ← Q[f]
        f ← (f + 1) mod N
        return o
```



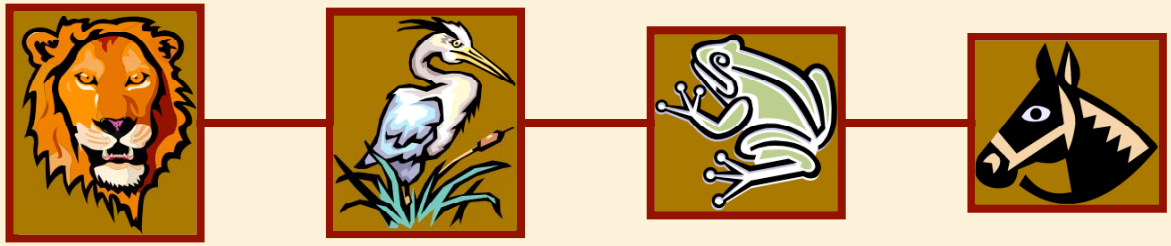
# Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Requires the definition of class EmptyQueueException
- No corresponding built-in Java class

```
public interface Queue {

    public int size();

    public boolean isEmpty();

    public Object front()
            throws EmptyQueueException;

    public void enqueue(Object o);

    public Object dequeue()
            throws EmptyQueueException;
}
```

# Linked Lists

Chapters 3.2 – 3.3

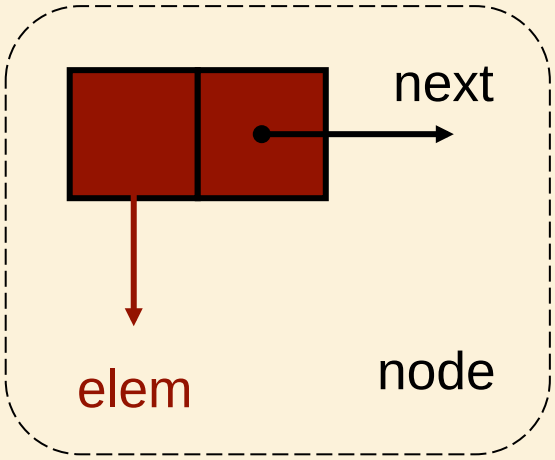


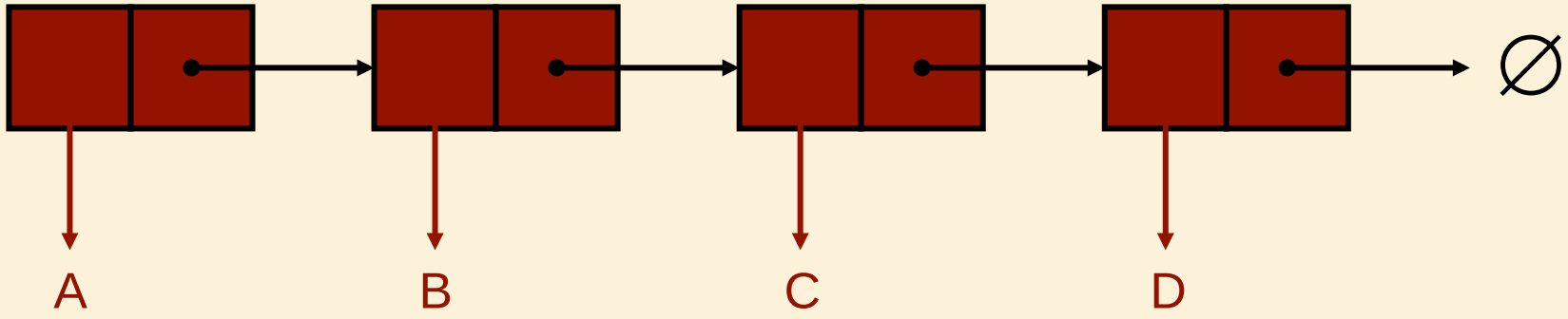
# Linked Lists

- By virture of their random access nature, **arrays** support non-structural read/write operations (e.g., **get(i)**, **set(i)**) in **O(1)** time.
- Unfortunately, structural operations (e.g., **add(i,e)** **remove(i)**) take **O(n)** time.
- For some algorithms, structural operations may dominate the running time.
- For such cases, **linked lists** may be more appropriate.

