CH6.
STACKS, QUEUES, AND DEQUES

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)

ABSTRACT DATA TYPES (ADTS)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - `order buy(stock, shares, price)`
    - `order sell(stock, shares, price)`
    - `void cancel(order)`
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

STACKS (CH 6.1)

STACKS

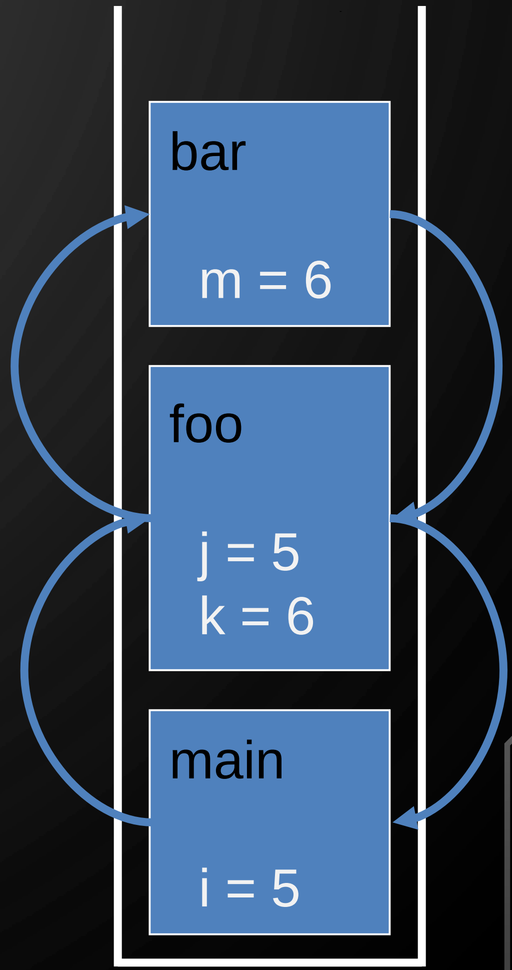
- A data structure similar to a neat stack of something, basically only access to top element is allowed – also reffered to as LIFO (last-in, first-out) storage
- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

EXAMPLE STACK

METHOD STACK IN THE JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a methods is called, the JVM pushes on the stack a frame containing local variables and return value
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack

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

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

bar(int m) {
   …
}
```


bar

m = 6

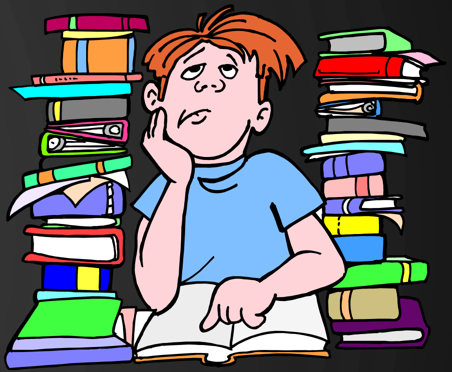
foo

j = 5
k = 6

main

i = 5

THE STACK ADT



- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out (LIFO) scheme
- Main stack operations:
  - `push(e)`: inserts element e at the top of the stack
  - `Element pop()`: removes and returns the top element of the stack (last inserted element)
- Auxiliary stack operations:
  - `Element top()`: returns reference to the top element without removing it
  - `Integer size()`: returns the number of elements in the stack
  - `Boolean isEmpty()`: a Boolean value indicating whether the stack is empty
- Attempting the execution of `pop` or `top` on an empty stack return `null`

EXAMPLE STACK INTERFACE IN JAVA

