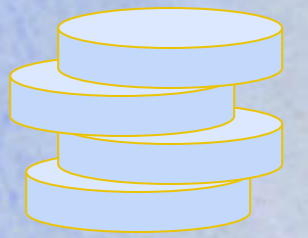
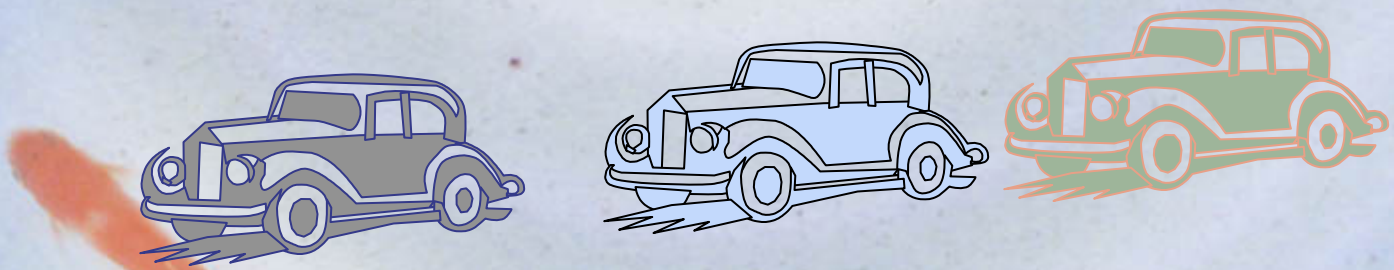
Data Structures and Algorithms