# Singly Linked List (§ 3.2)

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



# Example Java Node Class for List Nodes

```
public class Node {
   // Instance variables:
   private  Object element;
   private  Node next;
   /** Creates a node with null references to
     its element and next node. */
   public  Node()
 {
      this(null,  null);
  }
/** Creates a node with the given element
     and next node. */
   public  Node(Object e,  Node n)
 {
       element  =  e;
       next  =  n;
  }
```

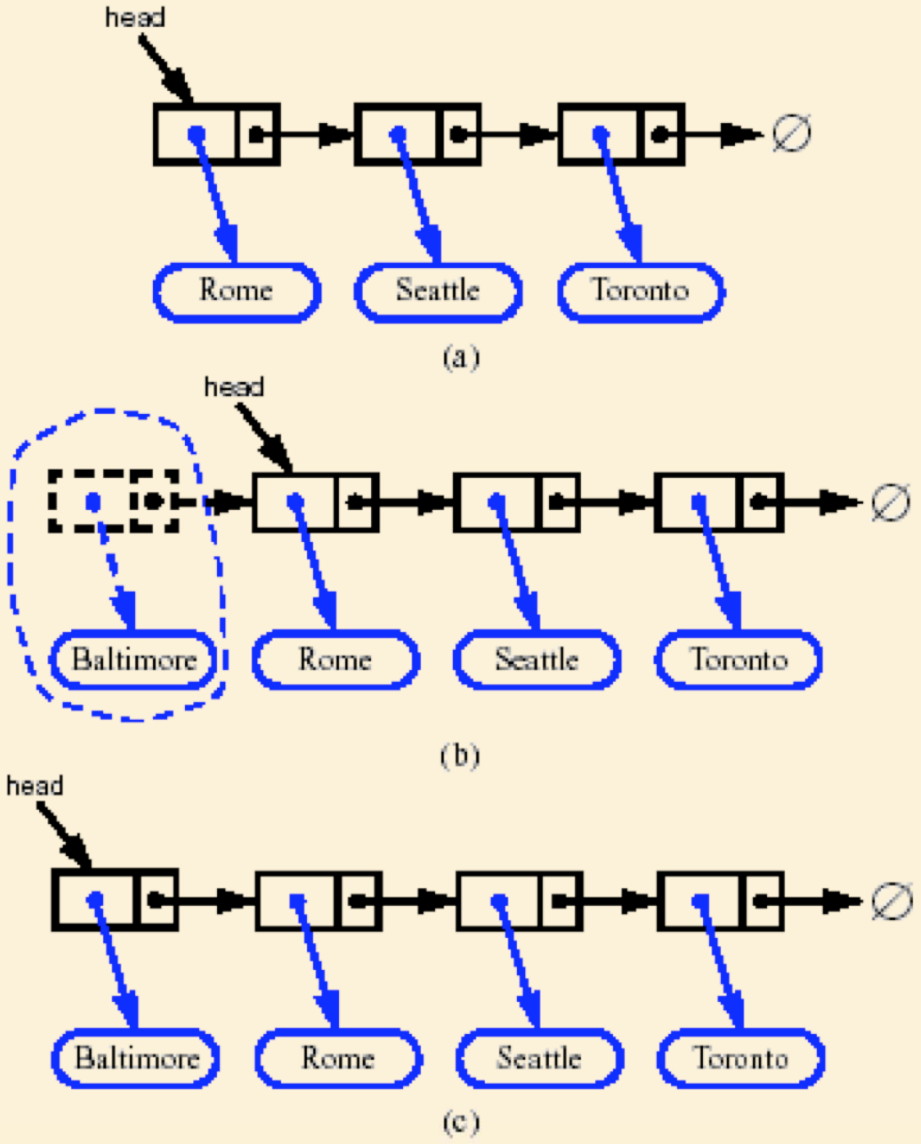
```
// Accessor methods:
   public  Object getElement()
 {
      return  element;
  }
   public  Node getNext()
 {
      return  next;
  }


// Modifier methods:
   public void  setElement(Object newElem)
{
       element  =  newElem;
  }
public void  setNext(Node newNext)
 {
       next  =  newNext;
 }
```

# Example Java Class for Singly-Linked List

```
public class SLinkedList {
    // Instance variables:
    protected Node head; //head node of list
    protected Node tail; //tail node of list
    protected long size; //number of nodes in list
    /** Default constructor that creates an empty list. */
    public  SLinkedList()
  {
      head = null;
      tail = null;
      size = 0;
   }
// update and search methods go here…
}
```

