

# Lecture 3

## Linear Data Structures

# Learning Outcomes

- Based on this lecture, you should:
  - Know the basic linear data structures
  - Be able to express each as an Abstract Data Type (ADT)
  - Be able to specify each of these ADTs as a Java interface.
  - Be able to outline the algorithms for creating, accessing and modifying each data structure
  - Be able to analyze the running time of these operations
  - Be able to identify particular applications for which each data structure would be suited.

# Linear Data Structures

- Arrays (Ch. 3.1)
- Array Lists (Ch. 7.2)
- Stacks (Ch. 6.1)
- Queues (Ch. 6.2)
- Linked Lists (Ch. 3.2 – 3.4)

# Linear Data Structures

- **Arrays (Ch. 3.1)**
- Array Lists (Ch. 7.2)
- Stacks (Ch. 6.1)
- Queues (Ch. 6.2)
- Linked Lists (Ch. 3.2 – 3.4)



# 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

- 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 –unallocated 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 owns 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 [ copying array references]

```
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;
```



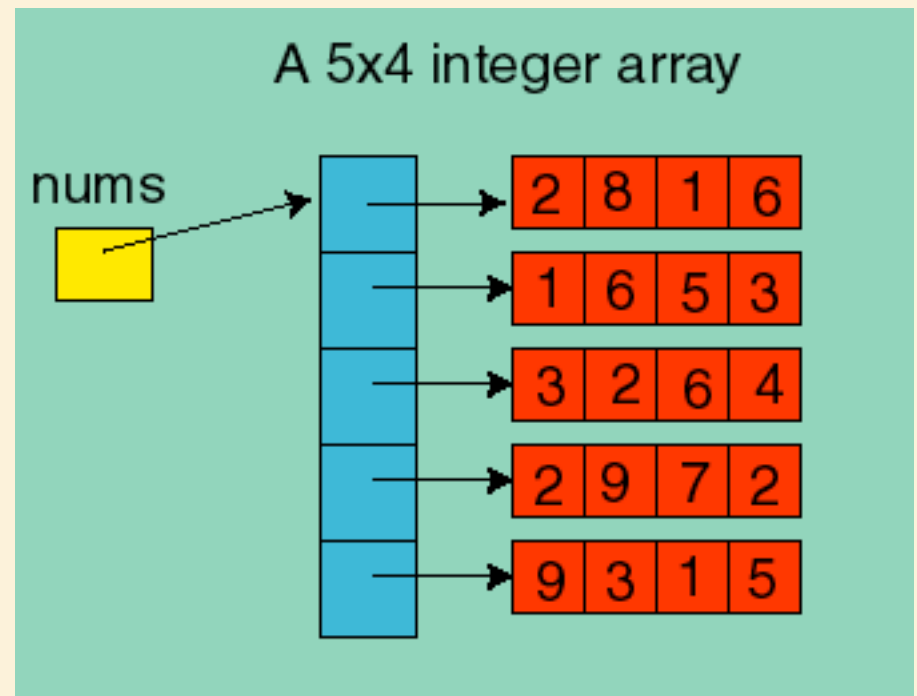
# Example

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

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

OR

- `int[][] nums;`  
`nums = new int[5][];`  
`for (int i=0; i<5; i++) {`  
    `nums[i] = new int[4];`  
`}`



# Assignment 1, Question 2

- In image processing, a frequent operation is to compute the average image value (intensity) or some other statistic over a rectangular sub-region of the image. The region of interest can be specified by 4 integer parameters: top, bottom, left and right.
- We will implement a class called **IntegrallImage**, which will allow the average over an arbitrary rectangular sub-image to be computed in  **$O(1)$  time**, using only  $O(n)$  memory.
- Images will be stored in 2D arrays. The first index of the array indicates the row number (vertical position) and the second index indicates the column number (horizontal position). Vertical position is indexed from the top down, and horizontal position is indexed from left to right. Thus `image[0][0]` is the top left pixel of the image.
- **The constructor method `IntegrallImage` will accept an input integer image and will construct, in  $O(n)$  time, a new data structure using  $O(n)$  memory that will allow  $O(1)$  computation of sub-image averages.** If the input array is not rectangular, `IntegrallImage` will throw an `InvalidImageException`.
- The query method `meanSubImage(top, bottom, left, right)` will return the average of all pixels in the subimage extending from top to bottom bounds vertically and left to right bounds horizontally, in  $O(1)$  time. The pixels lying at the bounds are included in this calculation.

End of Lecture

**JAN 15, 2015**

# 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 sorted ints for the specified value using the binary search algorithm (A must be sorted)

# What is Printed?

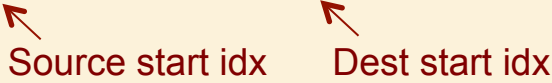
```
import java.util.Arrays;  
  
int[] A={12, 24, 37, 53, 67};  
  
int[] B=A.clone();  
  
if (A==B) {  
    System.out.println(" Superman ");  
}  
  
if (Arrays.equals(A, B)) {  
    System.out.println(" Snow White ");  
}
```

