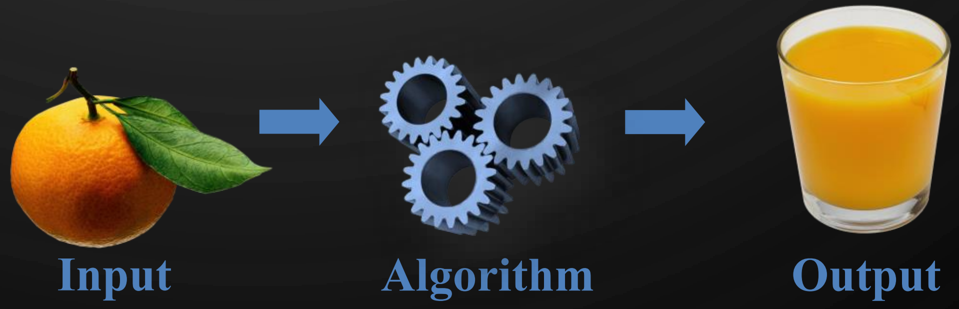
CH4.2-4.3.
ALGORITHM ANALYSIS
CH6.
STACKS, QUEUES, AND DEQUES

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

ANALYSIS OF ALGORITHMS (CH 4.2-4.3)



PSEUDOCODE

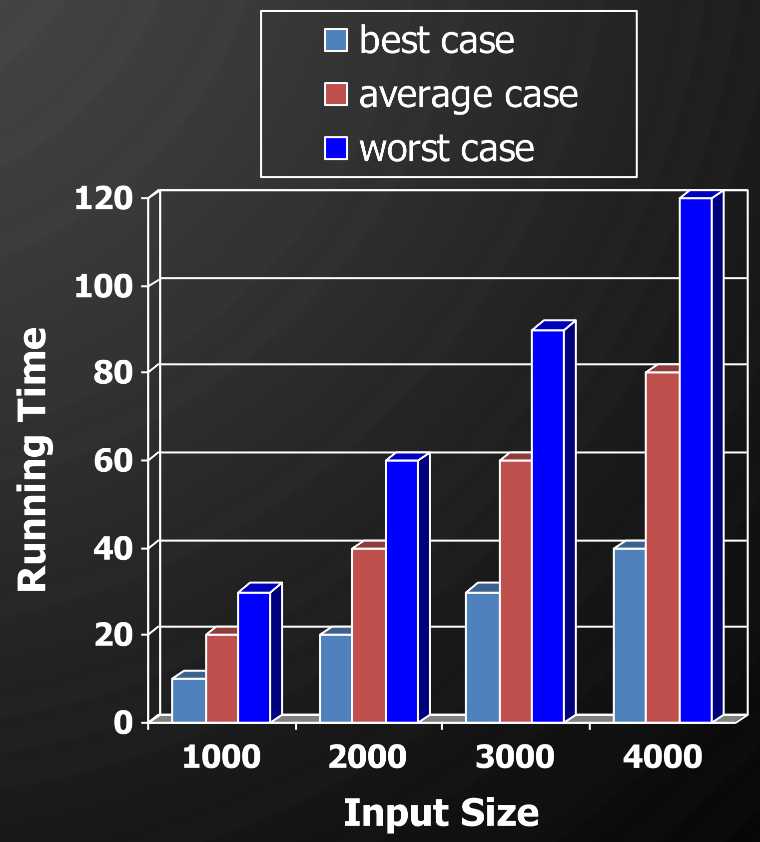
- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

PSEUDOCODE DETAILS

- Control flow
  - if … then … [else …]
  - while … do …
  - repeat … until …
  - for … do …
  - Indentation replaces braces
- Method declaration
  - Algorithm method (arg [, arg…])
  - Input …
  - Output …
- Method call
  - method (arg [, arg…])
- Return value
  - return expression
- Expressions:
  - Assignment (←, not =)
  - Equality testing (= not ==)
  - n^2 Superscripts and other mathematical formatting allowed