```
public interface Stack<E> {
  int size();          /** Return number of elements */
  boolean isEmpty(); /** True is size is 0 */
  E top();             /** Return visible stack element */
  void push(E e);      /** Add to the stack */
  E pop();             /** Remove from the stack */
}
```

EXERCISE: STACKS

- Illustrate the following operations starting with an empty stack
  - `push(8)`
  - `push(3)`
  - `pop()`
  - `push(2)`
  - `push(5)`
  - `pop()`
  - `pop()`
  - `push(9)`
  - `push(1)`

WHAT HAPPENS WHEN AN OPERATION IS INVALID? LIKE POP ON AN EMPTY STACK.

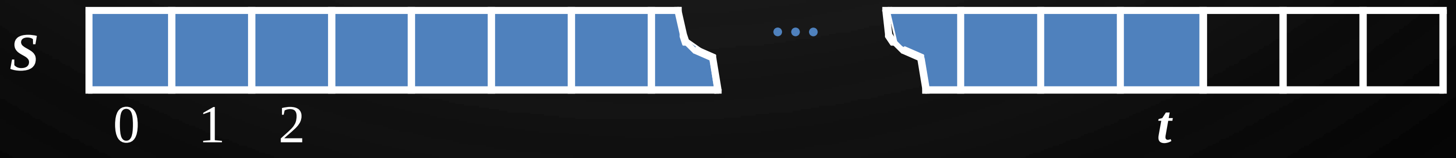
- It is a design decision!
- Attempting the execution of an operation of an ADT may sometimes cause an error condition
- Java supports a general abstraction for errors, called exceptions. Exceptions are said to be thrown by an operation that cannot be properly executed

- In our Stack ADT, we did not use exceptions
- Instead, we allow operations pop and top to be performed even if the stack is empty by returning `null`

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

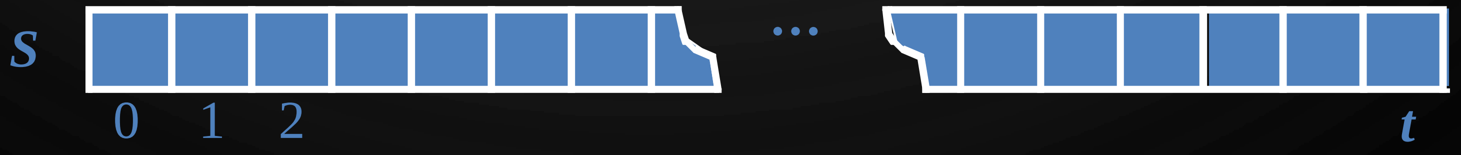
```
Algorithm size()
Output: Number of elements
return $t + 1$

Algorithm pop()
Output: Removed element
if isEmpty() then
    return null
$t \leftarrow t - 1$
return $S[t + 1]$
```



ARRAY-BASED STACK

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

```
Algorithm push(e)
Input: Element e
1. if $t = S.length - 1$ then
2.     throw IllegalStateException
3. $t \leftarrow t + 1$
4. $S[t] \leftarrow e$
```



JAVA IMPLEMENTATION

```java
public class ArrayStack<E> implements Stack<E> {
  private E[] stack;
  private int t = -1;

  public ArrayStack() {this(10);}

  public ArrayStack(int capacity) {
    stack = (E[]) new Object[capacity];
  }

  public int size() {return t + 1;}

  public boolean isEmpty() {return t == -1;}

  public E top() {
    if(isEmpty()) return null;
    return stack[t];
  }

  public void push(E e)
        throws IllegalStateException {
    if(size() == stack.length)
      throw IllegalStateException(
        "Stack is full");
    stack[++t] = e;
  }

  public E pop() {
    if(isEmpty()) return null;
    E e = stack[t];
    stack[t--] = null;
    return e;
  }
}
```

PERFORMANCE AND LIMITATIONS

ARRAY-BASED IMPLEMENTATION

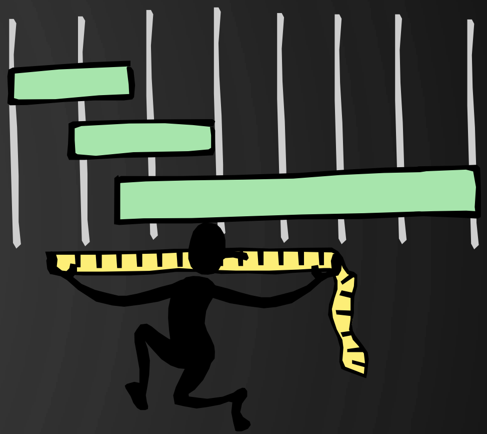
- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined *a priori*, and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific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

GROWABLE ARRAY-BASED STACK

- In a push 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 push(e)
Input: Element $e$
if $t = S.length - 1$ then
    $A \leftarrow$ new array of size ?
    for $i \leftarrow 0$ to $t$ do
        $A[i] \leftarrow S[i]$
    $S \leftarrow A$
$S[t] \leftarrow e$
$t \leftarrow t + 1$
```

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 push operations
- We assume that we start with an empty stack represented
- We then use amortized analysis to determine cost of a single operation. For this kind of operation, the amortized time will be the average time taken by a push over the series of operations, i.e., $T(n)/n$

AMORTIZATION MEANS PAYING OFF AN AMOUNT OVER TIME, NOT AVERAGE. THIS ANALYSIS RECOGNIZES EACH OPERATION IS INHERENTLY UNEQUAL. IT IS A STANDARD COMPUTER SCIENCE METHOD TO DEAL WITH THIS.