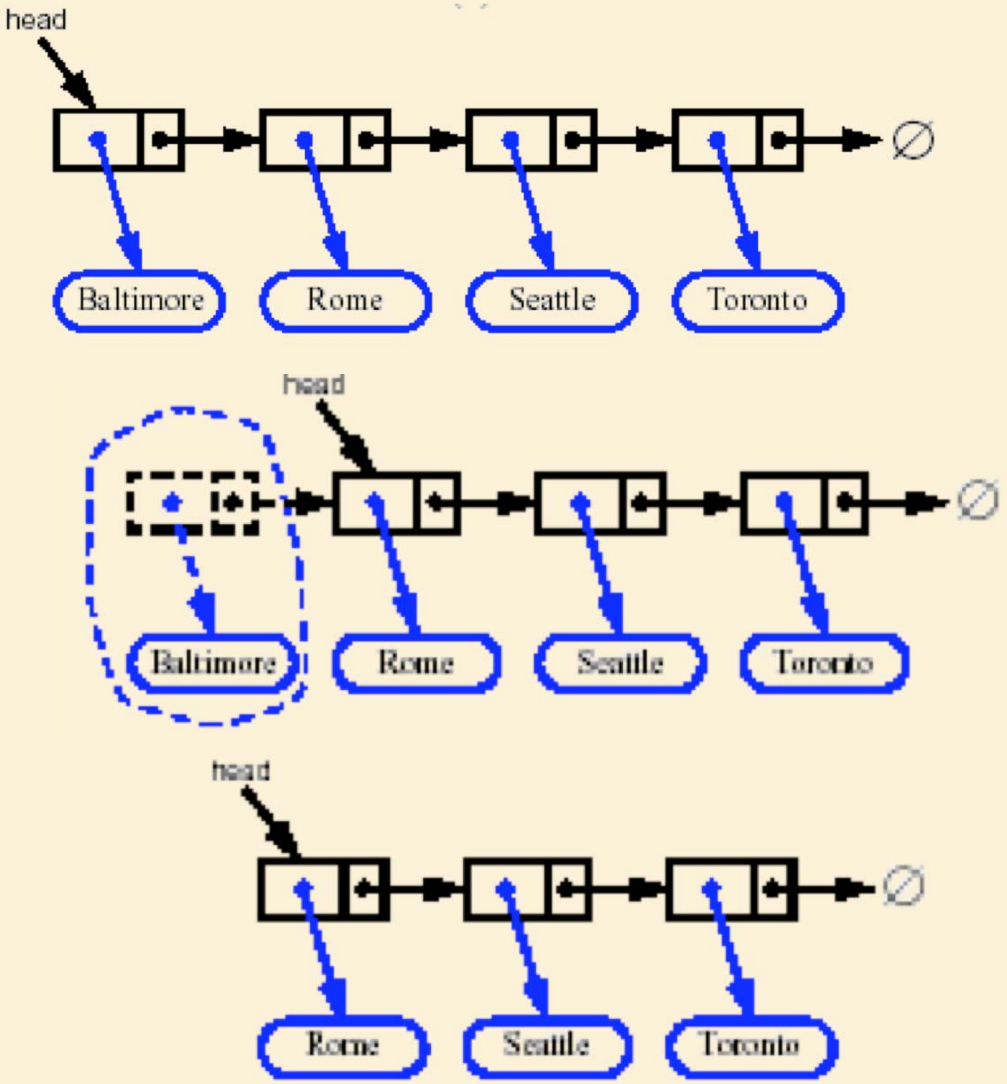
# Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old **head**
4. Update **head** to point to new node
5. If list was initially empty, have to update **tail** as well.



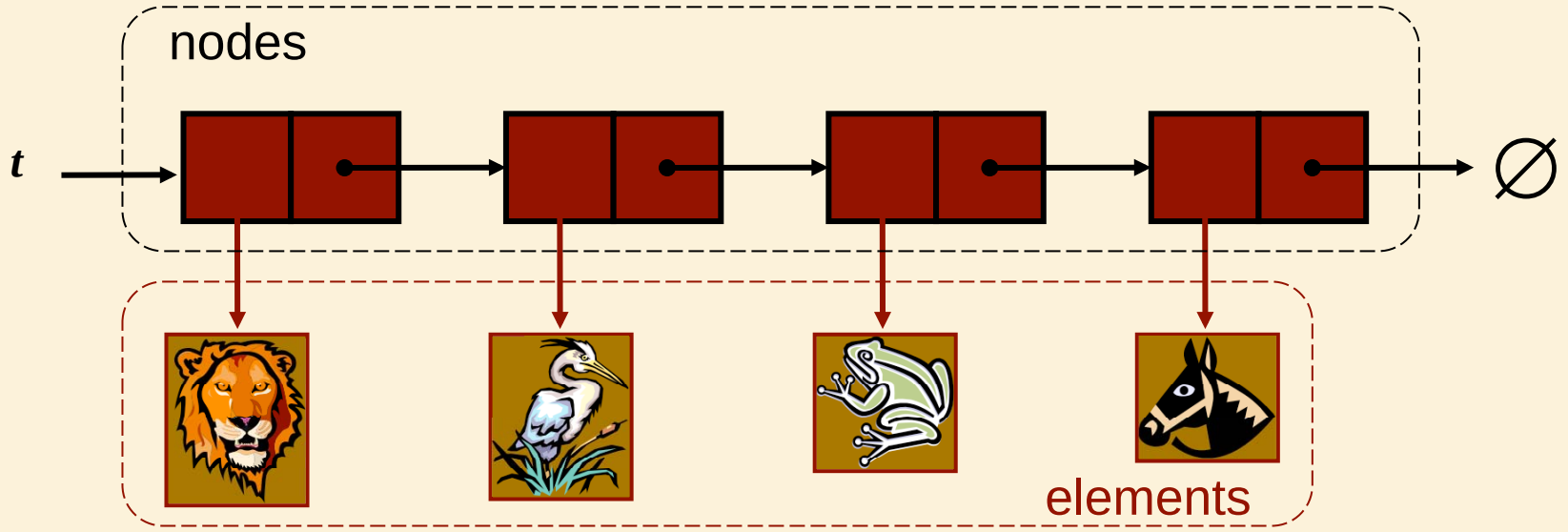
# Removing at the Head

1. Update **head** to point to next node in the list