**Answer:** 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  $\Rightarrow$  out of memory
    - fix 1: use array size > number of expected items  $\Rightarrow$  waste of memory
    - fix 2: increase array size to fit the number of items  $\Rightarrow$  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;  
}
```



# Linear Data Structures

- Arrays (Ch. 3.1)
- **Array Lists (Ch. 7.2)**
- Stacks (Ch. 6.1)
- Queues (Ch. 6.2)
- Linked Lists (Ch. 3.2 – 3.4)



# The Array List ADT

- The **Array List** ADT extends the notion of array by storing a sequence of arbitrary objects
- An element can be accessed, modified, 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

*/\*\* Simplified version of java.util.List \*/*

**public interface** List<E> {

*/\*\* Returns the number of elements in this list \*/*

**public int** size();

*/\*\* Returns whether the list is empty. \*/*

**public boolean** isEmpty();

*/\*\* Append element e to end of list \*/*

**public void** add(E e);

*/\*\* Inserts an element e to be at index I, shifting all elements after this right. \*/*

**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 left. \*/*

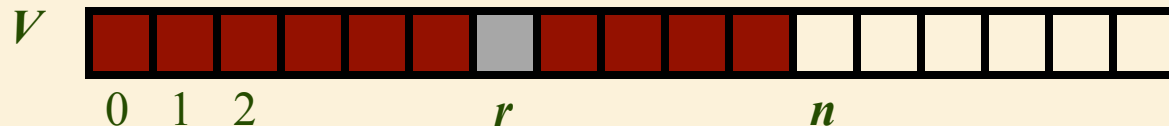
**public E** remove(int i) **throws** IndexOutOfBoundsException;

*/\*\* Replaces the element at index I with e, returning the previous element at i. \*/*

**public E** set(int I, E e) **throws** IndexOutOfBoundsException;

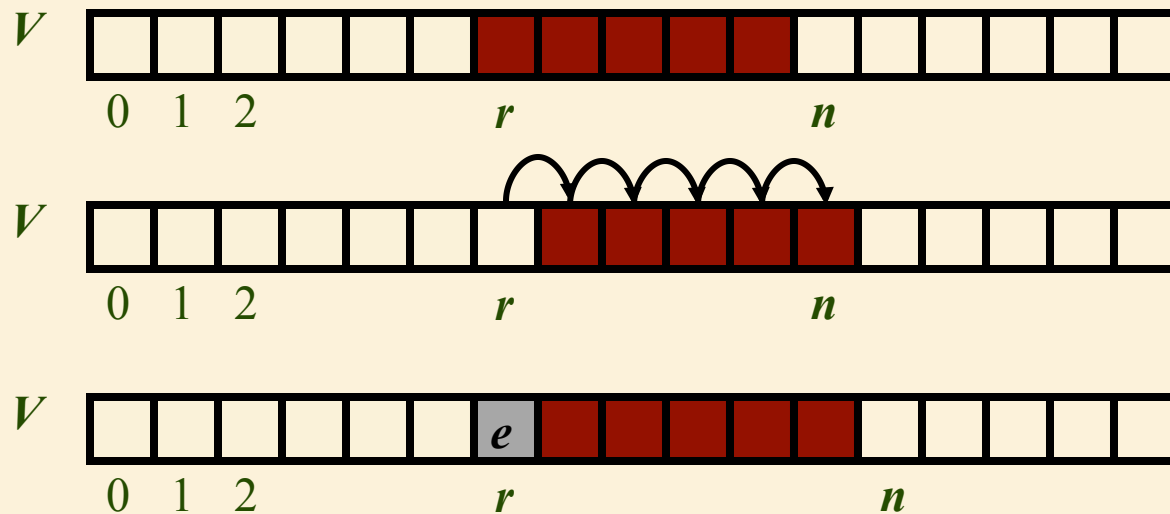
# A Simple Array-based Implementation

- 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  $V[r]$



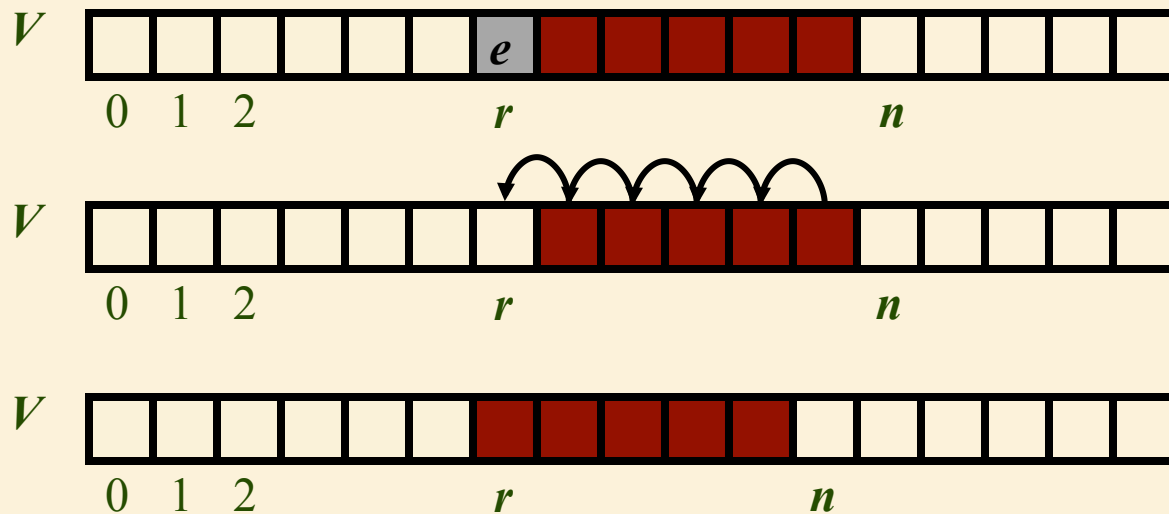
# Insertion

- In operation **add**( $r, e$ ), we need to make room for the new element by shifting forward the  $n - r$  elements  $V[r], \dots, V[n - 1]$
- 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  $V[r + 1], \dots, V[n - 1]$
- 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
- Let  $n$  = current number of elements in array

$N$  = capacity of array

## Algorithm **add**( $e$ )

```
if  $n = N$ 
then
   $A \leftarrow$  new array of size  $N^*$ 
   $N = N^*$ 
  for  $i \leftarrow 0$  to  $N-1$  do
     $A[i] \leftarrow S[i]$ 
   $S \leftarrow A$ 
 $n \leftarrow n + 1$ 
 $S[n] \leftarrow e$ 
```

# 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(e)** operations that append the object to the end of the list.
- We assume that we start with an empty array list ( $n = 0$ ) represented by an array of capacity 0 ( $N = 0$ ).
- The **amortized** time of an **add(e)** operation is the average time taken over the series of operations, i.e.,  $T(n)/n$



# Incremental Strategy

| <b>n</b> | <b>N</b>                               |              |
|----------|--|--------------|
| 0        | 0                                      |              |
| 1        | $c$                                    | Extend array |
| 2        | $c$                                    |              |
| $\vdots$ | $\vdots$                               |              |
| $c$      | $c$                                    |              |
| $c + 1$  | $2c$                                   | Extend array |
| $\vdots$ | $\vdots$                               |              |
| $2c$     | $2c$                                   |              |
| $2c + 1$ | $3c$                                   | Extend array |
| $\vdots$ | $\vdots$                               |              |
| $n$      | $kc$ , where $k = \lceil n / c \rceil$ |              |

# Incremental Strategy Analysis

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

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

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

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

- (Recall that JAVA initializes all elements of an allocated array.)
- Since  $c$  is a constant,  $T(n)$  is  $O(n + k^2)$ , i.e.,  $O(n^2)$
- The amortized time of an **add(e)** operation is  $O(n)$

# Doubling Strategy

| <b>n</b> | <b>N</b>                                 |              |
|----------|--|--------------|
| 0        | 0  |              |
| 1        | 1  | Extend array |
| 2        | 2  | Extend array |
| 3        | 4  | Extend array |
| 4        | 4  |              |
| 5        | 8  | Extend array |
| 6        | 8  |              |
| 7        | 8  |              |
| 8        | 8  |              |
| ⋮        | ⋮  |              |
| $n$      | $2^k$ , where $k = \lceil \log n \rceil$ |              |

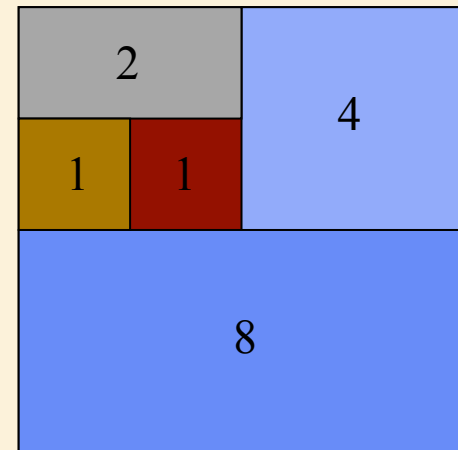
# Doubling Strategy Analysis

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

$$n + 1 + 2 + 4 + 8 + \dots + 2^k = n + 2^{k+1} - 1 \leq 5n$$

geometric series

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



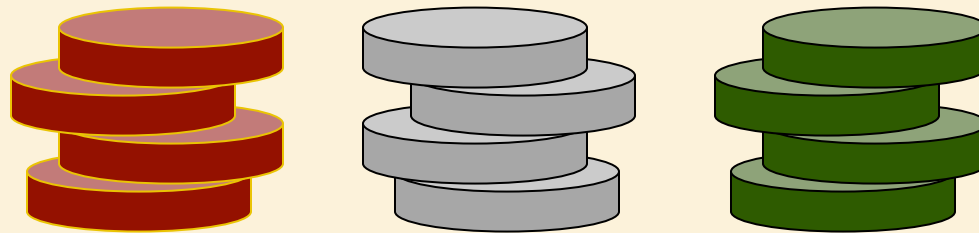
$$\left( \text{Recall: } \sum_{i=0}^n r^i = \frac{1 - r^{n+1}}{1 - r} \right)$$

# Applications of Array Lists

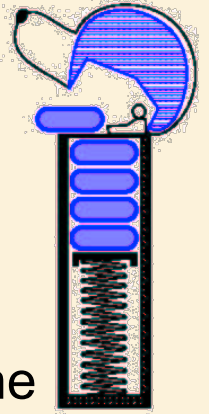
- Maintaining a sorted list when insertions and removals are relatively rare.
  - List of plant species by name
  - List of elements by atomic weight

# Linear Data Structures

- Arrays (Ch. 3.1)
- Array Lists (Ch. 7.2)
- **Stacks (Ch. 6.1)**
- Queues (Ch. 6.2)
- Linked Lists (Ch. 3.2 – 3.4)



# 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 or Pez dispenser
- Main stack operations:
  - **push**(e): 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

Note: `java.util.Stack` provides `push` and `pop`, but differs in other respects.

# 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
- Parsing math

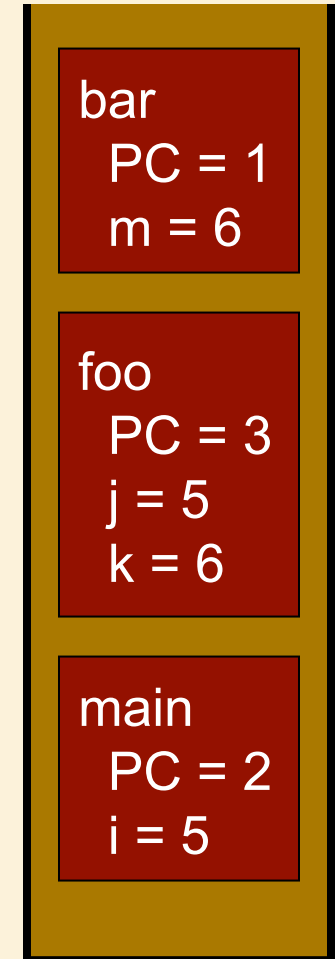
# Method Stack in the JVM

- 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

- The Stack ADT can be implemented with 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

$t \leftarrow t - 1$

return  $S[t + 1]$



# Array-based Stack (cont.)

- If using an array of fixed size, the stack may become full
- A push operation will then throw a **FullStackException**
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT
  - For example, in `java.util.Stack`, the array is extendable.