RUNNING TIME

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



LIMITATIONS OF EXPERIMENTS

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

THEORETICAL ANALYSIS

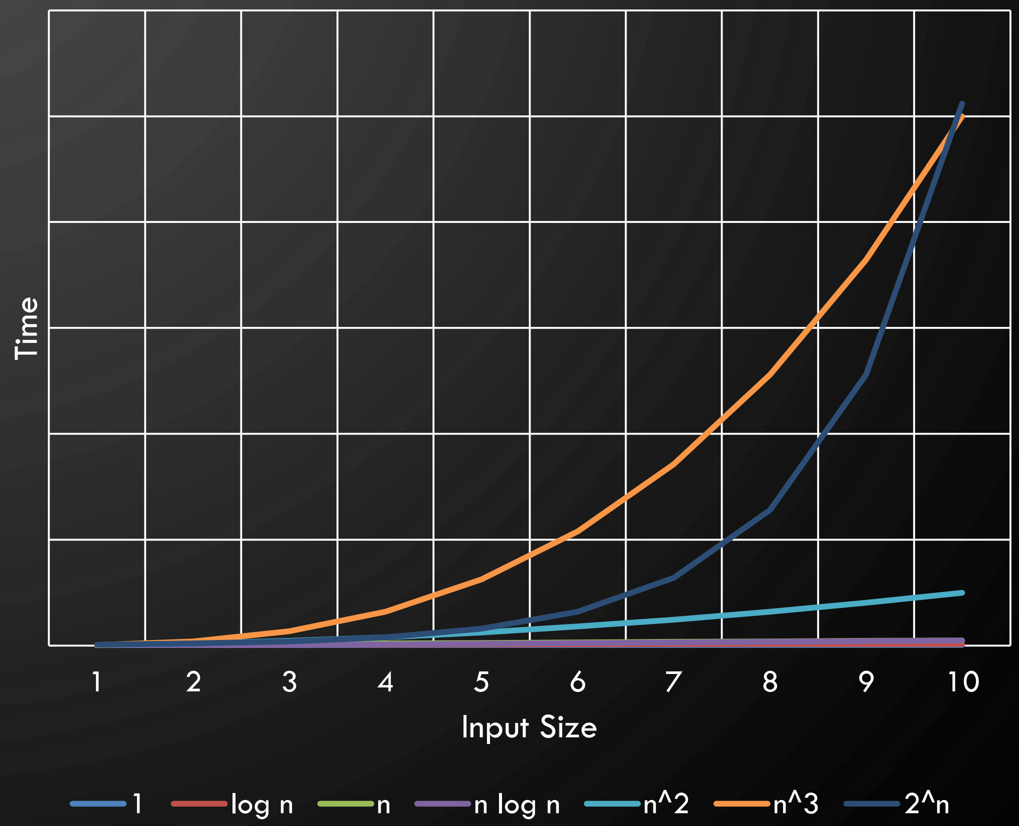
- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n (Big-Oh notation)
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment
- How
  - Count the operations!

BIG-OH NOTATION

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O\big(g(n)\big)$ if there are positive constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$
  - $f(n)$ - real computation time (measured time if you will)
  - $g(n)$ - approximation function
- Example: 2n + 10 is O(n)
  - $2n + 10 \leq cn$
  - $\frac{10}{c-2} \leq n$
  - Pick $c = 3$ and $n_0 = 10$
- Easy method: Strip constants, and take highest order terms only!
  - Constants do no matter because of limits as n goes to infinity

SEVEN IMPORTANT FUNCTIONS

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic ≈ $\log n$
  - Linear ≈ n
  - N-Log-N ≈ $n \log n$
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n
- In a log-log chart, the slope of the line corresponds to the growth rate



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

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
  - `object pop()`: removes and returns the top element of the stack (last inserted element)
- Auxiliary stack operations:
  - `object 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`