INCREMENTAL STRATEGY ANALYSIS

- Let c be the constant increase and n be the number of push operations
- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n push operations is proportional to

$$
\begin{aligned}
& n + c + 2c + 3c + 4c + \ldots + kc \\
&= n + c(1 + 2 + 3 + \ldots + k) \\
&= n + c\frac{k(k+1)}{2} \\
&= O(n + k^2) = O\left(n + \frac{n^2}{c^2}\right) = O(n^2)
\end{aligned}
$$

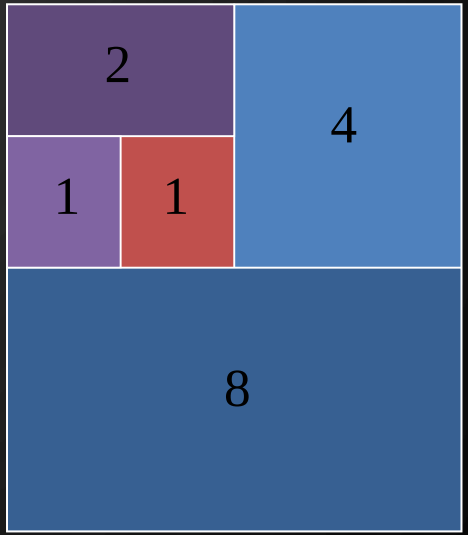
Side note:

$$
\begin{aligned}
& 1 + 2 + \cdots + k \\
&= \sum_{i=0}^{k} i \\
&= \frac{k(k+1)}{2}
\end{aligned}
$$

- $T(n)$ is $O(n^2)$ so the average time of a push is $\frac{\mathrm{O}(\mathrm{n}^2)}{\mathrm{n}} = O(n)$

DOUBLING STRATEGY ANALYSIS

- We will use an amortized technique here based on accounting. Performing a push operation will "cost" cyber-dollars (imaginary cost of the operation). Then we sum the total amount of cyber dollars to determine run-time
- Lets assume push costs 1 cyber dollar. Also assume growing an array from size k to size $2k$ requires k cyber-dollars (for k copies)
- So we can "charge" (overcharge) each push to be 3 cyber-dollars
  - One for their push into the array
  - One for their copy on resizing
  - One for one of the first $\frac{k}{2}$ elements copy on resizing
- Put another way, a resize occurs when there are 2^i elements. Doubling the size of the array costs 2^i cyber-dollars, which are found in cells 2^{i-1} through $2^i - 1$
- Thus all of the computation is paid for the the amortization is valid
- We pay $3n$ for n push operations which takes $O(n)$ time and one push is $\frac{O(n)}{n} = O(1)$ amortized time