```
Algorithm push(e)  
  if  $t = S.length - 1$  then  
    throw FullStackException  
  else  
     $t \leftarrow t + 1$   
     $S[t] \leftarrow e$ 
```



# 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  $O(1)$

# 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;
}
```

# Example Application: Parenthesis Matching

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

End of Lecture  
**JAN 20, 2015**



# 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**

$S.\text{push}(X[i])$

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

**if**  $S.\text{isEmpty}()$  **then**

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

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

**return false** {wrong type}

**if**  $S.\text{isEmpty}()$  **then**

**return true** {every symbol matched}

**else**

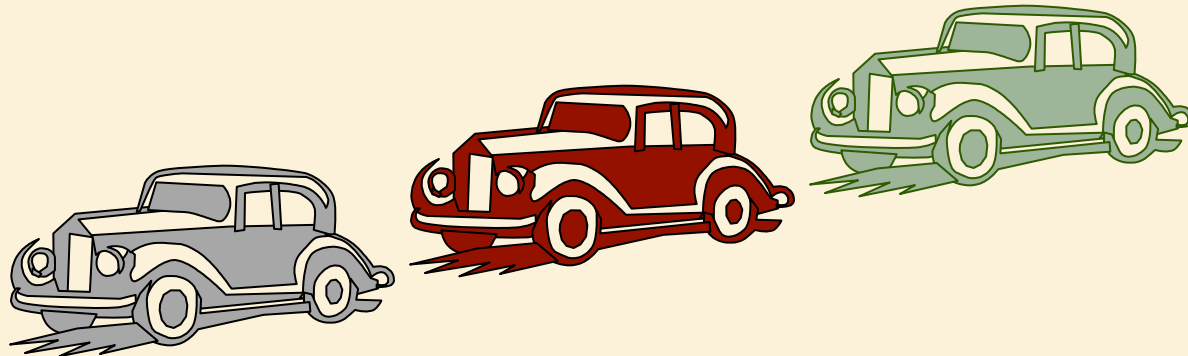
**return false** {some symbols were never matched}

# Assignment 1 Q3. Minimum Value Stack

- You are designing a new class **MinStack** that specializes the `java.util.Stack` class to `Comparable` objects and provides a **new method `min()`** that will return the minimum value on the stack in  **$O(1)$  time**.

# Linear Data Structures

- Arrays (Ch. 3.1)
- Array Lists (Ch. 7.2)
- Stacks (Ch. 6.1)
- **Queues (Ch. 6.2)**
- Linked Lists (Ch. 3.2 – 3.4)



# The Queue ADT

- The **Queue** ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out (FIFO) 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**.

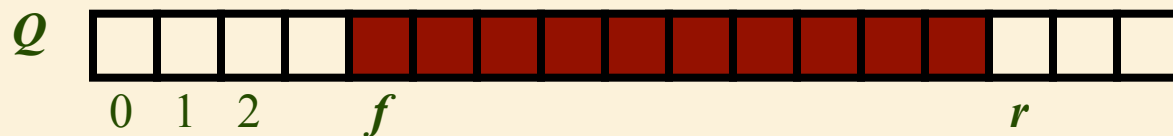
# Queue Example

| <i>Operation</i> | <i>Output</i> | <i>Q</i>     |
|------------------|---------------|--------------|
| 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)    |

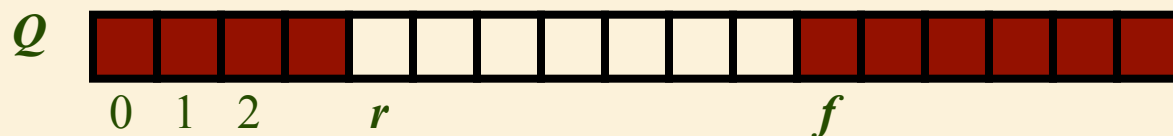
# Array-Based Queue

- 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

normal configuration



wrapped-around configuration



# Queue Operations

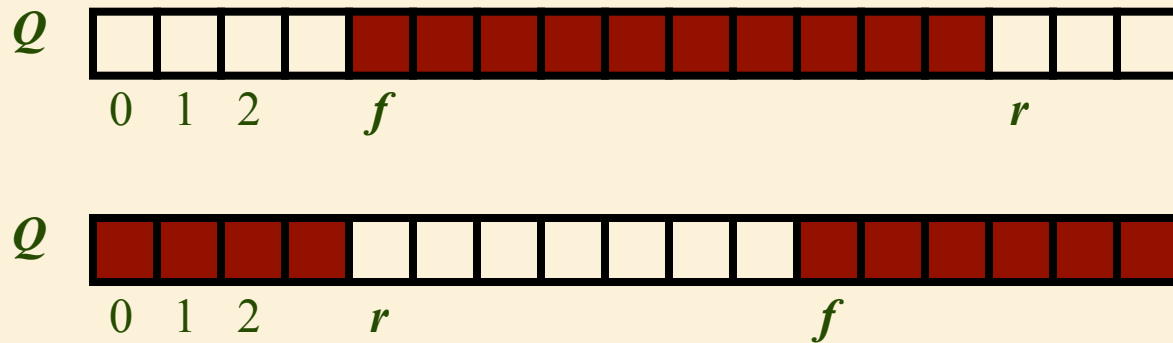
- We use the modulo operator (remainder of division)