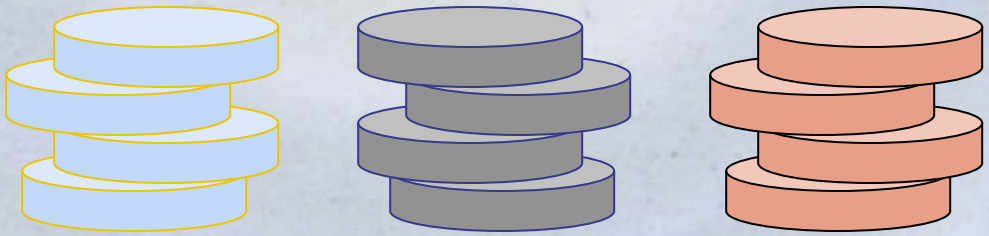
Stacks and Queues



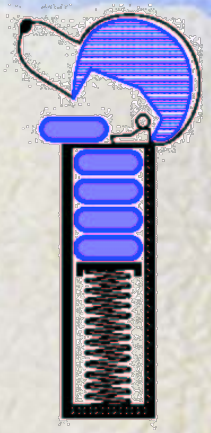
Outline

- Stacks
- Queues
- Sequence

Stacks



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

- Java interface corresponding to our Stack ADT
- Requires the definition of class EmptyStackException
- Different from the built-in Java class java.util.Stack

```
public interface Stack {

   public int size();

   public boolean isEmpty();

   public Object top()
         throws EmptyStackException;

   public void push(Object o);

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

Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be "thrown" by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException

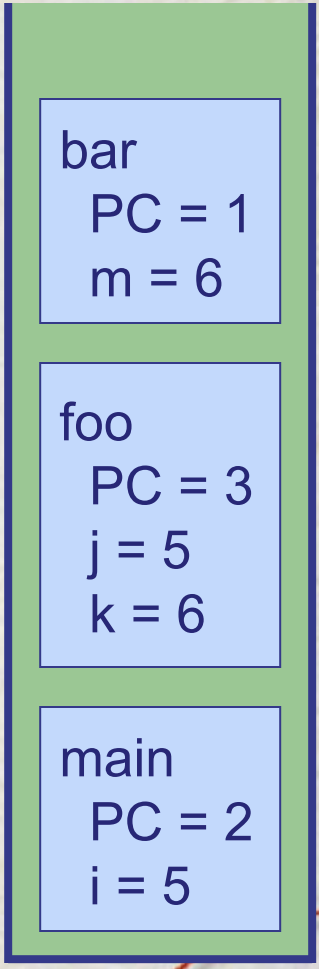
Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

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

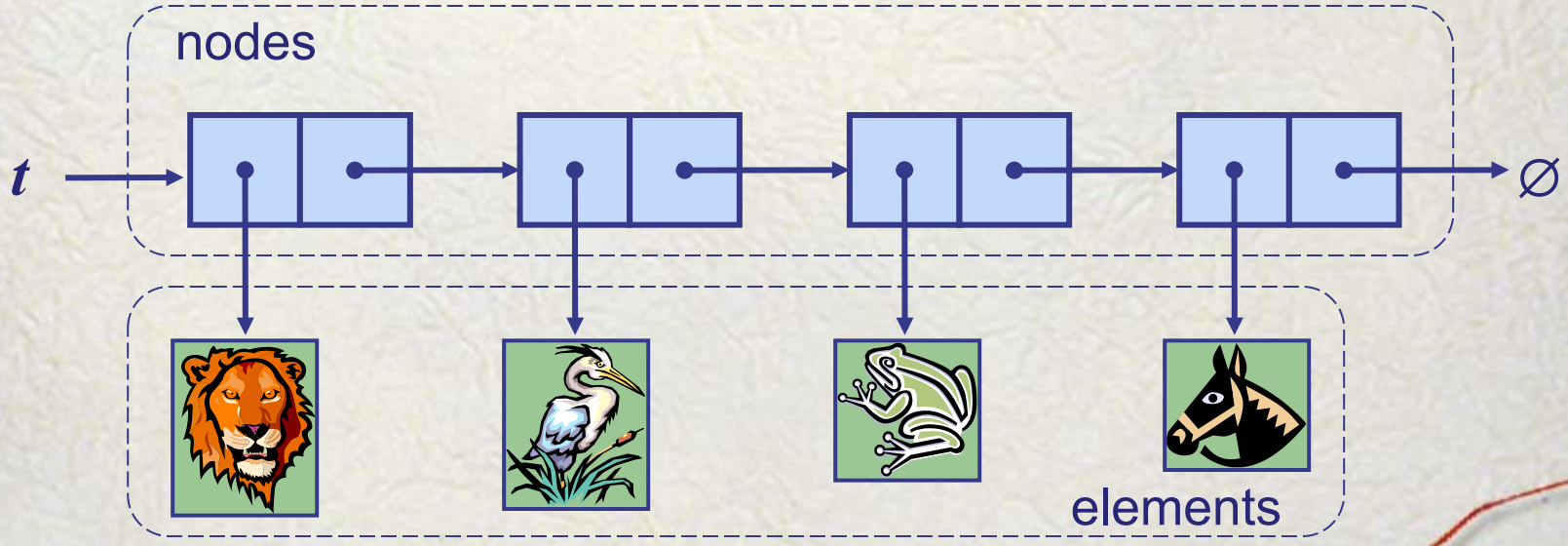
Array-based Stack in Java