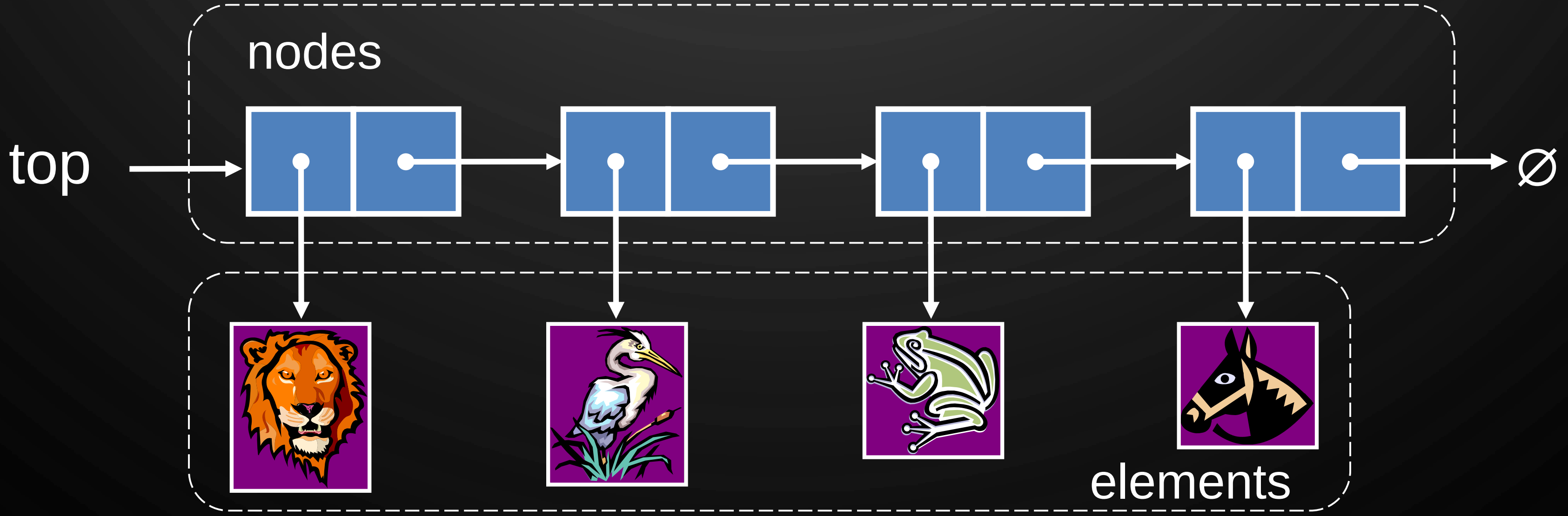


EXERCISE
STACK WITH A SINGLY-LINKED LIST

- Describe how to implement a stack using a singly-linked list
  - Stack operations: `push(`e`)`, `pop()`, `size()`, `isEmpty()`
  - For each operation, give the running time
  - What are the downsides of such an implementation?

EXERCISE
STACK WITH A SINGLY-LINKED LIST

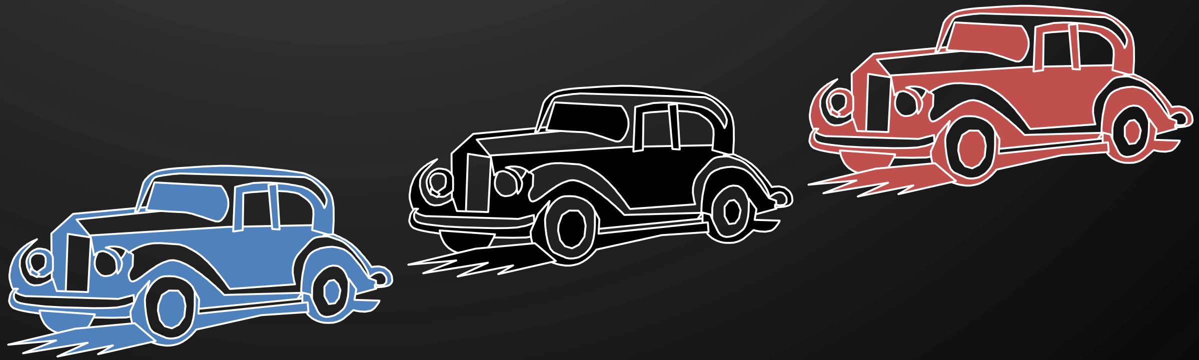
- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



STACK SUMMARY

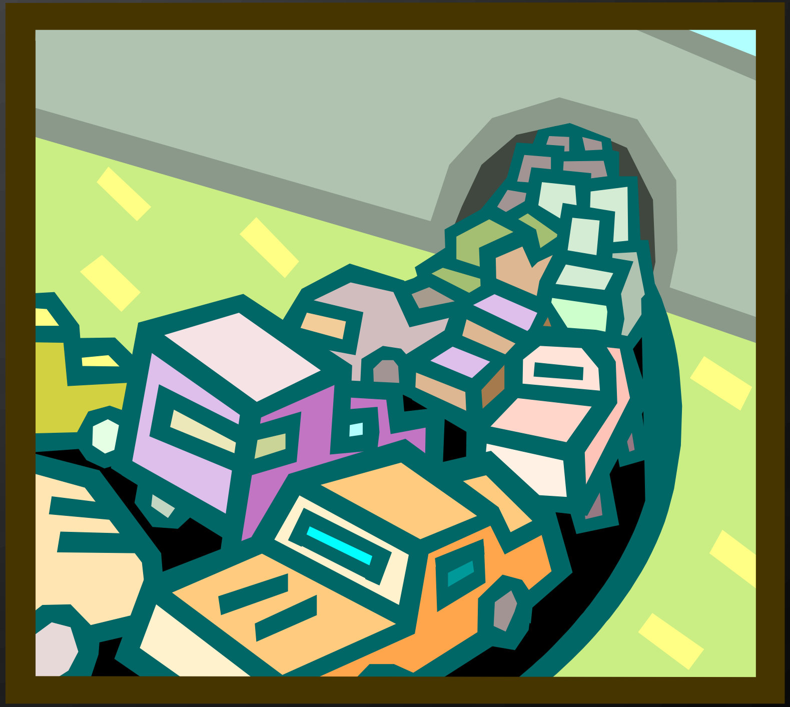
	Array Fixed-Size	Array Expandable (doubling strategy)	List Singly-Linked
`pop()`	$O(1)$	$O(1)$	$O(1)$
`push(o)`	$O(1)$	$O(n)$ Worst Case $O(1)$ Best Case $O(1)$ Average Case	$O(1)$
`top()`	$O(1)$	$O(1)$	$O(1)$
`size()`, `empty()`	$O(1)$	$O(1)$	$O(1)$