EXERCISE: STACKS

- Describe the output of the following series of stack operations
  - `Push(8)`
  - `Push(3)`
  - `Pop()`
  - `Push(2)`
  - `Push(5)`
  - `Pop()`
  - `Pop()`
  - `Push(9)`
  - `Push(1)`

EXCEPTIONS VS. RETURNING NULL

- Attempting the execution of an operation of an ADT may sometimes cause an error condition
- Java supports a general abstraction for errors, called exception
- An exception is said to be thrown by an operation that cannot be properly executed

- In our Stack ADT, we do not use exceptions
- Instead, we allow operations pop and top to be performed even if the stack is empty
- For an empty stack, pop and top simply return null

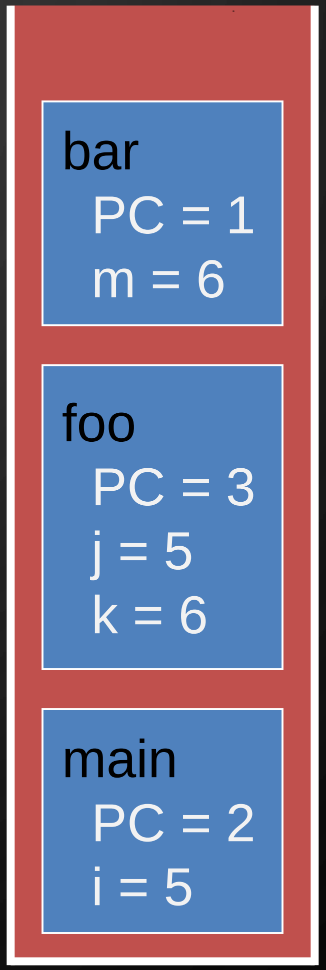
METHOD STACK IN THE JVM

- 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
  - 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

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

foo(int j) {
   int 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

<u>size()</u>

1. **return** $t + 1$

<u>pop()</u>

1. **if** isEmpty() **then**
2. **return** null
3. $t \leftarrow t - 1$
4. **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

<u>push(o)</u>

1. **if** $t = S.length - 1$ **then**
2.   **throw** IllegalStateException
3. $t \leftarrow t + 1$
4. $S[t] \leftarrow o$

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

GROWABLE ARRAY-BASED STACK

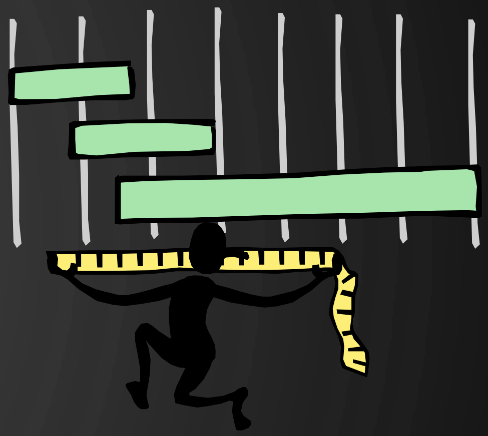
- In a push operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- How large should the new array be?
  - incremental strategy: increase the size by a constant c
  - doubling strategy: double the size

push

Input: Element o

if $t = S.length - 1$ **then**
    $A \leftarrow$ new array of size …
    for $i \leftarrow 0$ to t **do**
        $A[i] \leftarrow S[i]$
    $S \leftarrow A$
$S[t] \leftarrow o$
$t \leftarrow t + 1$

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 call amortized time of a push operation the average time taken by a push over the series of operations, i.e., $T(n)/n$

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

$$n + c + 2c + 3c + 4c + \dots + kc$$
$$= n + c(1 + 2 + 3 + \dots + k)$$
$$= n + c\frac{k(k+1)}{2}$$
$$= O(n + k^2) = O\left(n + \frac{n^2}{c^2}\right) = O(n^2)$$

Side note:

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

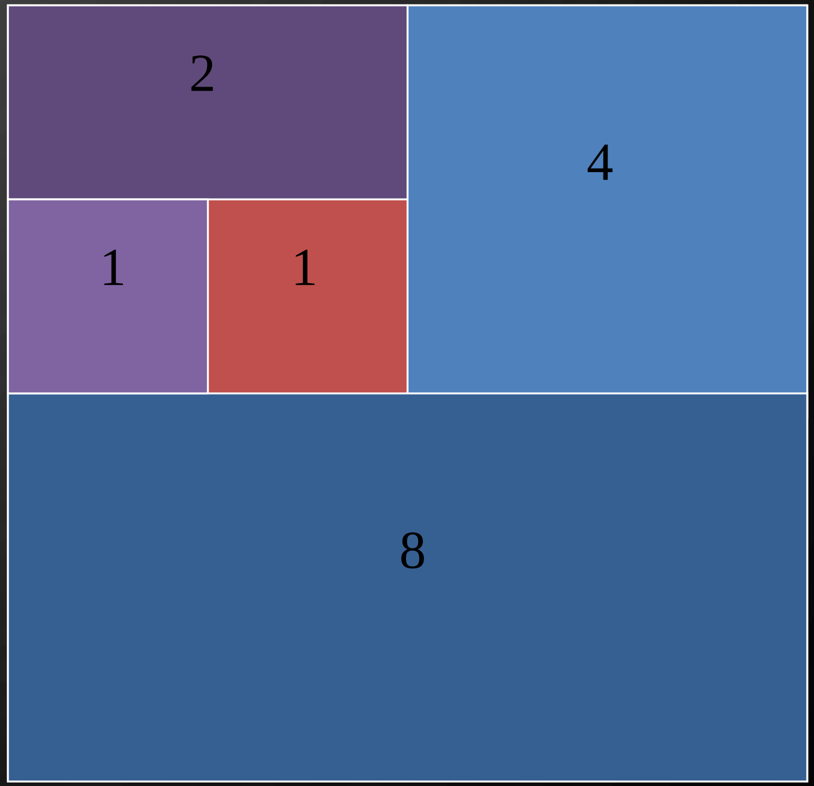
- $T(n)$ is $O(n^2)$ so the amortized time of a push is $\frac{\mathrm{O(n^2)}}{\mathrm{n}} = O(n)$

DOUBLING STRATEGY ANALYSIS

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of $\boldsymbol{n}$ push operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k$$
$$= n + 2^{k+1} - 1$$
$$= O\left(n + 2^k\right) = O\left(n + 2^{\log_2 n}\right) = O(n)$$

- $T(n)$ is $O(n)$ so the amortized time of a push is $\frac{\mathrm{O(n)}}{\mathrm{n}} = O(1)$