```
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;
}
```

Stack with a Singly Linked List

- We can 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



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}

HTML Tag Matching

- For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

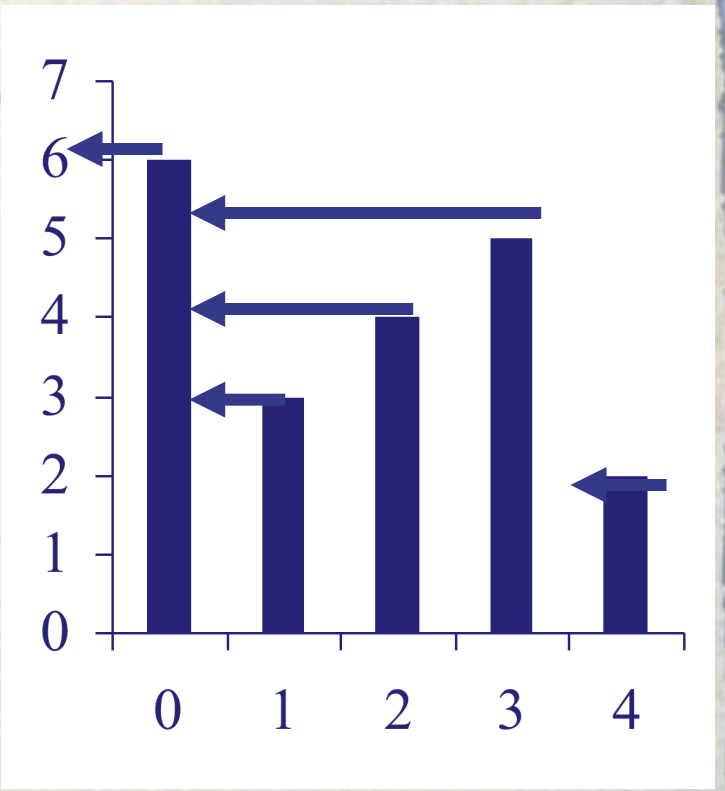
The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Computing Spans

- We show how to use a stack as an auxiliary data structure in an algorithm
- Given an an array X, the span $S[i]$ of $X[i]$ is the maximum number of consecutive elements $X[j]$ immediately preceding $X[i]$ and such that $X[j] \leq X[i]$
- Spans have applications to financial analysis
  - E.g., stock at 52-week high



X	6	3	4	5	2
S	1	1	2	3	1

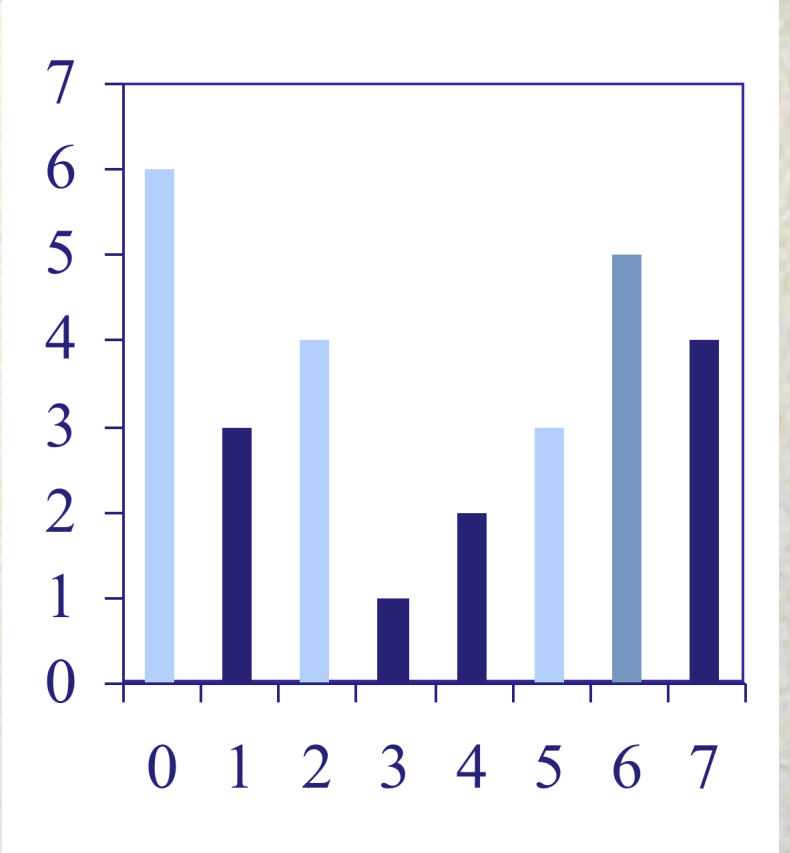
Quadratic Algorithm