QUEUES (CH 6.2)



APPLICATIONS OF QUEUES

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



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(e)`: inserts element *e* at the end of the queue
  - `Element dequeue()`: removes and returns the element at the front of the queue

- Auxiliary queue operations:
  - `Element first()`: 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
- Boundary cases
  - Attempting the execution of `dequeue` or `first` on an empty queue returns **`null`**

EXAMPLE QUEUE INTERFACE IN JAVA

```java
public interface Queue<E> {
  int size();          /** Return number of elements */
  boolean isEmpty(); /** True is size is 0 */
  E first();           /** Return visible queue element */
  void enqueue(E e); /** Add to the queue */
  E dequeue();         /** Remove from the queue */
}
```

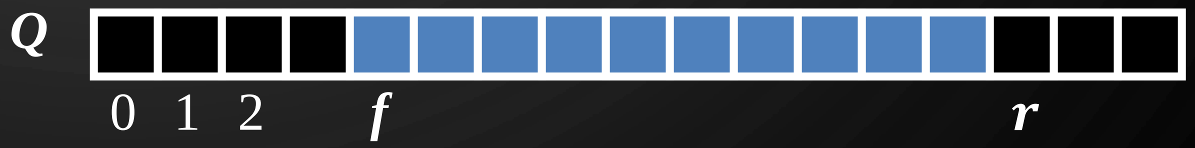
EXERCISE: QUEUES

- Illustrate the following operations starting with an empty queue
  - `enqueue(8)`
  - `enqueue(3)`
  - `dequeue()`
  - `enqueue(2)`
  - `enqueue(5)`
  - `dequeue()`
  - `dequeue()`
  - `enqueue(9)`
  - `enqueue(1)`

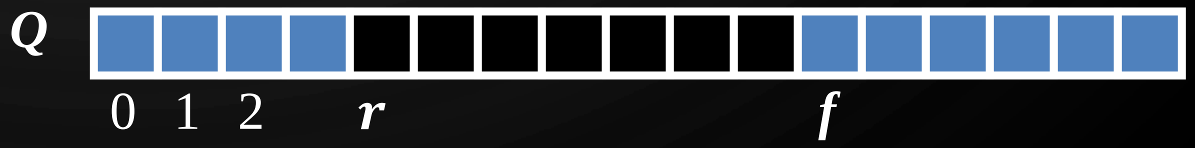
ARRAY-BASED QUEUE

- Use an array of size N in a circular fashion, i.e., a circular array
- Two variables keep track of the front and rear
  - f index of the front element
  - sz number of stored elements
- When the queue has fewer than N elements, array location $r \leftarrow (f + sz) \bmod N$ is the first empty slot past the rear of the queue

normal configuration



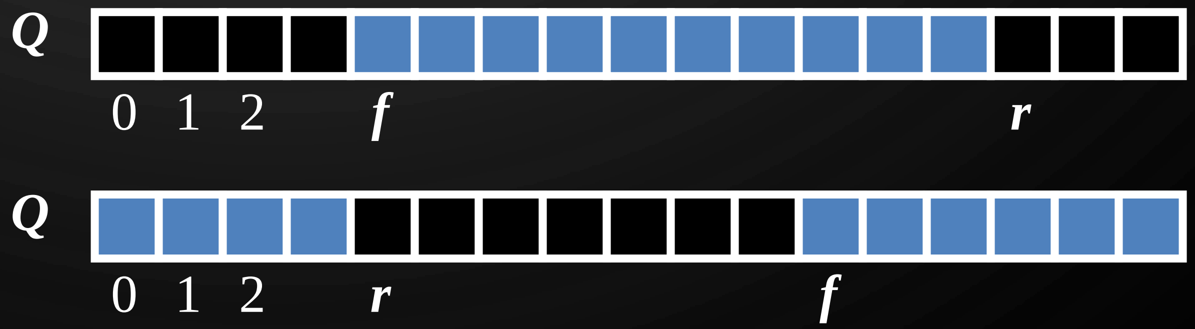
wrapped-around configuration



QUEUE OPERATIONS

- We use the modulo operator (remainder of division)

