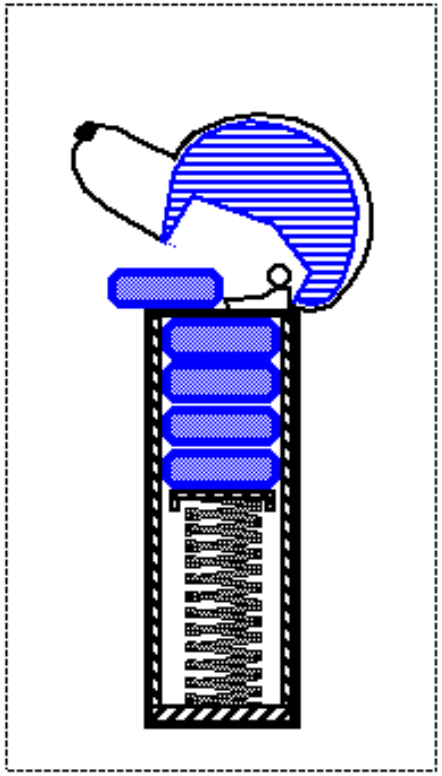
Stacks & Queues

Data structures and Algorithms

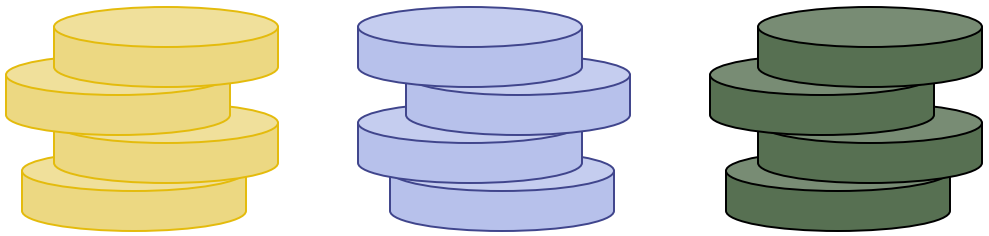


Acknowledgement:
These slides are adapted from slides provided with *Data Structures and Algorithms in C++*
Goodrich, Tamassia and Mount (Wiley, 2004)

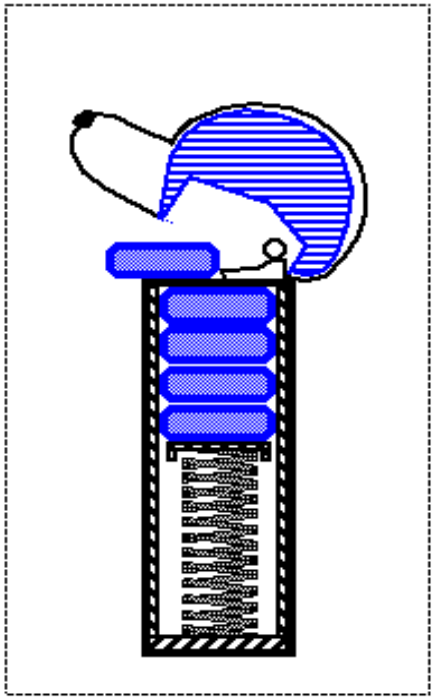
Outline and Reading

- The Stack ADT (§5.1.1)
  - Applications of Stacks (§5.1.5)
  - Array-based implementation (§5.1.2)
  - List-based stack (§5.1.3)
  - Applications (§5.1.5)
- The Queue ADT (§5.2.1)
  - Implementation with a circular array (§5.2.2)
  - List-based queue (§5.2.3)
  - Round Robin schedulers (§5.2.4)

Stacks



The Stack ADT



- Stack ADT stores arbitrary objects
- Insertions and deletions follow last-in first-out (LIFO) scheme
- Main stack operations:
  - push(object): inserts an element
  - pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - top(): returns the last inserted element without removing it
  - size(): returns the number of elements stored
  - isEmpty(): returns a Boolean value indicating whether no elements are stored

Stack Example

Operation	output	stack
push(8)	-	(8)
push(3)	-	(3, 8)
pop()	3	(8)
push(2)	-	(2, 8)
push(5)	-	(5, 2, 8)
top()	5	(5, 2, 8)
pop()	5	(2, 8)
pop()	2	(8)
pop()	8	()
pop()	"error"	()
push(9)	-	(9)
push(1)	-	(1, 9)