```
Algorithm spans1(X, n)
    Input array X of n integers
    Output array S of spans of X                    #
    S ← new array of n integers                     n
    for i ← 0 to n − 1 do                           n
        s ← 1                                       n
        while s ≤ i ∧ X[i − s] ≤ X[i]               1 + 2 + …+ (n − 1)
            s ← s + 1                               1 + 2 + …+ (n − 1)
        S[i] ← s                                    n
    return S                                        1
```

- Algorithm *spans1* runs in $O(n^2)$ time

Computing Spans with a Stack

- We keep in a stack the indices of the elements visible when "looking back"
- We scan the array from left to right
  - Let i be the current index
  - We pop indices from the stack until we find index j such that $X[i] < X[j]$
  - We set $S[i] \leftarrow i - j$
  - We push i onto the stack

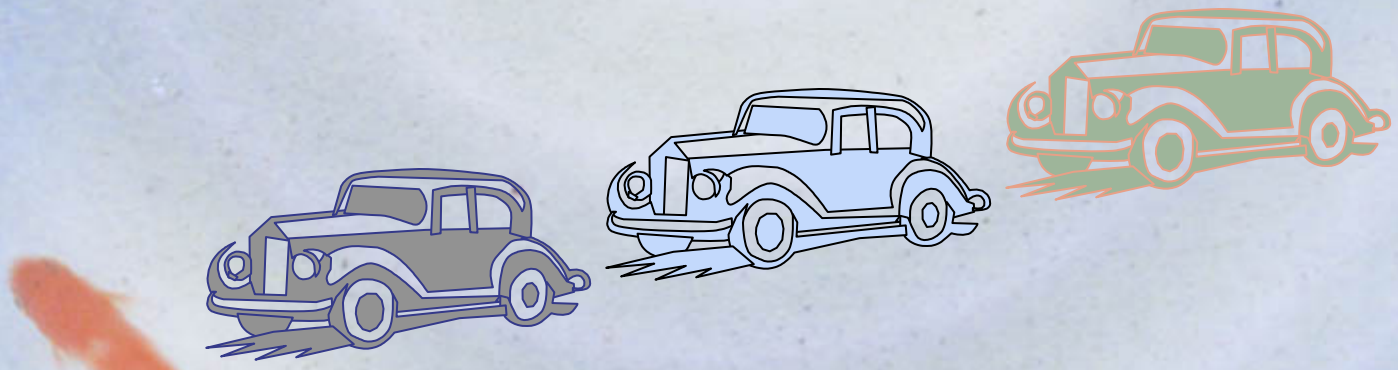


Linear Algorithm

- Each index of the array
  - Is pushed into the stack exactly one
  - Is popped from the stack at most once
- The statements in the while-loop are executed at most n times
- Algorithm *spans2* runs in $O(n)$ time