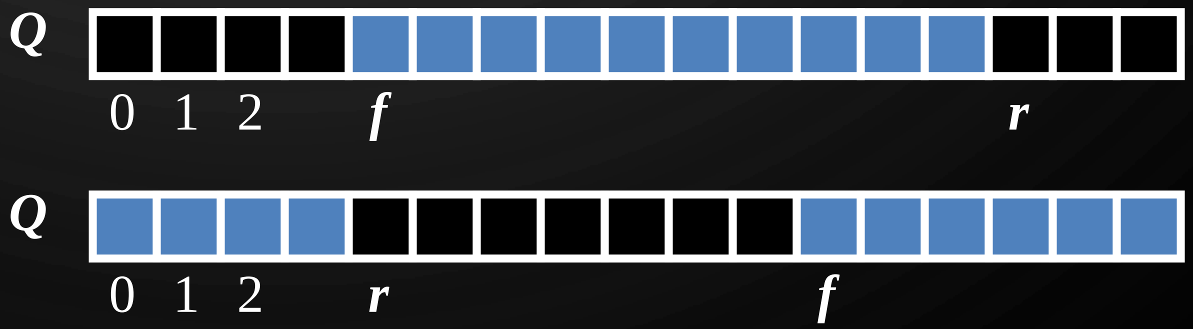
```
Algorithm size()
Output: Number of elements
return $sz$
Algorithm isEmpty()
Output: True if no elements in queue, false otherwise
return $sz = 0$
```



QUEUE OPERATIONS

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

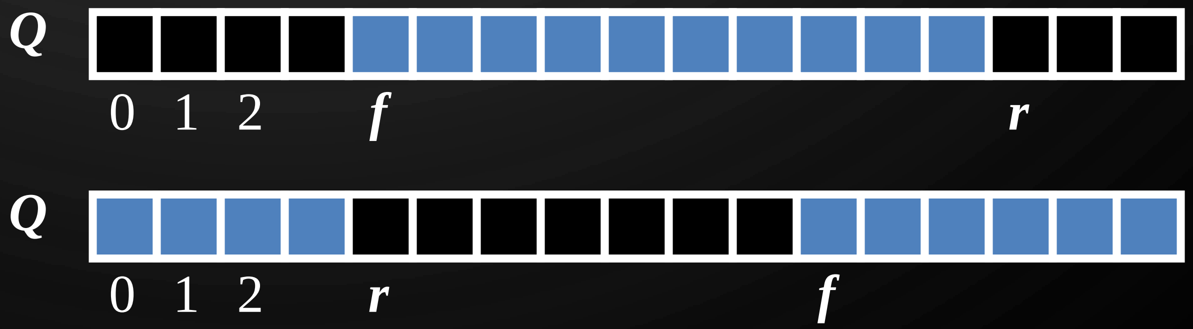
```
Algorithm enqueue(e)
Input: Element e
if $\text{size}() = N - 1$ then
    throw IllegalStateException
$r \leftarrow (f + sz) \bmod N$
$Q[r] \leftarrow e$
$sz \leftarrow sz + 1$
```



QUEUE OPERATIONS

- Operation dequeue returns null if the queue is empty

```
Algorithm dequeue()
Output: Removed element
if isEmpty() then
    return null
Element e ← $Q[f]$
$f \leftarrow f + 1 \bmod N$
$sz \leftarrow sz - 1$
return e
```



JAVA IMPLEMENTATION

```java
public class ArrayQueue<E> implements Queue<E> {
  private E[] queue;
  private int f = 0, sz = 0;

  public ArrayQueue() {this(10);}

  public ArrayQueue(int capacity) {
    queue = (E[]) new Object[capacity];
  }

  public int size() {return sz;}

  public boolean isEmpty() {return sz == 0;}

  public E first() {
    if(isEmpty()) return null;
    return queue[f];
  }
```

```java
  public void enqueue(E e)
      throws IllegalStateException {
    if(size() == data.length)
      throw IllegalStateException(
        "Queue is full");
    stack[(f+sz)%queue.length] = e;
    ++sz;
  }

  public E dequeue() {
    if(isEmpty()) return null;
    E e = queue[f];
    queue[f] = null;
    f = (f+1)%queue.length;
    --sz;
    return e;
  }
}
```