Stack Interface in C++

- Interface corresponding to our Stack ADT
- Requires the definition of class EmptyStackException
- Corresponding STL construct: stack

```
template <typename Object>
class Stack {
public:
    int size() const;
    bool isEmpty() const;
    Object& top()
        throw(EmptyStackException);
    void push(const Object& o);
    Object pop()
        throw(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
  - Saving local variables when one function calls another, and this one calls another, and so on.
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

C++ Run-time Stack

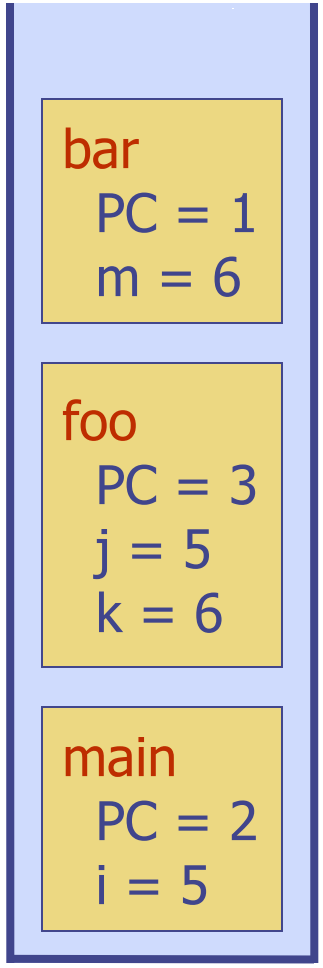
- The C++ run-time system keeps track of the chain of active functions with a stack
- When a function is called, the run-time system pushes on the stack a frame containing:
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
  int i;

  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
- array-based implementation of stack ADT

- 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 C++

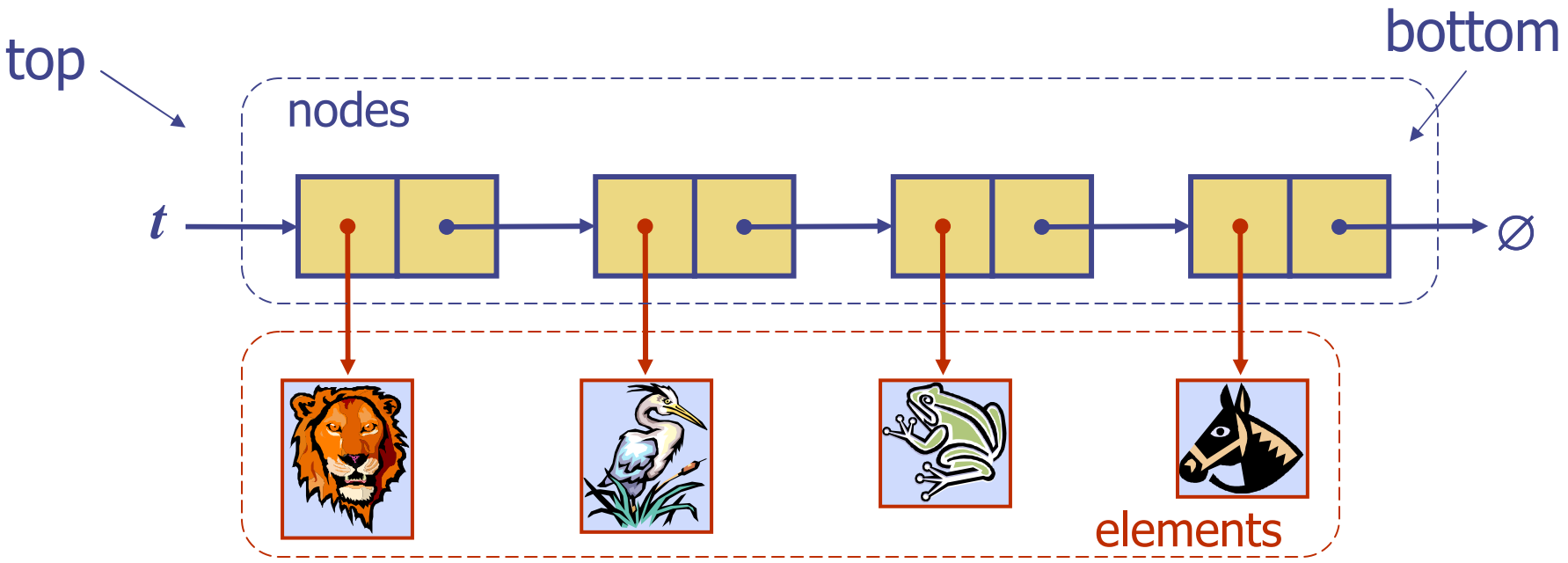
```
template <typename Object>
class ArrayStack {
private:
    int capacity;      // stack capacity
    Object *S;         // stack array
    int t;             // top of stack
public:
    ArrayStack(int c) {
        capacity = c;
        S = new Object[capacity];
        t =  –1;
    }
```