```
Algorithm spans2(X, n)                          #
    S ← new array of n integers                 n
    A ← new empty stack                         1
    for i ← 0 to n − 1 do                       n
        while (¬A.isEmpty() ∧
                X[A.top()] ≤ X[i] ) do          n
            A.pop()                             n
        if A.isEmpty() then                     n
            S[i] ← i + 1                        n
        else
            S[i] ← i − A.top()                  n
        A.push(i)                               n
    return S                                    1
```

Queues



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

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)

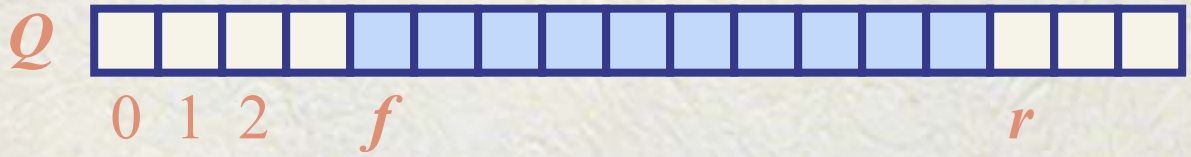
Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

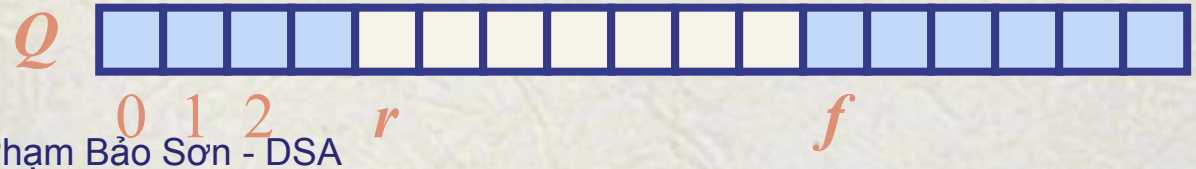
Array-based Queue

- 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

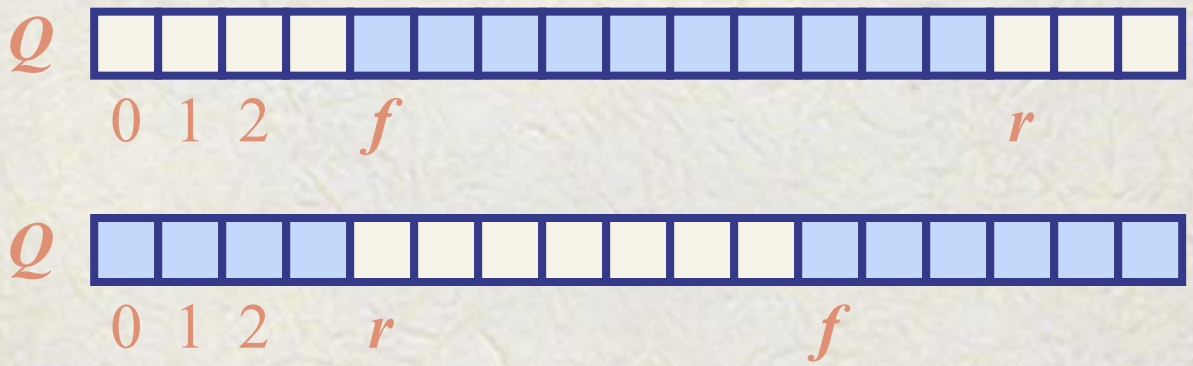


Queue Operations

- We use the modulo operator (remainder of division)

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

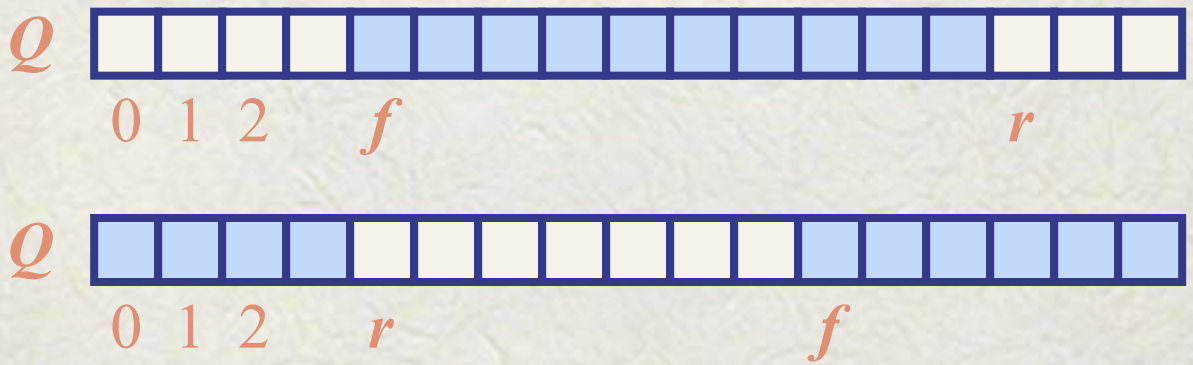
Algorithm *isEmpty*()
  return $(f = r)$



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

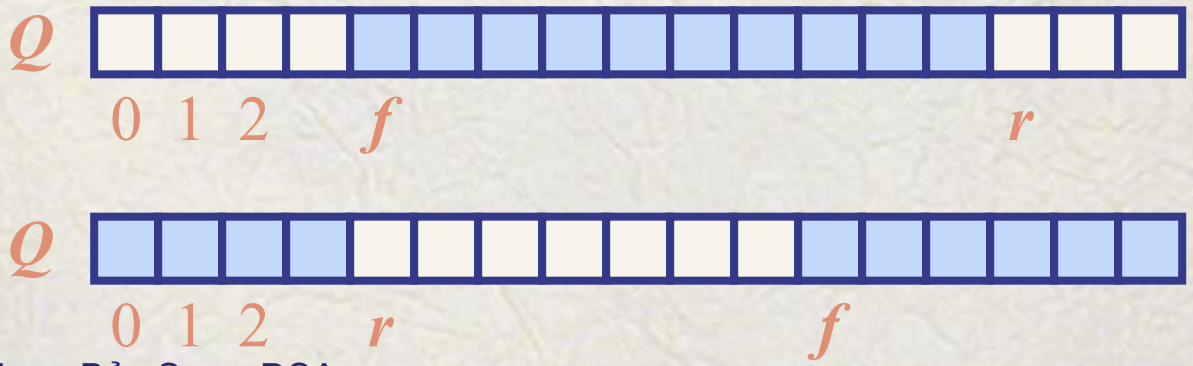
```
Algorithm enqueue(o)
  if size() = N - 1 then
    throw FullQueueException
  else
    Q[r] ← o
    r ← (r + 1) mod N
```



Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

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



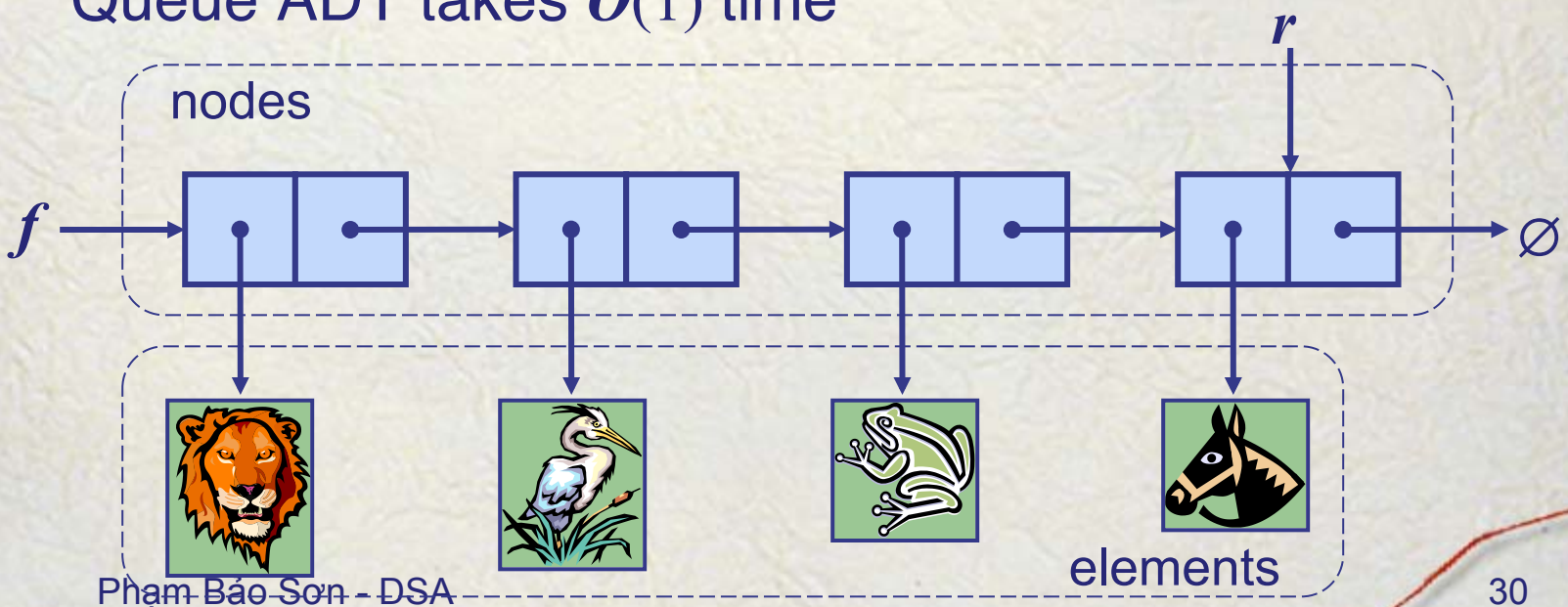
Queue Interface in Java

- 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;
}
```