2. Allow garbage collector to reclaim the former first node
3. If list is now empty, have to update **tail** as well.



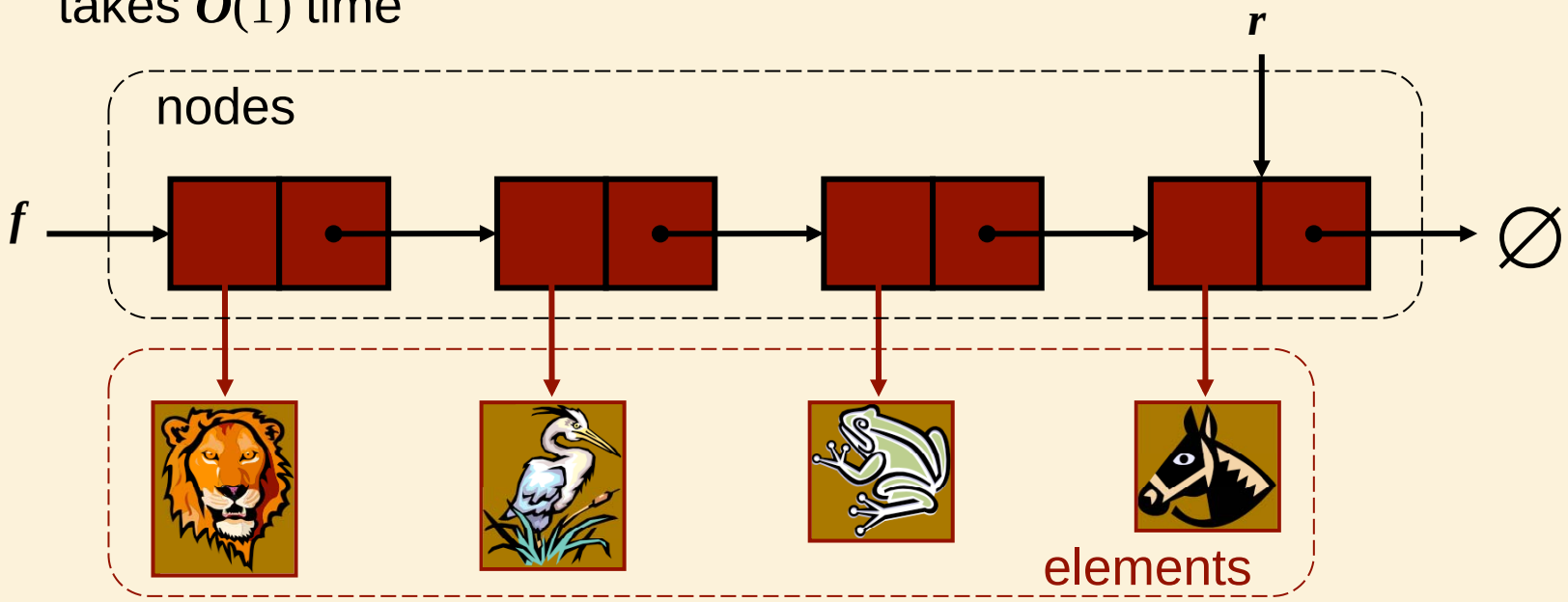
# Implementing a Stack with a Singly-Linked List

- Earlier we saw an array implementation of a stack.