Algorithm *size()*

return  $(r + N - f) \bmod N$

Algorithm *isEmpty()*

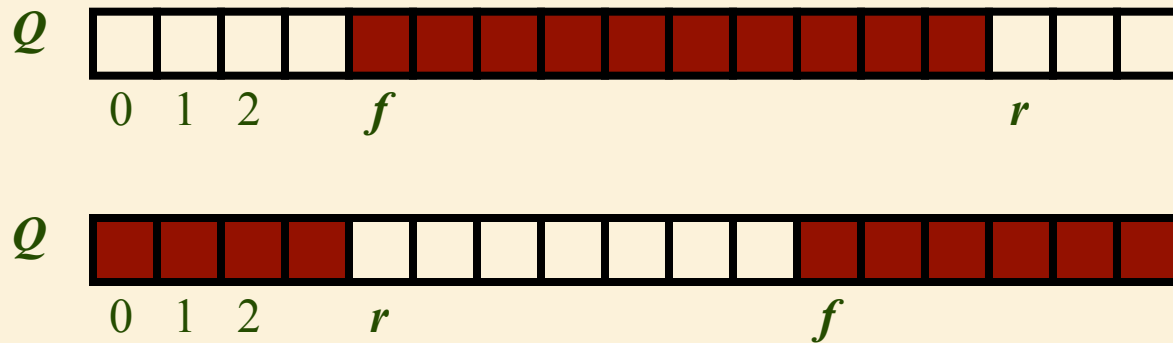
return  $(f = r)$



# Queue Operations (cont.)

- Operation enqueue may throw an exception if the array is full

```
Algorithm enqueue(e)  
  if size() =  $N - 1$  then  
    throw FullQueueException  
  else  
     $Q[r] \leftarrow e$   
     $r \leftarrow (r + 1) \bmod N$ 
```

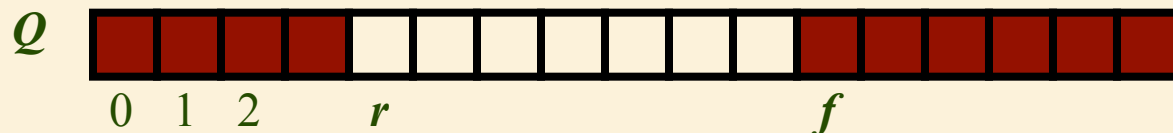




# Queue Operations (cont.)

- Operation `dequeue` throws an exception if the queue is empty

```
Algorithm dequeue()  
  if isEmpty() then  
    throw EmptyQueueException  
  else  
     $e \leftarrow Q[f]$   
     $f \leftarrow (f + 1) \bmod N$   
    return  $e$ 
```



# Queue Interface in Java

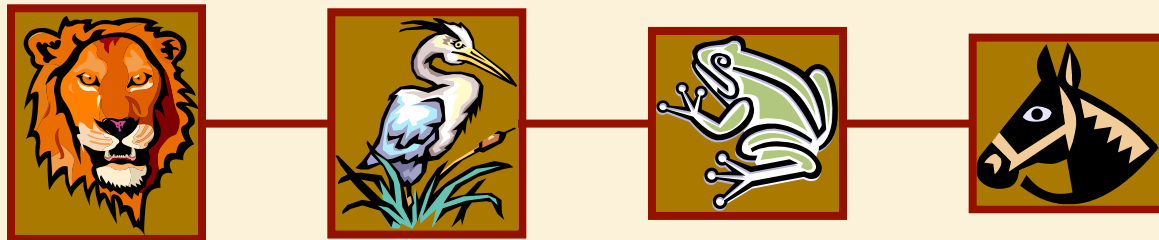
- Java interface corresponding to our Queue ADT
- Requires the definition of class `EmptyQueueException`

NB: The `java.util.Queue` interface uses quite different signatures, and does not insist that the queue be FIFO.

```
public interface Queue {  
    public int size();  
    public boolean isEmpty();  
    public Object front()  
        throws EmptyQueueException;  
    public void enqueue(Element e);  
    public Object dequeue()  
        throws EmptyQueueException;  
}
```

# Linear Data Structures

- Arrays (Ch. 3.1)
- Array Lists (Ch. 7.2)
- Stacks (Ch. 6.1)
- Queues (Ch. 6.2)
- **Linked Lists (Ch. 3.2 – 3.4)**

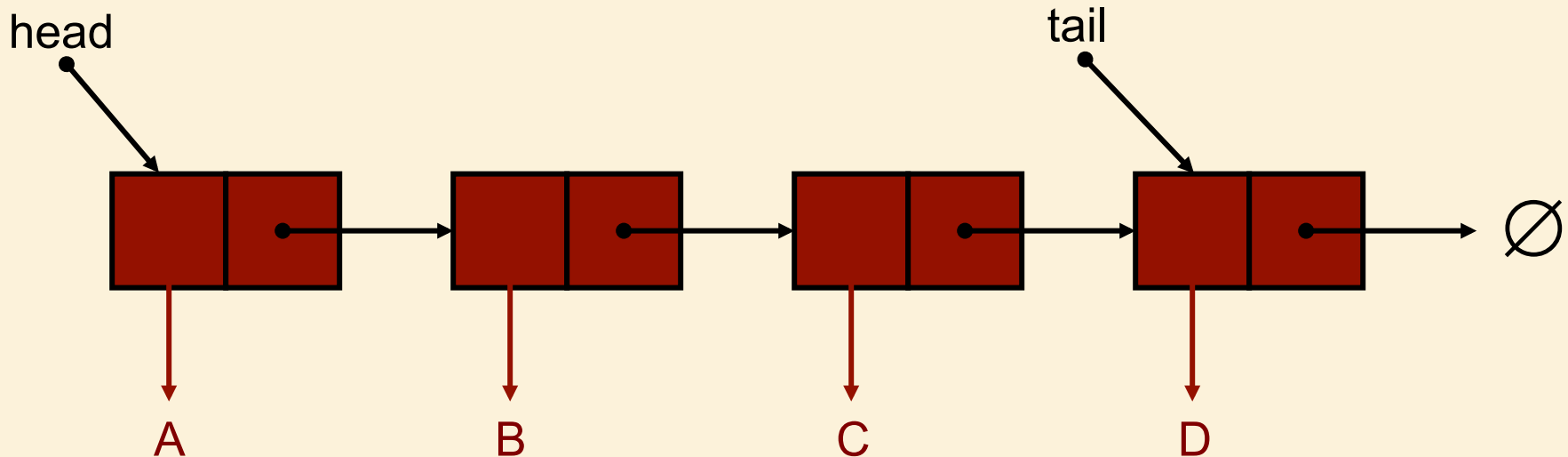
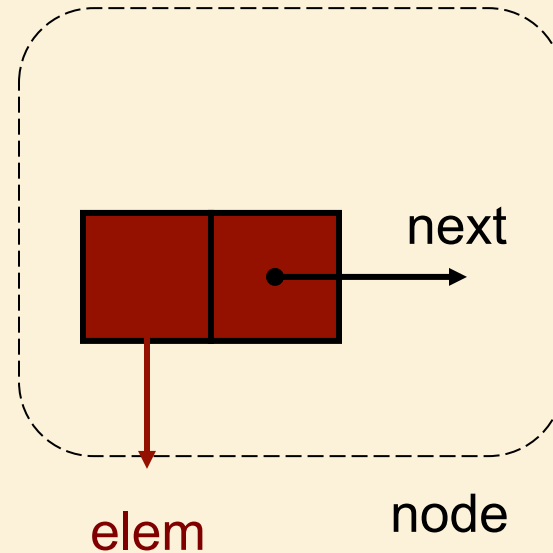


# Linked Lists

- By virtue 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 and inputs, structural operations may dominate the running time.
- For such cases, **linked lists** may be more appropriate.