```
    bool isEmpty()
      {  return (t < 0);  }

    Object pop()
        throw(EmptyStackException) {
        if(isEmpty())
            throw EmptyStackException
                ("Access to empty stack");
        return S[t--];
    }
// … (other functions omitted)
```

Stack with a Singly Linked List

- We can implement a stack with a singly linked list
- The front 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: ( )(( )){([( )])}
  - incorrect: ((( )(( )){([( )])}
  - 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>

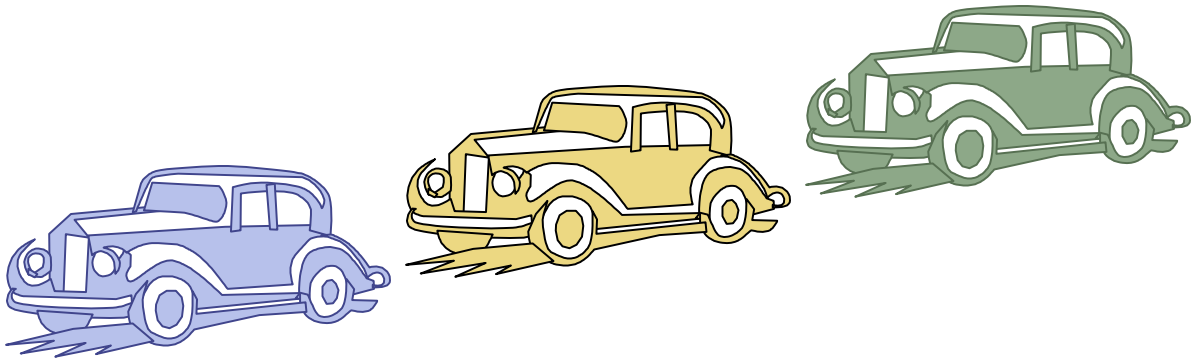
```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little boat like a
cheap sneaker in an old washing machine. The
three drunken fishermen were used to such
treatment, of course, but not the tree
salesman, who even as a stowaway now felt
that he had overpaid for the voyage.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

Queues



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

The Queue ADT (cont.)

- Main queue operations:
  - enqueue(o): inserts element o at the end of the queue
  - dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - front(): returns the element at the front without removing it
  - size(): returns the number of elements stored
  - isEmpty(): returns a Boolean value indicating whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

Queue Example

Operation	output	queue
• 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)
• size()	1	(9)

Informal C++ Queue Interface

- Informal C++ interface for our Queue ADT
- Requires the definition of class EmptyQueueException
- Corresponding built-in STL class: queue

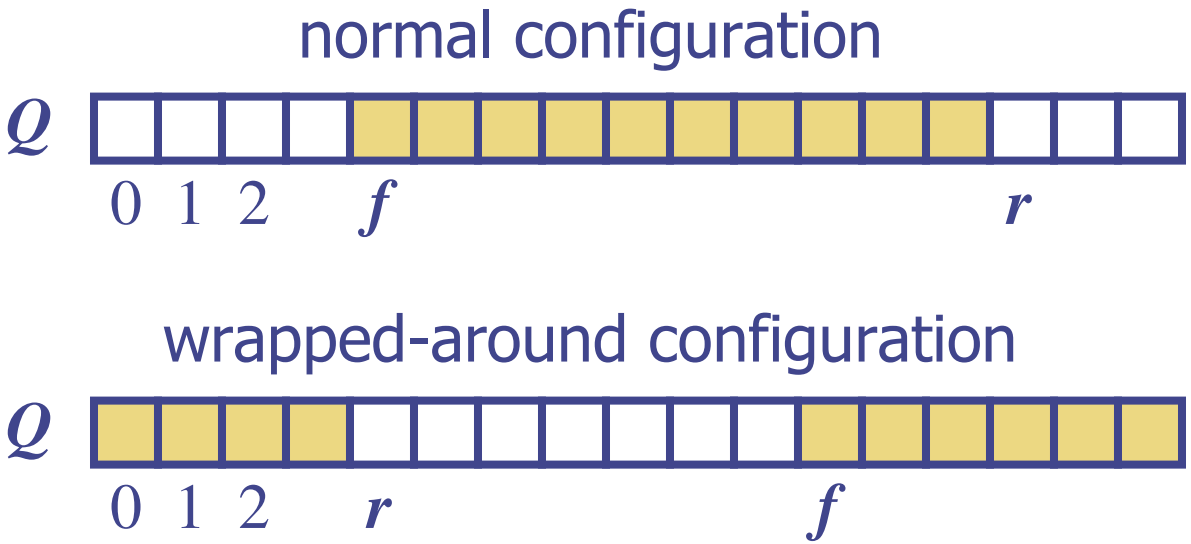
```
template <typename Object>
class Queue {
public:
    int size();
    bool isEmpty();
    Object& front()
        throw(EmptyQueueException);
    void enqueue(Object o);
    Object dequeue()
        throw(EmptyQueueException);
};
```