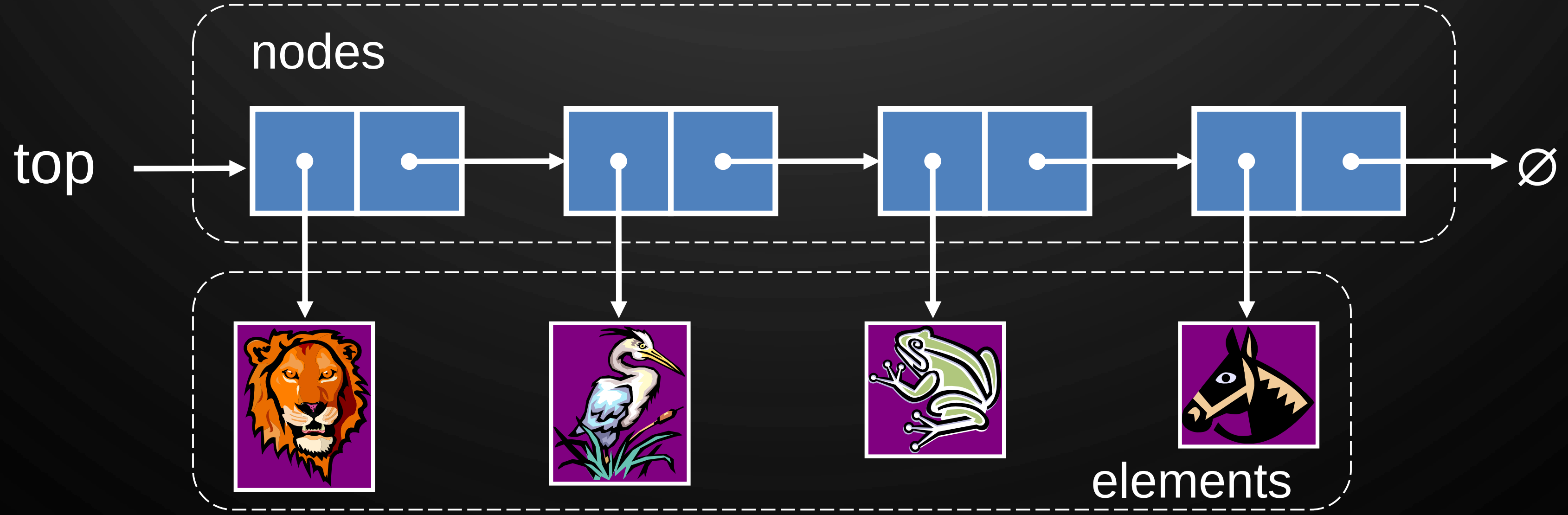


EXERCISE

- 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

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



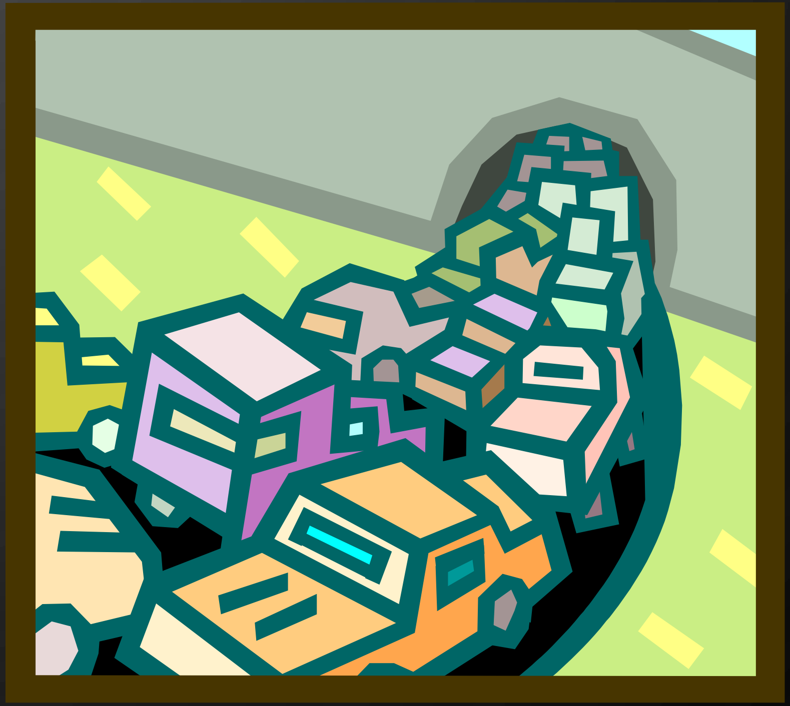
STACK SUMMARY

	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

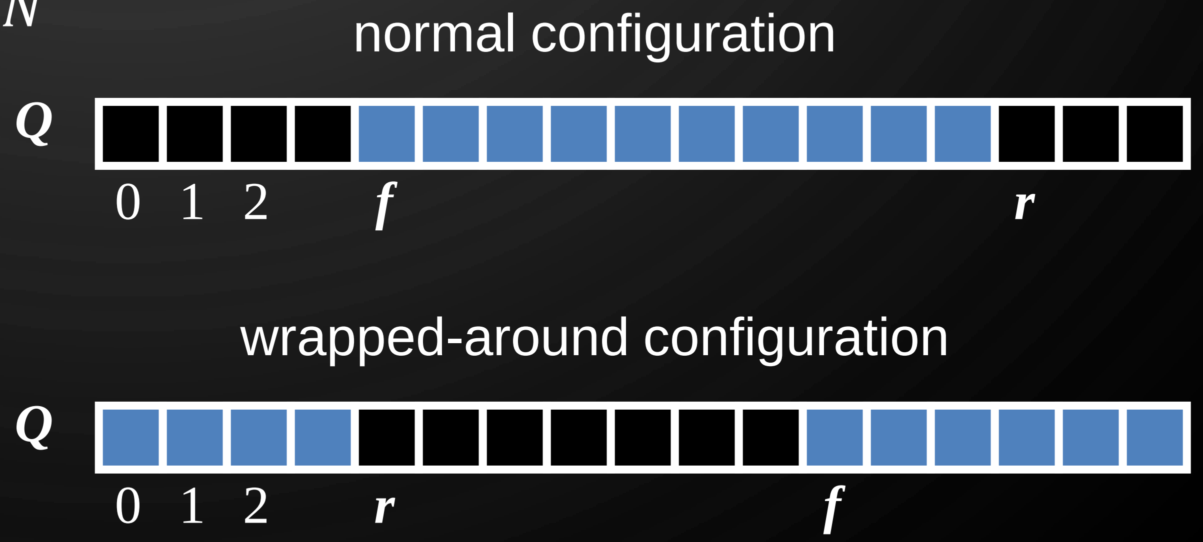
- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out (FIFO) scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - `enqueue(e)`: inserts element *e* at the end of the queue