PERFORMANCE AND LIMITATIONS
ARRAY-BASED IMPLEMENTATION

- 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

GROWABLE ARRAY-BASED QUEUE

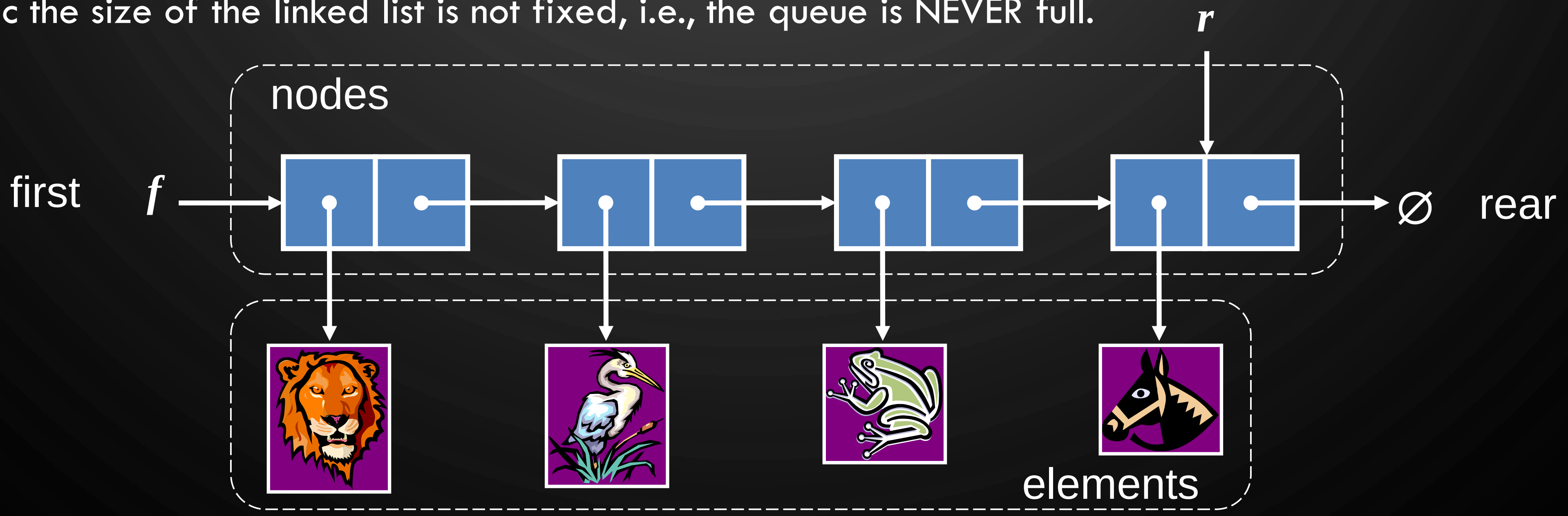
- In `enqueue`(e), when the array is full, instead of throwing an exception, we can replace the array with a larger one
- Similar to what we did for an array-based stack
- `enqueue`(e) has amortized running time
  - $O(n)$ with the incremental strategy
  - $O(1)$ with the doubling strategy

EXERCISE
QUEUE WITH A SINGLY-LINKED LIST

- Describe how to implement a queue using a singly-linked list
  - Queue operations: `enqueue(`e`)`, `dequeue()`, `size()`, `empty()`
  - For each operation, give the running time

EXERCISE
QUEUE WITH A SINGLY-LINKED LIST

- The first element is stored at the head of the list, The rear element is stored at the tail of the list
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time
- NOTE: we do not have the limitation of the array based implementation on the size of the stack b/c the size of the linked list is not fixed, i.e., the queue is NEVER full.



QUEUE SUMMARY

	Array Fixed-Size	Array Expandable (doubling strategy)	List Singly-Linked
`dequeue()`	$O(1)$	$O(1)$	$O(1)$
`enqueue(e)`	$O(1)$	$O(n)$ Worst Case $O(1)$ Best Case $O(1)$ Average Case	$O(1)$
`first()`	$O(1)$	$O(1)$	$O(1)$