Applications of Queues

- Direct applications
  - Waiting lists
  - 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

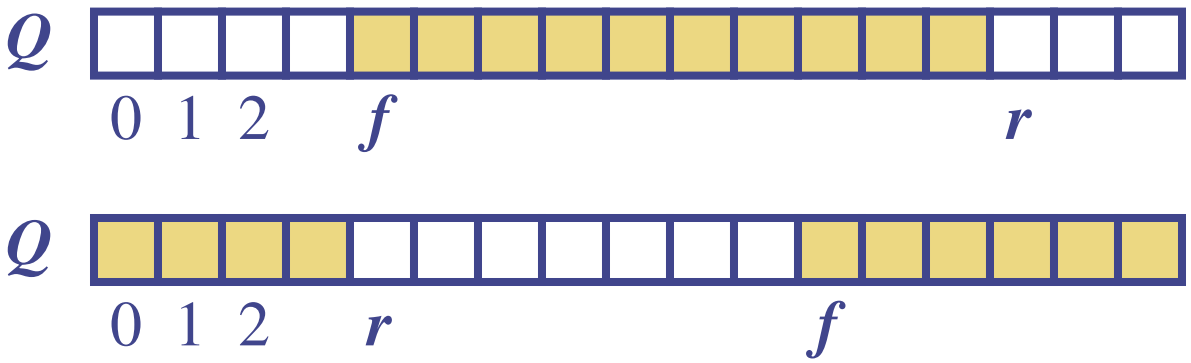


Queue Operations

- We use the modulo operator (remainder of division)

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

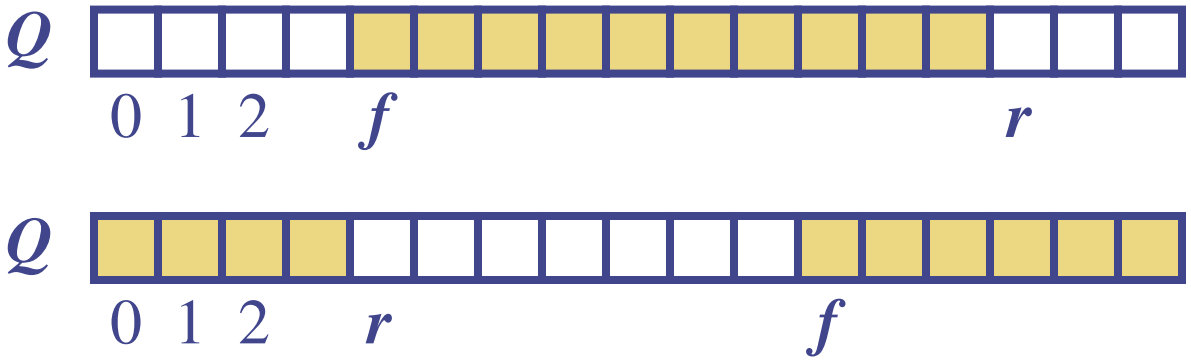
Algorithm *isEmpty*()
    return $(f = r)$



Queue Operations (cont.)