- We could also implement a stack with a singly-linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



# Implementing a Queue with a Singly-Linked List

- Just as for stacks, queue implementations can be based upon either arrays or linked lists.
- In a linked list implementation:
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time

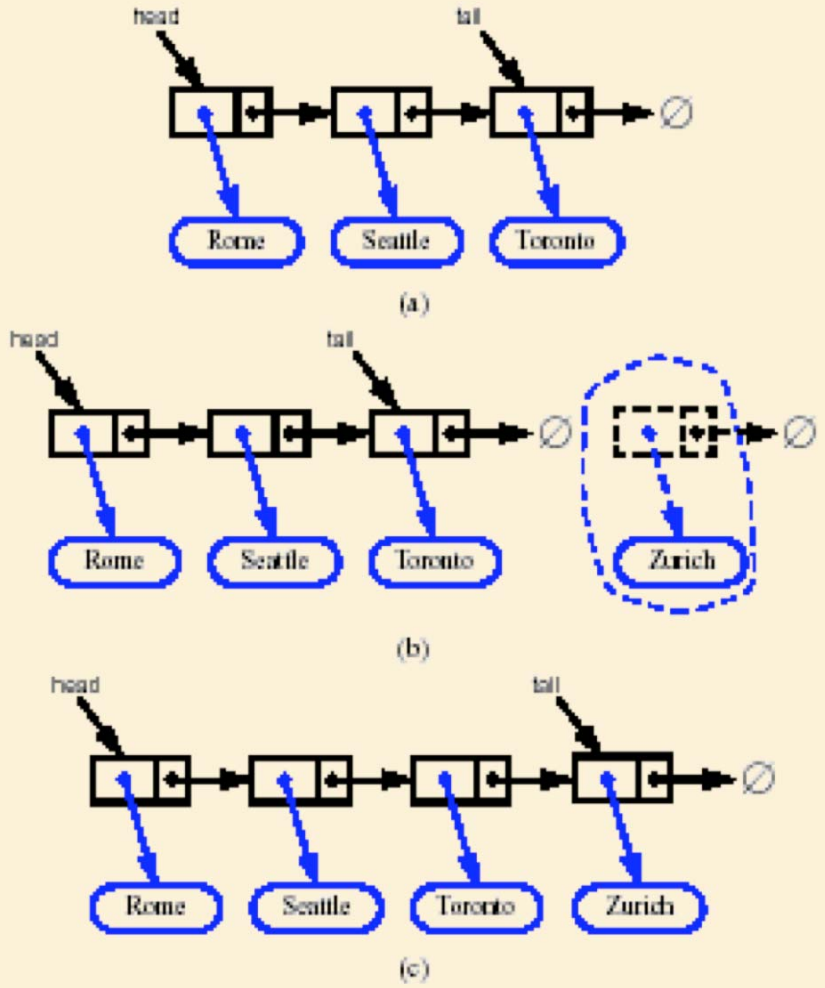


# Running Time

- Adding at the head is O(1)
- Removing at the head is O(1)
- **How about tail operations?**

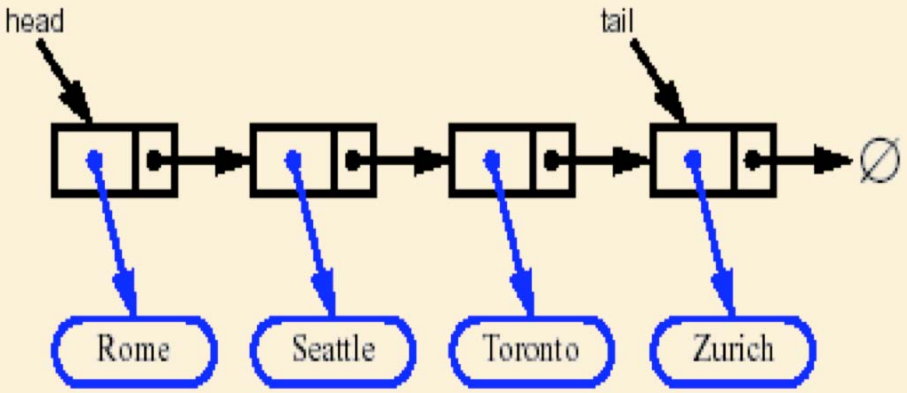
# Inserting at the Tail