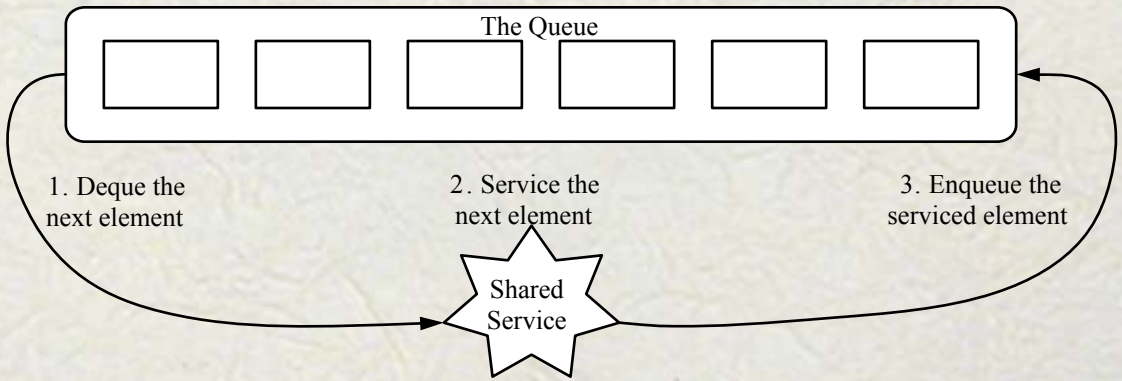
Queue with a Singly Linked List

- We can implement a queue with a singly linked list
  - 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



Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue, *Q*, by repeatedly performing the following steps:
  1. *e* = *Q*.dequeue()
  2. Service element *e*
  3. *Q*.enqueue(*e*)



Sequence ADT

- The Sequence ADT is the union of the Vector and List ADTs
- Elements accessed by
  - Rank, or
  - Position
- Generic methods:
  - size(), isEmpty()
- Vector-based methods:
  - elemAtRank(r), replaceAtRank(r, o), insertAtRank(r, o), removeAtRank(r)
- List-based methods:
  - first(), last(), prev(p), next(p), replace(p, o), insertBefore(p, o), insertAfter(p, o), insertFirst(o), insertLast(o), remove(p)
- Bridge methods:
  - atRank(r), rankOf(p)

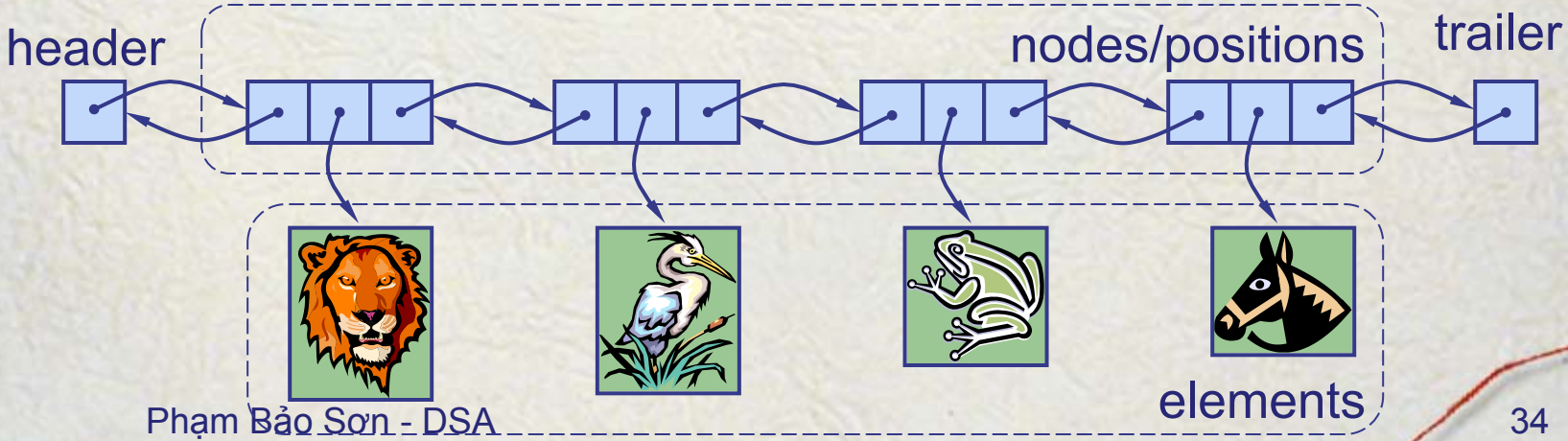
Applications of Sequences

- The Sequence ADT is a basic, general-purpose, data structure for storing an ordered collection of elements
- Direct applications:
  - Generic replacement for stack, queue, vector, or list
  - small database (e.g., address book)
- Indirect applications:
  - Building block of more complex data structures