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

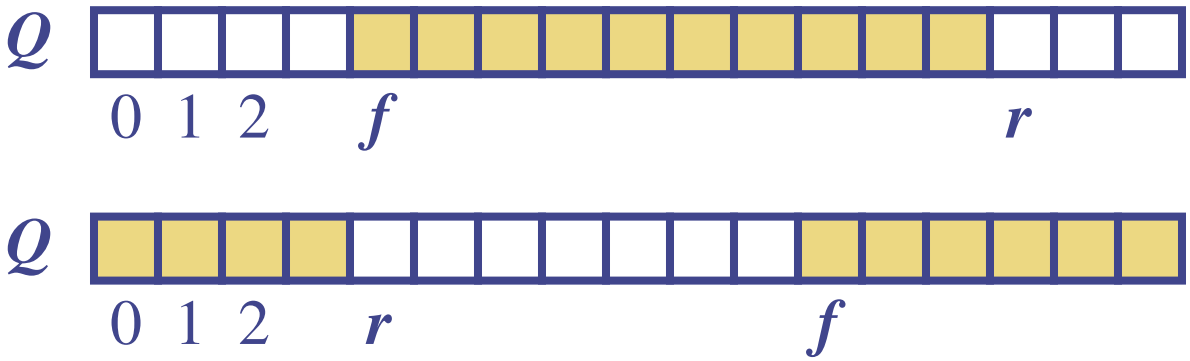
```
Algorithm enqueue(o)
    if size() = N - 1 then
        throw FullQueueException
    else
        Q[r] ← o
        r ← (r + 1) mod N
```



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
```

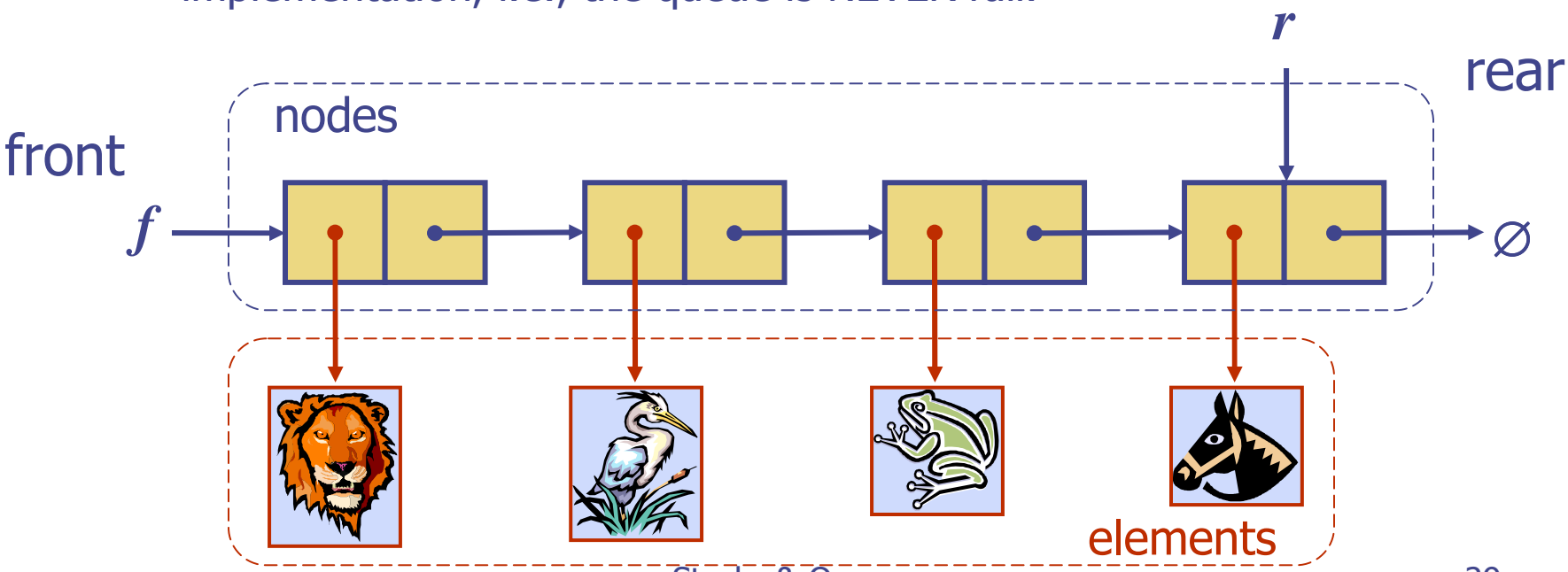


Performance and Limitations
- array-based implementation of queue ADT

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

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
- NOTE: we do not have the size-limitation of the array based implementation, i.e., the queue is NEVER full.



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*)