# Singly Linked List

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



# Example Java Class for Singly-Linked Nodes

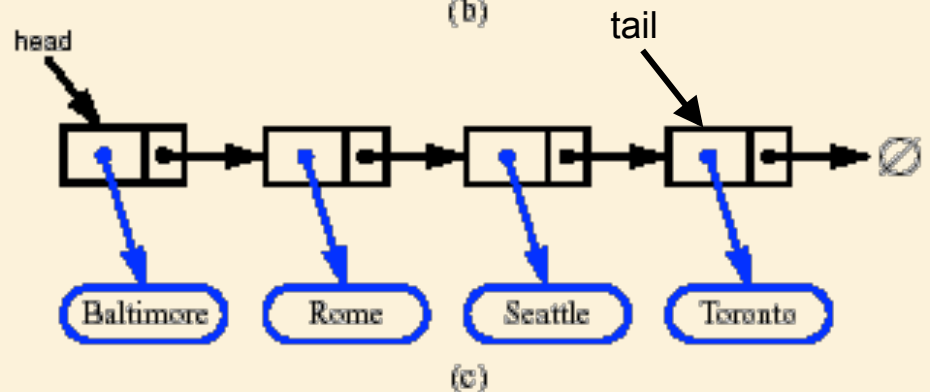
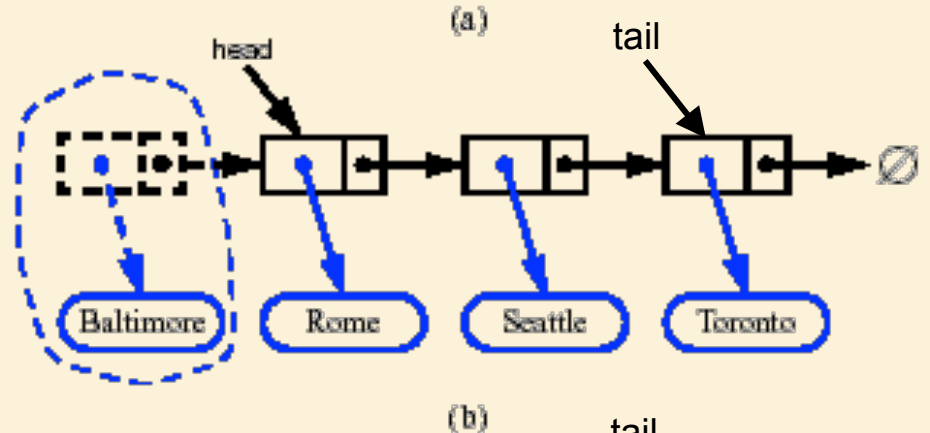
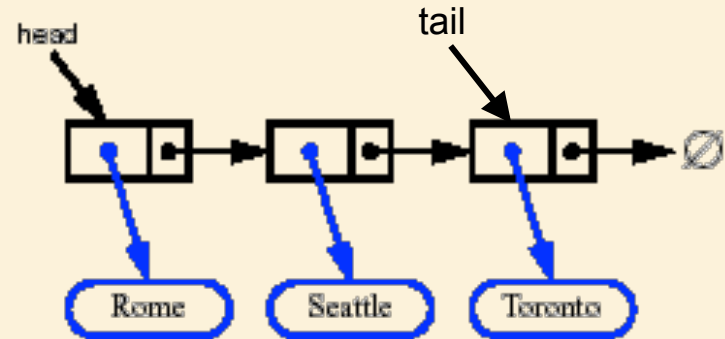
```
Private static class Node<E> {  
    // Instance variables:  
    private E element;  
    private Node<E> next;  
  
    /** Create a node with the given element and next node. */  
    public Node(E e, Node<E> n) {  
        element = e;  
        next = n;  
    }  
  
    // Accessor methods:  
    public E getElement() { return element; }  
    public Node<E> getNext() { return next; }  
  
    // Modifier methods:  
    public void setNext(Node<E> n) { next = n; }
```

# Example Java Class for Singly-Linked List

```
public class SinglyLinkedList<E> {  
    // Instance variables:  
    private Node head = null; //head node of list  
    private Node tail = null; //tail node of list  
    private int size = 0; //number of nodes in list  
    /** Default constructor that creates an empty list. */  
    public SinglyLinkedList() {}  
    // update and search methods go here...  
}
```

# Inserting at the Head

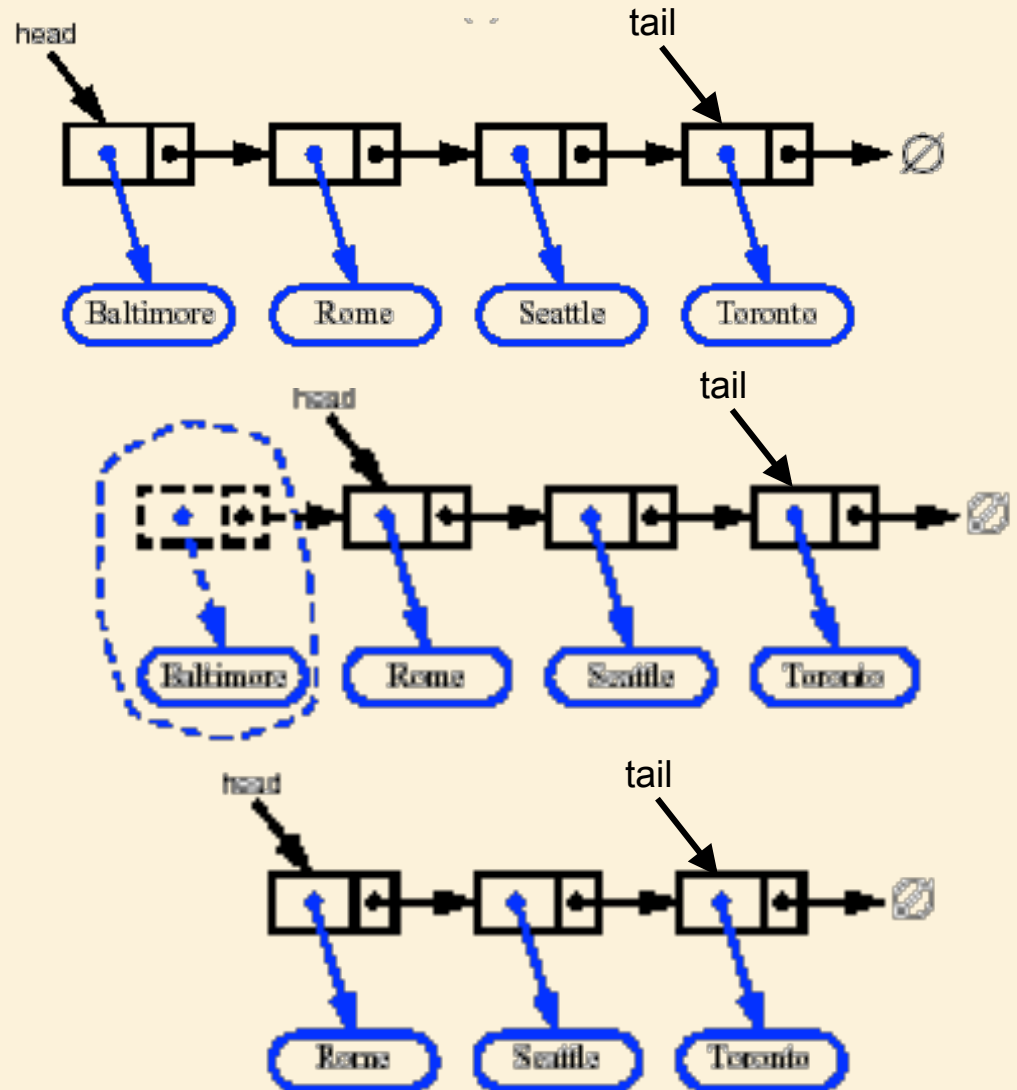
1. Construct new node with new element
2. Have new node point to old **head**
3. Update **head** to point to new node
4. If list was initially empty, have to update **tail** as well.