  - `object dequeue()`: removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - `object 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 front on an empty queue returns null

EXERCISE: QUEUES

- Describe the output of the following series of queue operations
  - `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
- 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



QUEUE OPERATIONS

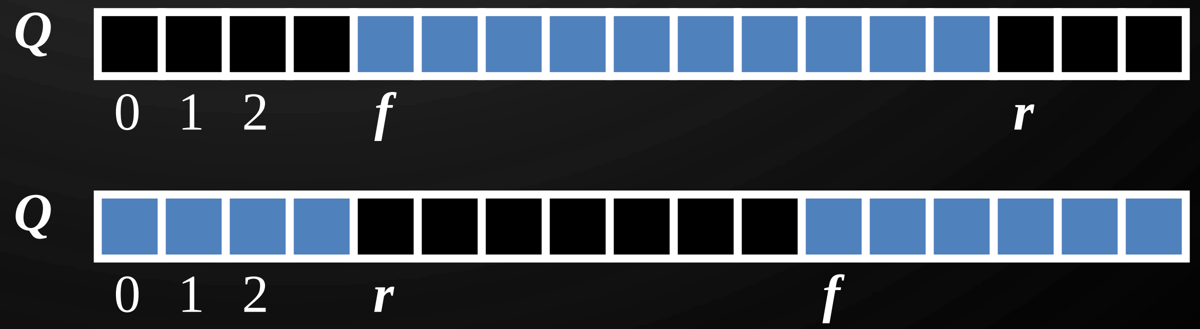
- We use the modulo operator (remainder of division)