1. Allocate a new node
2. Update new element
3. Have new node point to null
4. Have old last node point to new node
5. Update **tail** to point to new node
6. If list initially empty, have to update **head** as well.



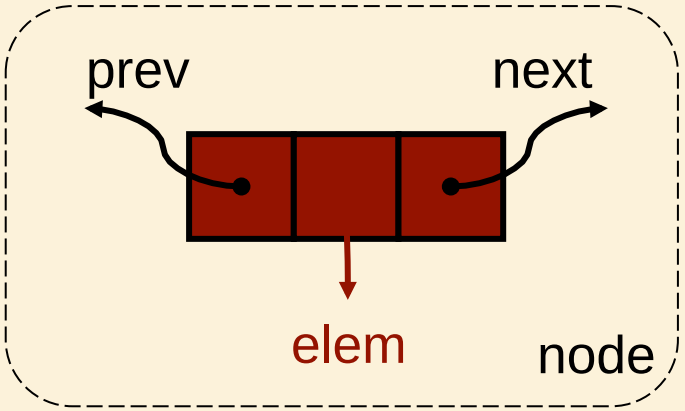
# Removing at the Tail

- 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

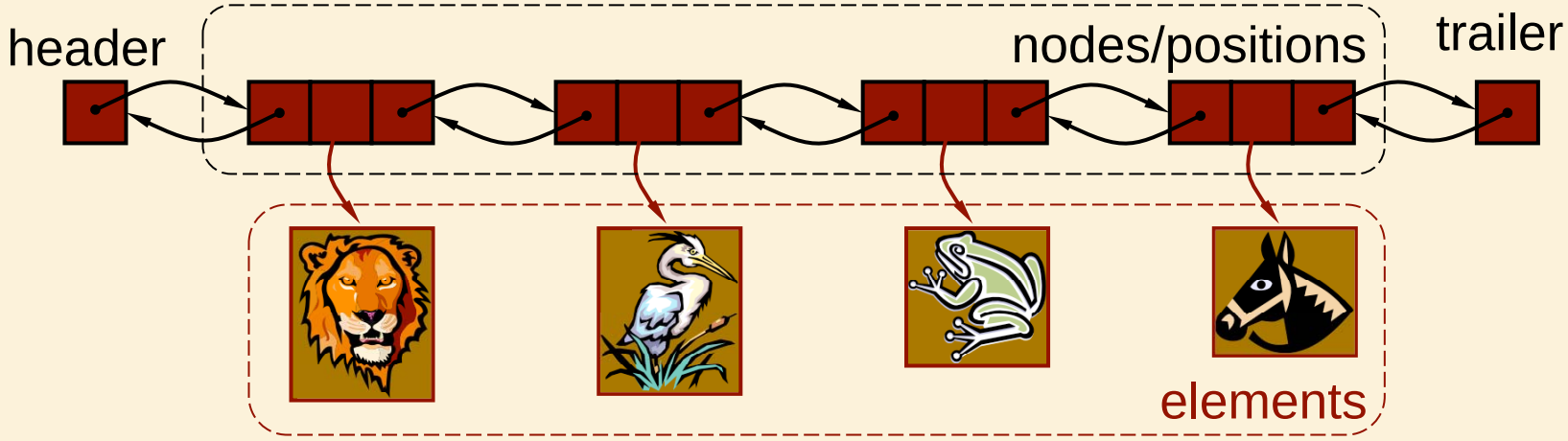


# Doubly Linked List

- Doubly-linked lists allow more flexible list management (constant time operations at both ends).