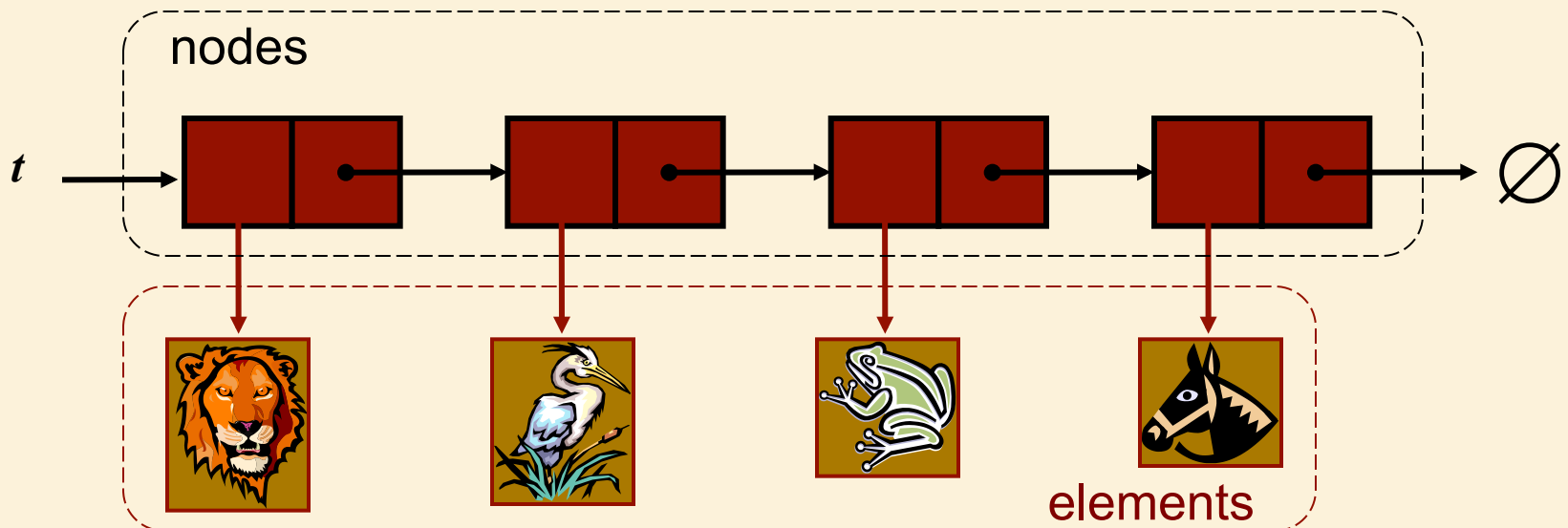
# Removing at the Head

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.



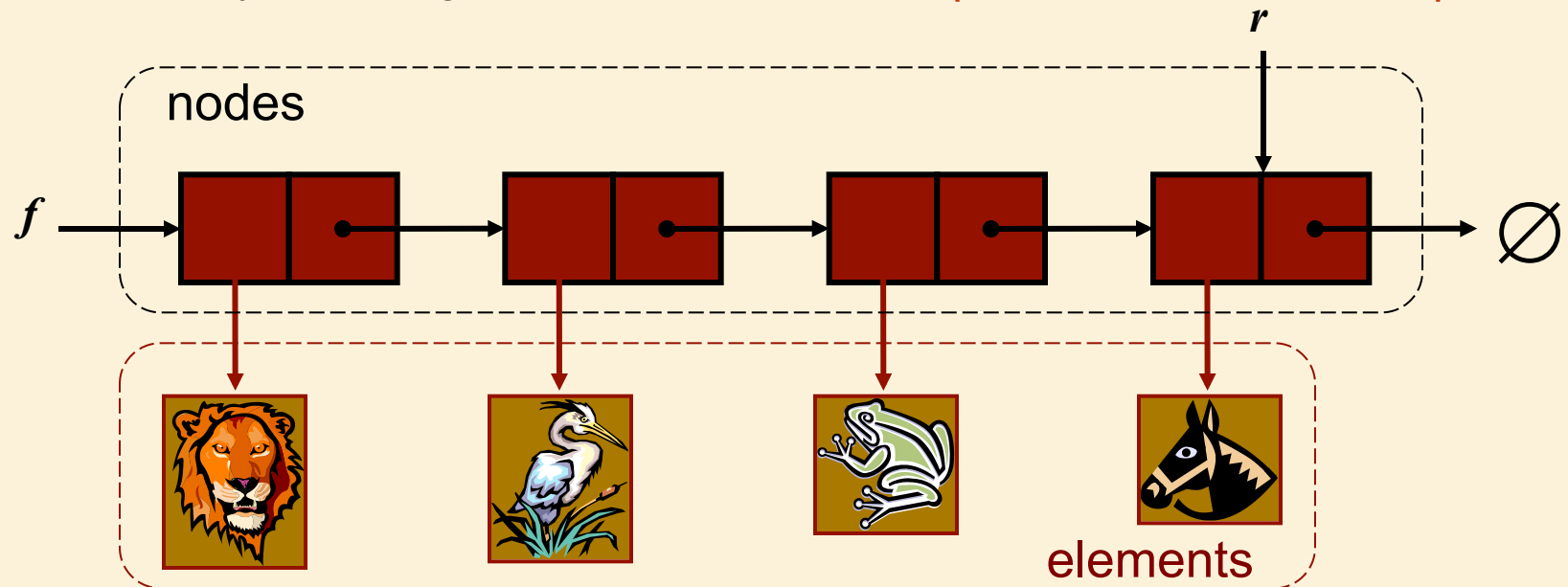
# Example Application: Implementing a Stack with a Singly-Linked List

- 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
- Are there any advantages? No **FullStackException** or **FullQueueException** !



# Assignment 1 Q1. Sparse Vectors

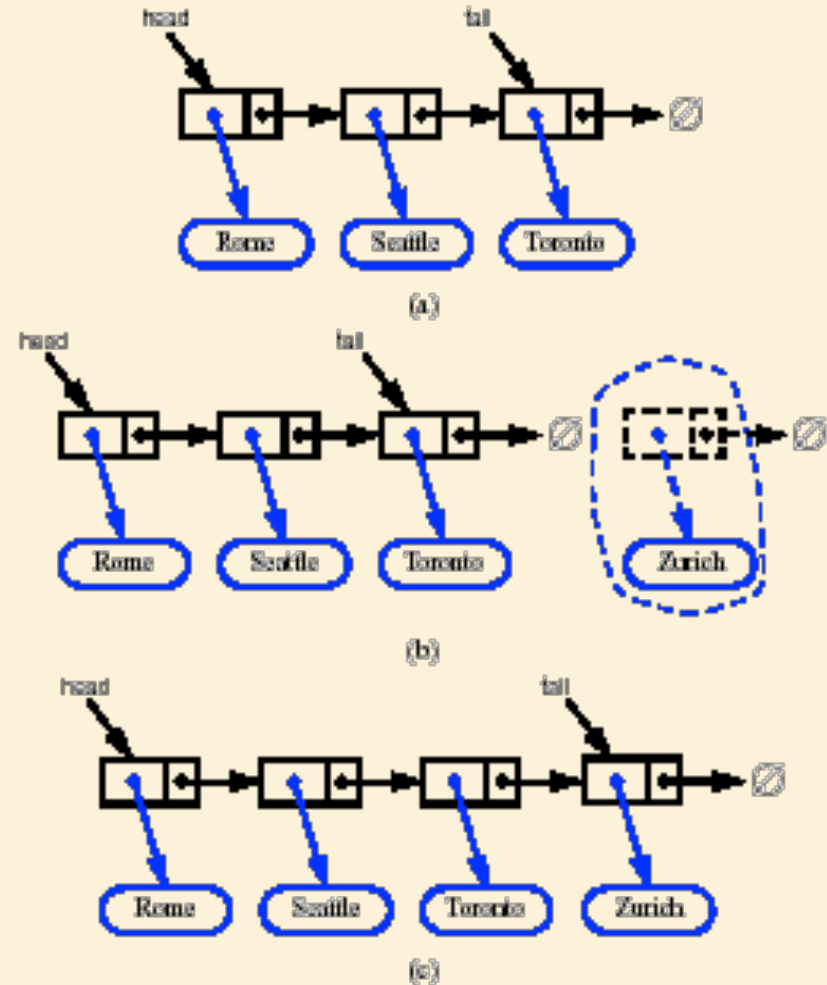
- In many applications, vectors are sparse, that is, they contain mainly 0 elements, e.g., [0 0 0 0 13.4 0 0 14.5 0 0 0 12.2].
- It is inefficient to represent such vectors as regular arrays. Instead, we consider a data structure that only represents data values that are non-zero.
- It will do this by maintaining both the value and the one-based position of each value in the vector. Thus the previous vector can be represented as [(5, 13.4), (8, 14.5), (12, 12.2)].
- We will represent this data structure as a singly-linked list, with elements containing both position indices and values.