<u>`size()`</u>

1. **return** sz

<u>`isEmpty()`</u>

1. **return** $sz = 0$



QUEUE OPERATIONS

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

`enqueue(o)`

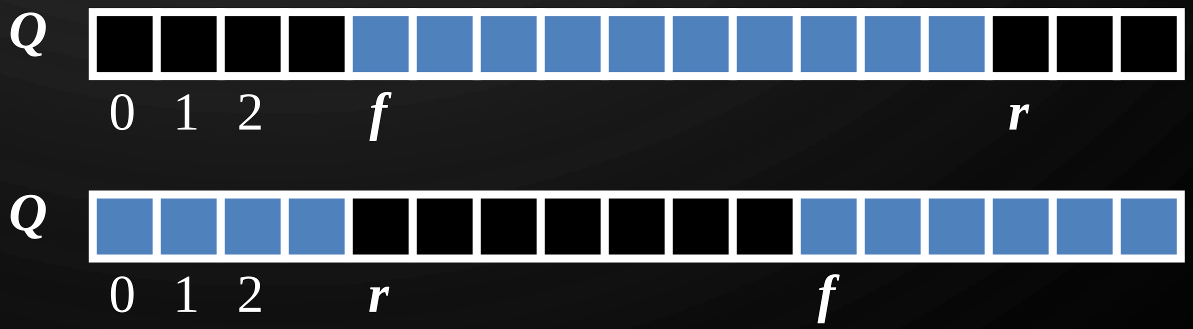
1. **if** $\text{size}() = N - 1$ **then**
2. **throw** IllegalStateException
3. $r \leftarrow (f + sz) \bmod N$
4. $Q[r] \leftarrow o$
5. $sz \leftarrow sz + 1$