`size()`
`empty()`
	$O(1)$	$O(1)$	$O(1)$

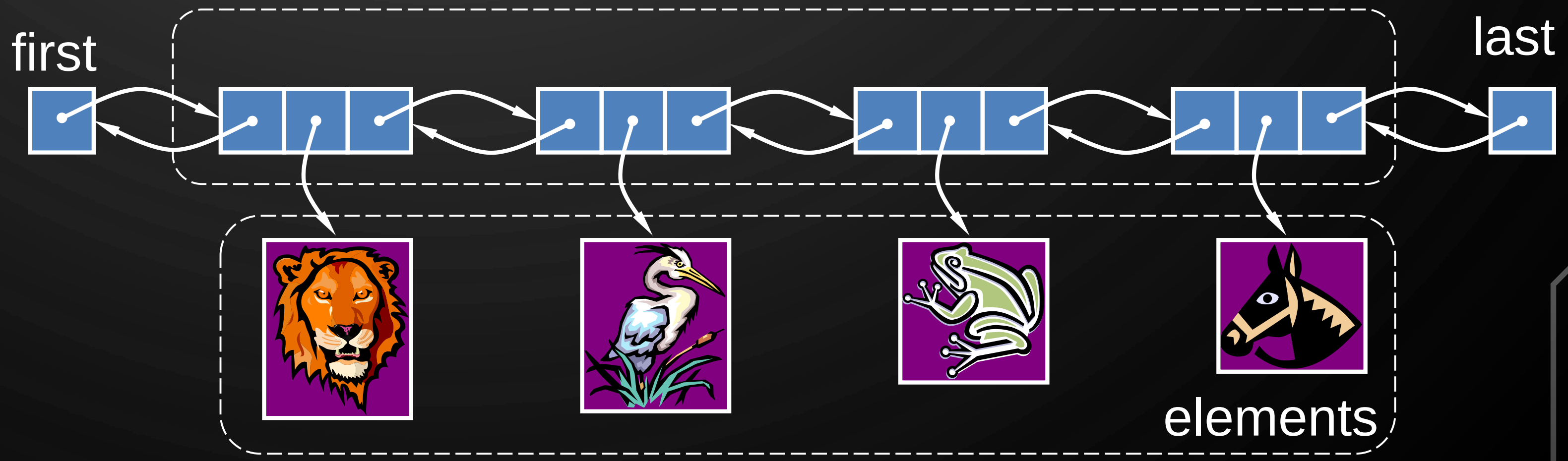
THE DOUBLE-ENDED QUEUE ADT (CH. 6.3)

- The Double-Ended Queue, or Deque, ADT stores arbitrary objects. (Pronounced 'deck')
- Richer than stack or queue ADTs. Supports insertions and deletions at both the front and the end.
- Main deque operations:
  - `addFirst(e)`: inserts element *e* at the beginning of the deque
  - `addLast(e)`: inserts element *e* at the end of the deque
  - `Element removeFirst()`: removes and returns the element at the front of the queue
  - `Element removeLast()`: removes and returns the element at the end of the queue

- Auxiliary queue operations:
  - `Element first()`: returns the element at the front without removing it
  - `Element last()`: 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

DEQUE WITH A DOUBLY 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 Deque ADT takes $O(1)$ time



PERFORMANCE AND LIMITATIONS
DOUBLY LINKED LIST IMPLEMENTATION

- Performance
  - Let n be the number of elements in the deque
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$

DEQUE SUMMARY

	Array Fixed-Size	Array Expandable (doubling strategy)	List Singly-Linked	List Doubly-Linked

`removeFirst()`
`removeLast()`
	$O(1)$	$O(1)$	$O(n)$ for one at list tail,
			$O(1)$ for other
				$O(1)$

`addFirst(o)`
`addLast(o)`
	$O(1)$	$O(n)$ Worst Case
		$O(1)$ Best Case
		$O(1)$ Average Case
			$O(1)$	$O(1)$

`first()`
`last()`
	$O(1)$	$O(1)$	$O(1)$	$O(1)$

`size()`
`isEmpty()`
	$O(1)$	$O(1)$	$O(1)$	$O(1)$

INTERVIEW QUESTION 1

- How would you design a stack which, in addition to push and pop, also has a function `min` which returns the minimum element? `push`, `pop` and `min` should all operate in $O(1)$ time

GAYLE LAAKMANN MCDOWELL, "CRACKING THE CODE INTERVIEW: 150 PROGRAMMING QUESTIONS AND SOLUTIONS", 5TH EDITION, CAREERCUP PUBLISHING, 2011.

INTERVIEW QUESTION 2

- An animal shelter holds only dogs and cats, and operates in a strictly "first in, first out" basis. People must adopt either the "oldest" (based on arrival time) of all animals at the shelter, or they can select whether they prefer a dog or cat (and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structure(s) to maintain this system and implement operations such as enqueue, dequeueAny, dequeueDog, and dequeueCat.

GAYLE LAAKMANN MCDOWELL, "CRACKING THE CODE INTERVIEW: 150 PROGRAMMING QUESTIONS AND SOLUTIONS", 5TH EDITION, CAREERCUP PUBLISHING, 2011.