Linked List Implementation

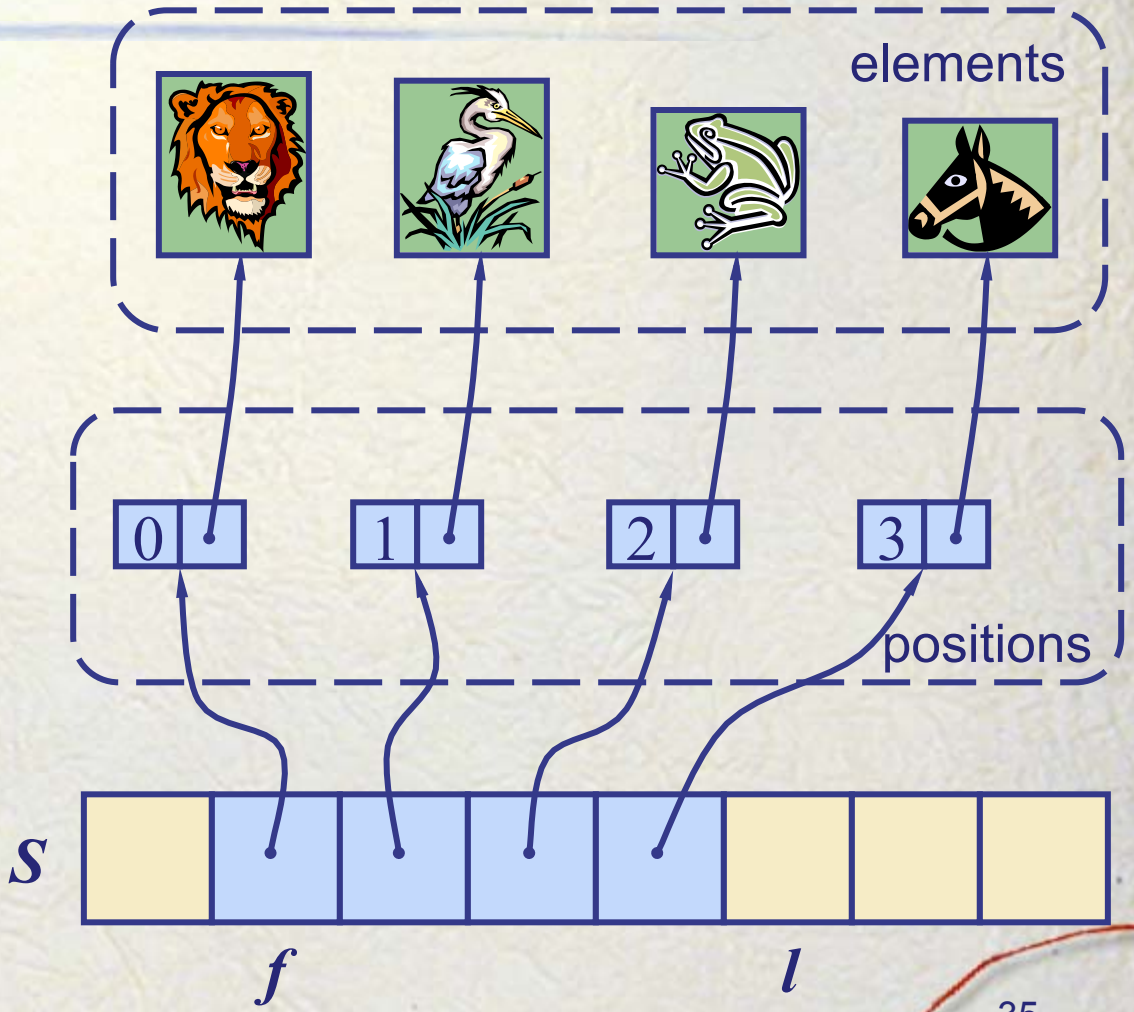
- A doubly linked list provides a reasonable implementation of the Sequence ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes

- Position-based methods run in constant time
- Rank-based methods require searching from header or trailer while keeping track of ranks; hence, run in linear time



Array-based Implementation

- We use a circular array storing positions
- A position object stores:
  - Element
  - Rank
- Indices f and l keep track of first and last positions



Sequence Implementations

Operation	Array	List
size, isEmpty	1	1
atRank, rankOf, elemAtRank	1	n
first, last, prev, next	1	1
replace	1	1
replaceAtRank	1	n
insertAtRank, removeAtRank	n	n
insertFirst, insertLast	1	1
insertAfter, insertBefore	n	1
remove	n	1