- Nodes store:
  - element
  - link to the previous node
  - link to the next node

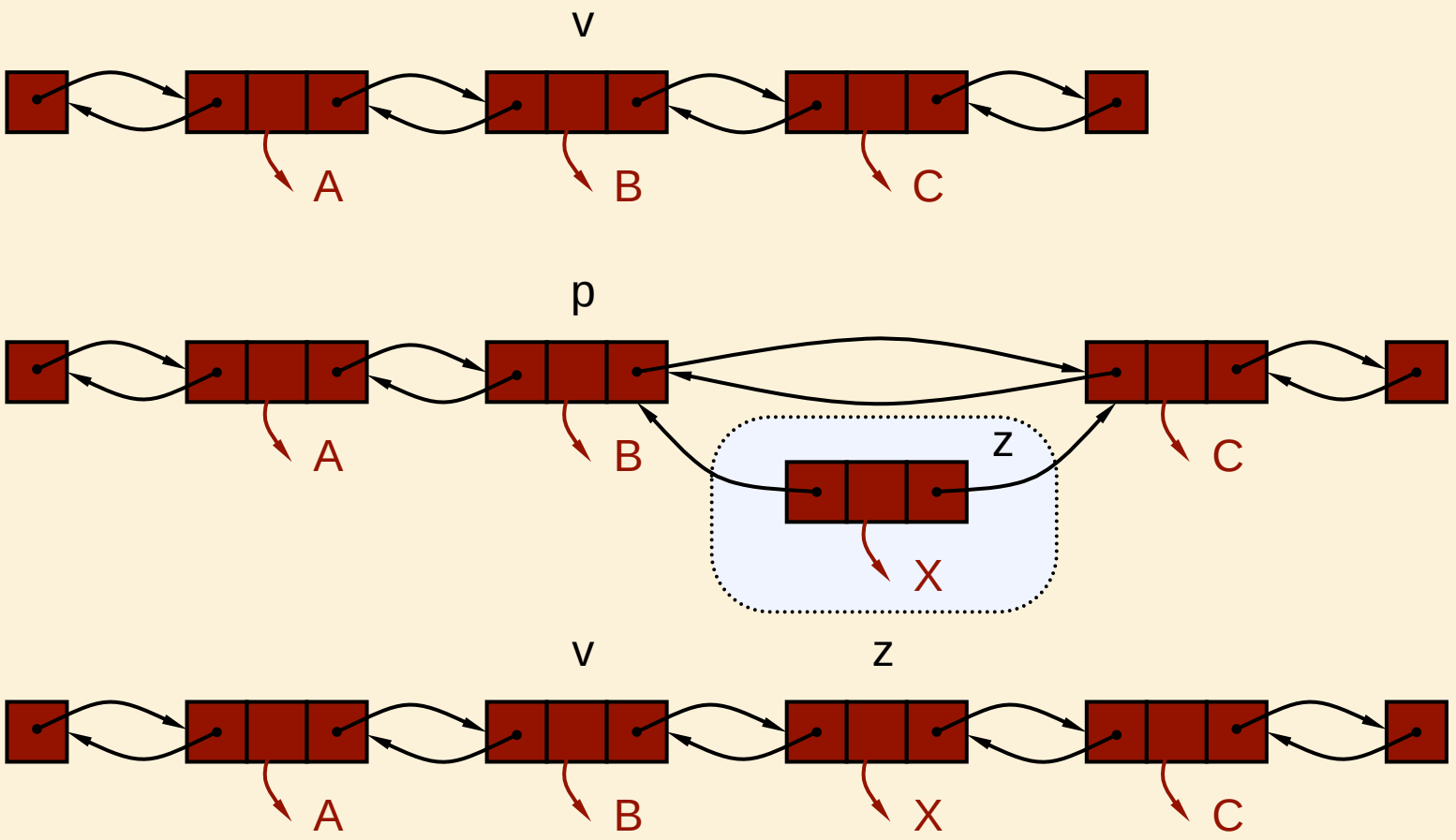



- Special trailer and header (sentinel) nodes



# Insertion

- addAfter(v, z) inserts node z after node v in the list



# Insertion Algorithm

**Algorithm** addAfter(*v*, *z*):

w ←*v.getNext()*

*z*.setPrev(*v*) {link *z* to its predecessor}

*z*.setNext(*v*.getNext()) {link *z* to its successor}

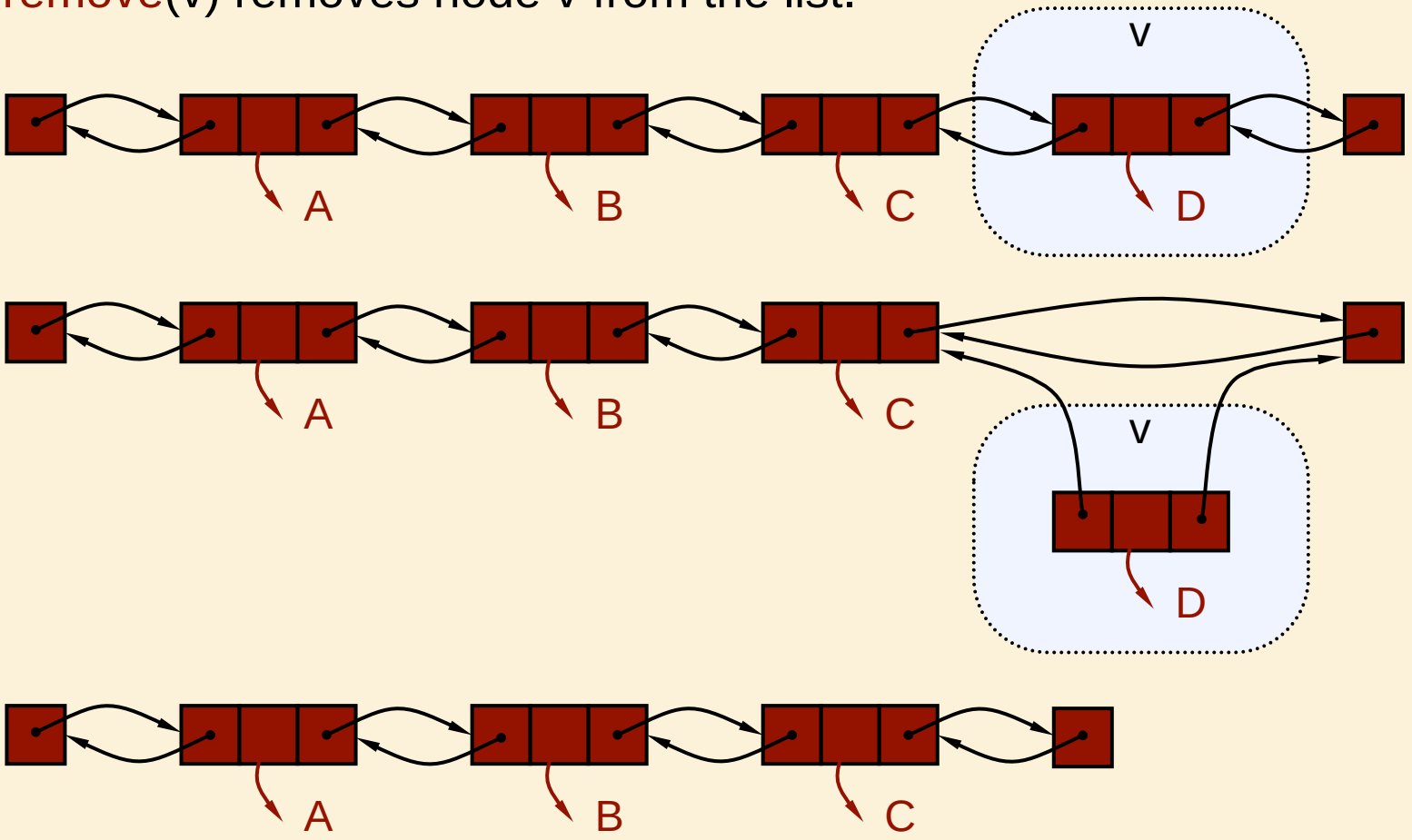
(*v*.getNext()).setPrev(*z*) {link z's successor back to z}

*v*.setNext(*z*) {link v to its new successor, z}

size ←size + 1

# Deletion

- remove(v) removes node v from the list.



# Deletion Algorithm

**Algorithm** remove(*v*):

*u* ← *v.getPrev()*  {node before v}

*w* ← *v.getNext()*  {node after v}

*w.*setPrev(**u**)  {link out v}

*u.*setNext(**w**)

*v.*setPrev(**null**)  {null out fields of v}

*v.*setNext(**null**)

size ← size - 1

# Running Time

- Insertion and Deletion of any given node takes O(1) time.
- However, depending upon the application, finding the insertion location or the node to delete may take longer!