# Running Time

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

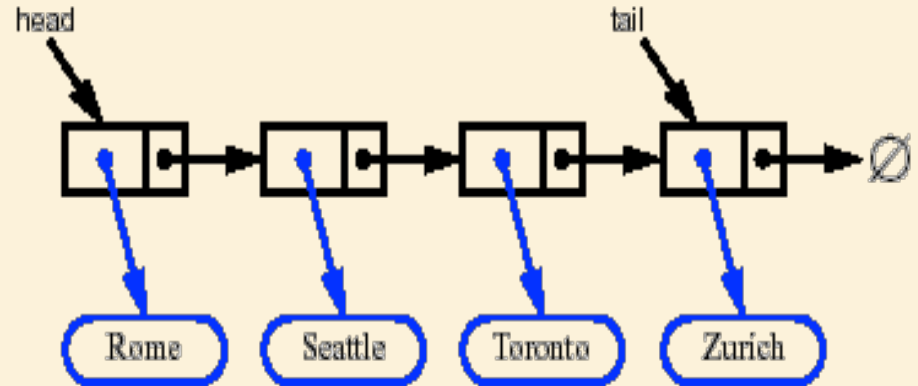
# Inserting at the Tail

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



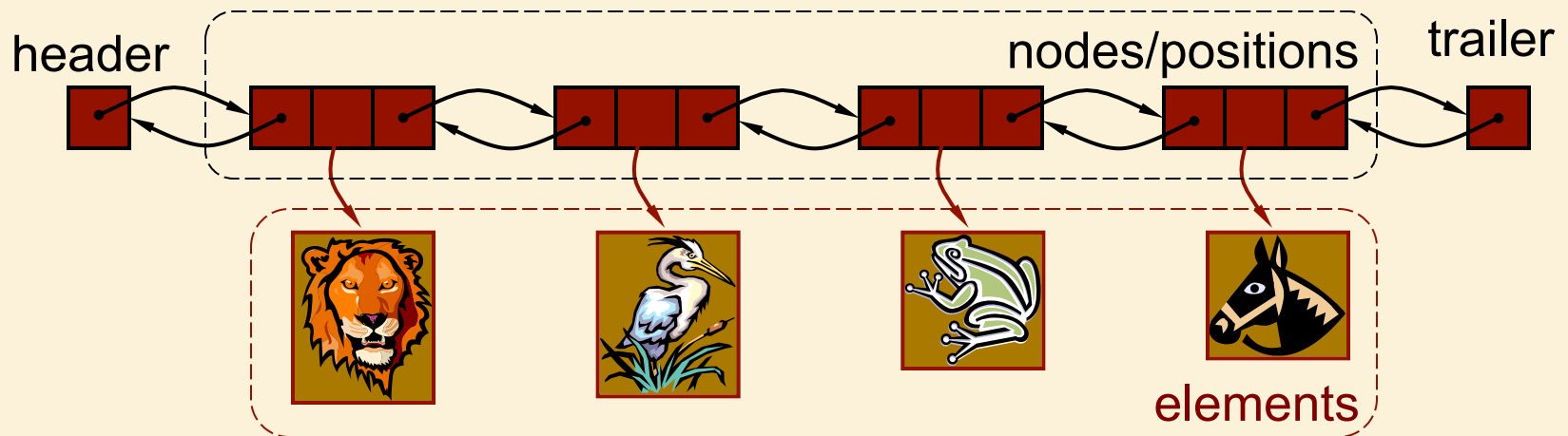
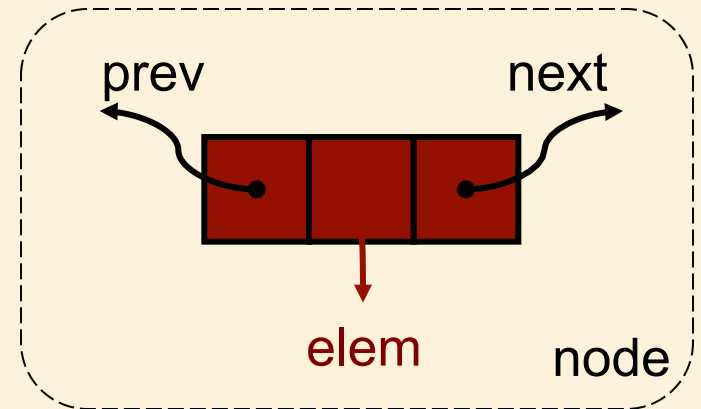
# Removing at the Tail

- 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
- How could we solve this problem?