QUEUE OPERATIONS

- Operation dequeue returns null if the queue is empty

<u>dequeue()</u>

```
if empty() then
    return null
o ← Q[f]
f ← f + 1 mod N
sz ← sz − 1
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

GROWABLE ARRAY-BASED QUEUE

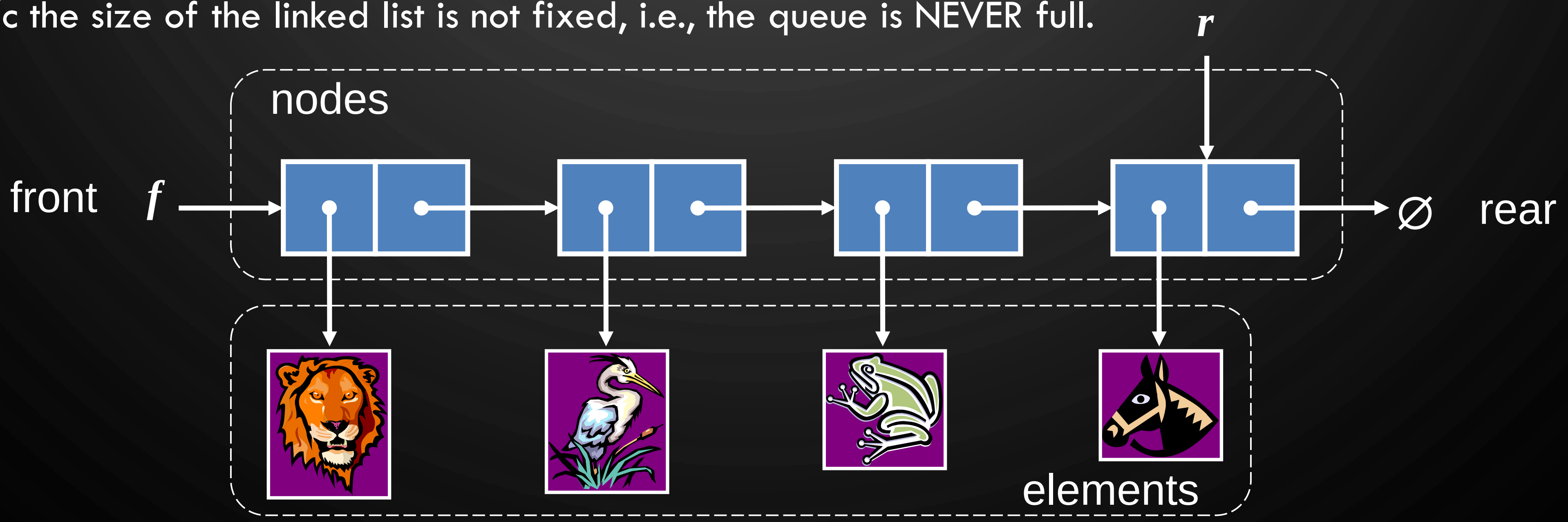
- In `enqueue(`e`)`, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- Similar to what we did for an array-based stack
- `enqueue(`e`)` has amortized running time
  - $O(n)$ with the incremental strategy
  - $O(1)$ with the doubling strategy

EXERCISE

- 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

QUEUE WITH A SINGLY LINKED LIST

- The front 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(o)	$O(1)$	$O(n)$ Worst Case $O(1)$ Best Case $O(1)$ Average Case	$O(1)$
front()	$O(1)$	$O(1)$	$O(1)$
size(), empty()	$O(1)$	$O(1)$	$O(1)$

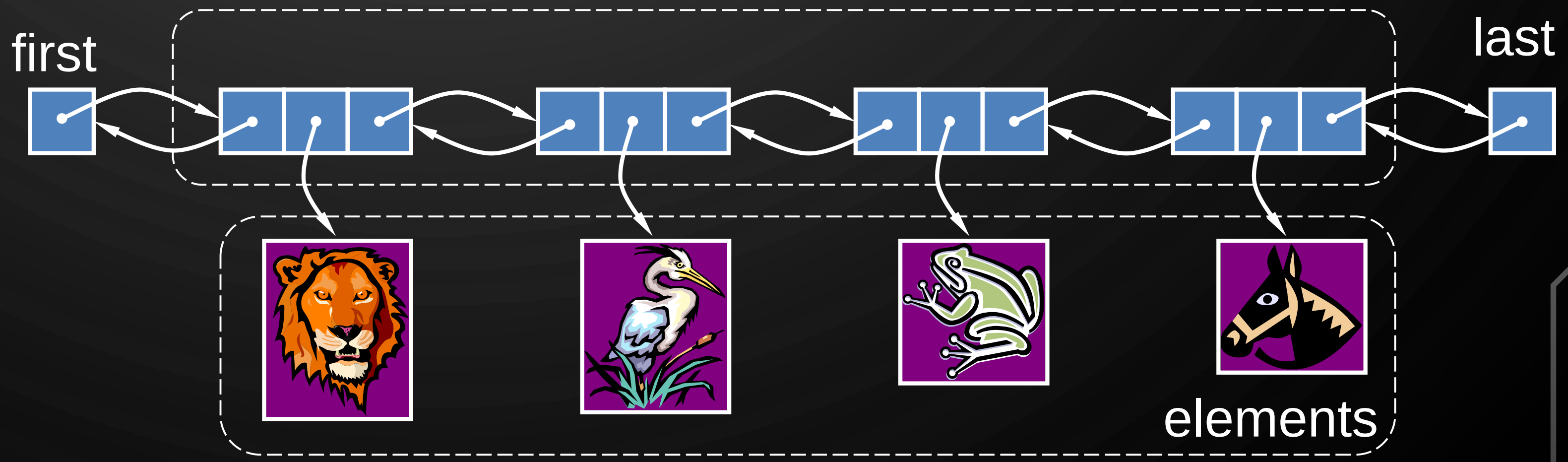
THE DOUBLE-ENDED QUEUE ADT (CH. 6.3)

- 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
  - `object removeFirst()`: removes and returns the element at the front of the queue
  - `object removeLast()`: removes and returns the element at the end of the queue

- Auxiliary queue operations:
  - `object first()`: returns the element at the front without removing it
  - `object 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 OF DEQUE ADT

- 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

INTERVIEW QUESTION 2

- In the classic problem of the Towers of Hanoi, you have 3 towers and N disks of different sizes which can slide onto any tower. The puzzle starts with disks sorted in ascending order of size from top to bottom (i.e. , each disk sits on top of an even larger one). You have the following constraints:
  (1) Only one disk can be moved at a time.
  (2) A disk is slid off the top of one tower onto the next tower.
  (3) A disk can only be placed on top of a larger disk.
  Write pseudocode to move the disks from the first tower to the last using stacks.