# Doubly Linked List

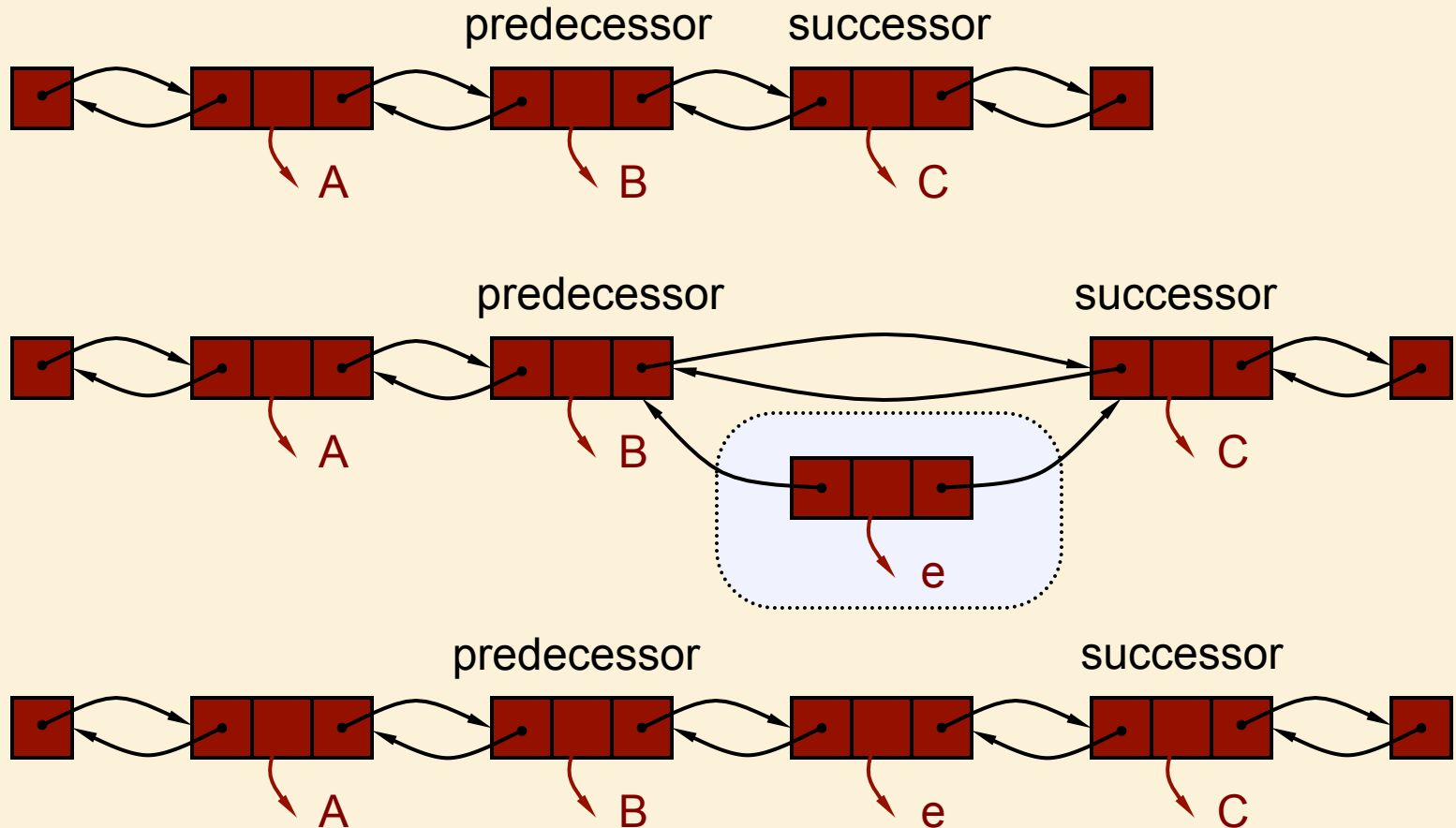
- Doubly-linked lists allow more flexible list management (constant time operations at both ends).
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header (sentinel) nodes





# Insertion

- **addBetween**(E e Node<E> predecessor, Node<E> successor)  
constructs a new node containing element e and inserts it  
between predecessor and successor nodes

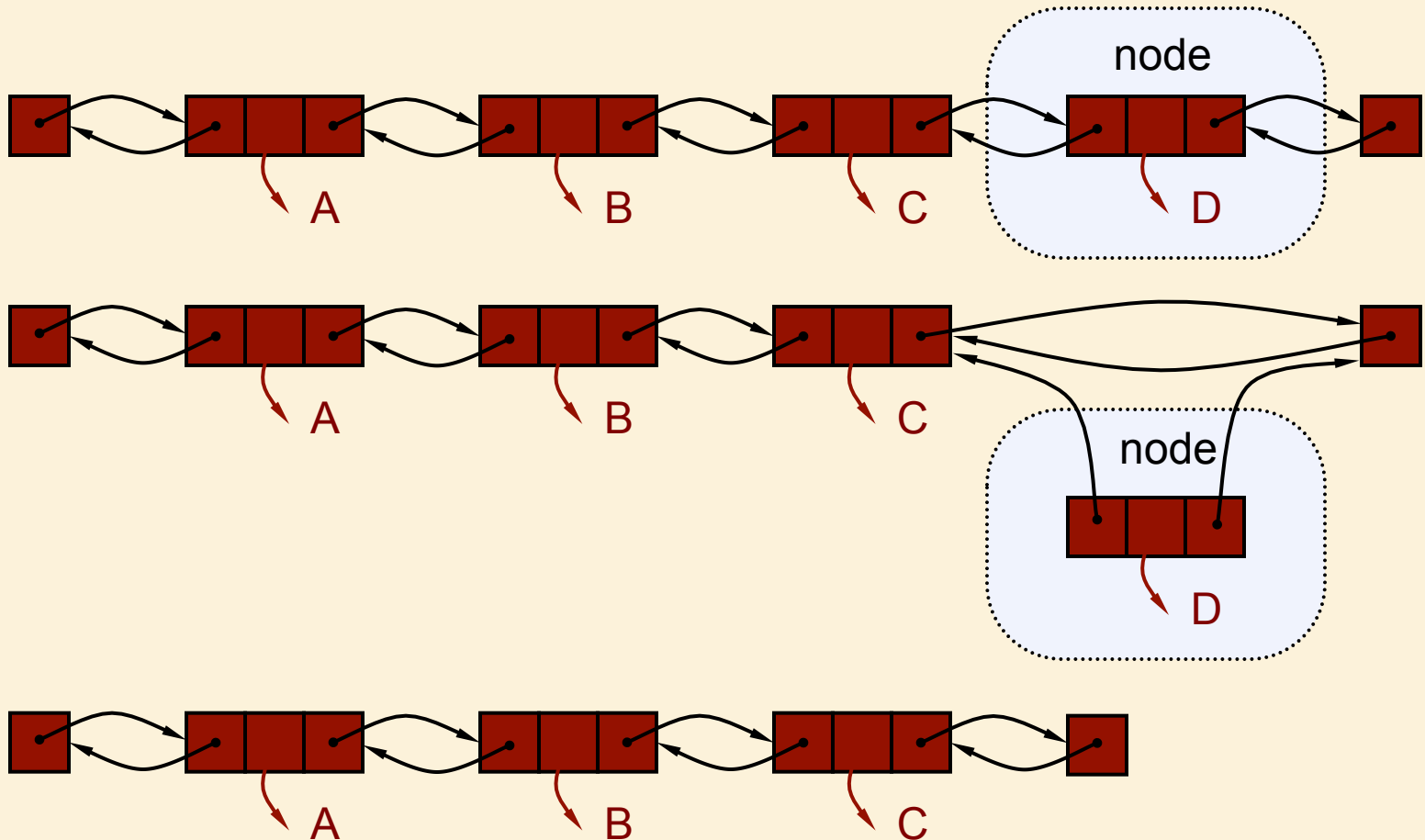


# Insertion Algorithm

```
private void addBetween(E e, Node<E> predecessor, Node<E>
    successor) {
    Node<E> newest = new Node<>(e, predecessor, successor);
    predecessor.setNext(newest); {link forward}
    successor.setPrev(newest);   {link back}
    size++;
}
```

# Deletion

- **Remove**(Node<E> node) removes node from the list and returns the element that had been stored there.



# Deletion Algorithm

```
Private E remove(Node<E> node) {  
    Node<E> predecessor = node.getPrev();  
    Node<E> sucessor = node.getNext();  
    predecessor.setNext(successor);           {link forward}  
    sucessor.setPrev(predecessor);           {link back}  
    size--;  
    return node.getElement();  
}
```

# 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!

# Linear Data Structures

- Arrays (Ch. 3.1)
- Array Lists (Ch. 7.2)
- Stacks (Ch. 6.1)
- Queues (Ch. 6.2)
- Linked Lists (Ch. 3.2 – 3.4)

# Learning Outcomes

- Based on this lecture, you should:
  - Know the basic linear data structures
  - Be able to express each as an Abstract Data Type (ADT)
  - Be able to specify each of these ADTs as a Java interface.
  - Be able to outline the algorithms for creating, accessing and modifying each data structure
  - Be able to analyze the running time of these operations
  - Be able to identify particular applications for which each data structure would be suited.