Object Oriented Programming II

Algorithms and Data Structures

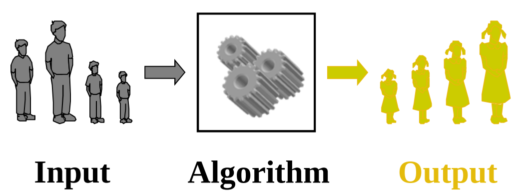
Analysis

Singly and double linked lists

Stacks, Queues, Sequences

Iterators

Analysis of Algorithms

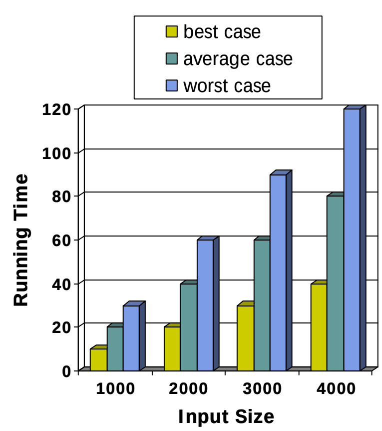


Important Concepts

- Running time
- Pseudo-code
- Counting primitive operations
- Asymptotic notation
- Asymptotic analysis
- Case study

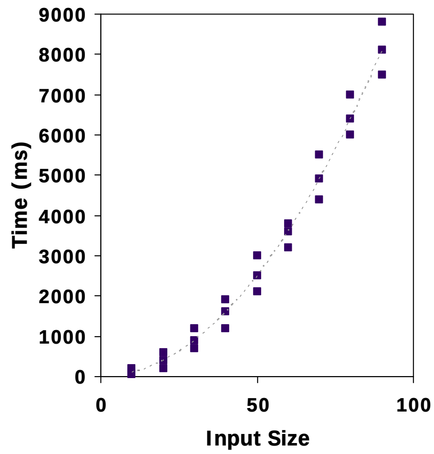
Running Time

- The running time of an algorithm varies with the input and typically grows with the input size
- Average case difficult to determine
- We focus on the worst case running time
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



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
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

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

Example: find max element of an array

Algorithm ***arrayMax***(***A***, ***n***)
  Input array ***A*** of ***n*** integers
  Output maximum element of ***A***

  currentMax ← ***A***[0]
  for ***i*** ← 1 **to** ***n*** − 1 **do**
    if *A*[*i*] > ***currentMax*** **then**
      currentMax ← *A*[***i***]
  return ***currentMax***

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
  var.method (***arg*** [, ***arg***…])
- Return value
  return ***expression***
- Expressions
  - ← Assignment (like = in Java)
  - = Equality testing (like == in Java)
  - n^2 Superscripts and other mathematical formatting allowed

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm	# operations
Algorithm ***arrayMax***(***A***, ***n***)	
currentMax ← *A*[0]	2
for *i* ← 1 **to** *n* − 1 **do**	2 + ***n***
if *A*[*i*] > *currentMax* **then**	2(***n*** − 1)
currentMax ← *A*[*i*]	2(***n*** − 1)
{ increment counter ***i*** }	2(***n*** − 1)
return *currentMax*	1
Total	7***n*** − 1

Estimating Running Time

- Algorithm ***arrayMax*** executes 7***n*** – 1 primitive operations in the worst case
- Define
  - ***a*** Time taken by the fastest primitive operation
  - ***b*** Time taken by the slowest primitive operation
- Let ***T***(***n***) be the actual worst-case running time of ***arrayMax***. We have

$$a\,(7n - 1) \le T(n) \le b(7n - 1)$$

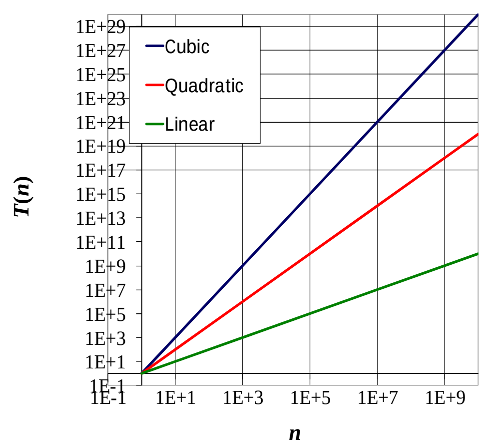
- Hence, the running time ***T***(***n***) is bounded by two linear functions

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects ***T***(***n***) by a constant factor, but
  - Does not alter the growth rate of ***T***(***n***)
- The linear growth rate of the running time ***T***(***n***) is an intrinsic property of algorithm ***arrayMax***

Growth Rates

- Growth rates of functions:
  - Linear ≈ ***n***
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
- In a log-log chart, the slope of the line corresponds to the growth rate of the function



Big-Oh Notation

- Useful to be able to estimate the CPU or memory resources an algorithm requires.
- "complexity analysis" attempts to characterize the relationship between the size of the data and resource usage with a simple formula.
- This can be useful if you are testing a program with a small amount of data, but later production runs will be with large data sets.

Big-Oh Notation …

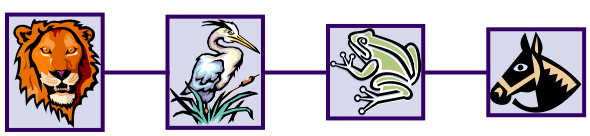
- Notation
  - The "O" stands for "order of".
- Dominance
  - Big-oh notation is only concerned with what happens for very large values of *N*, only the largest term in the formula is needed.
  - E.g. the number of operations in some sorts is N^2 - N. For large values, N is insignificant compared to N^2, therefore O(N) is N^2, and the N term is ignored.

Copyleft (c) 2000 www.leepoint.net/fred/
.Modified 2003 by Dyer Consulting

Big-Oh Notation…

- **Best, worst, and average cases**
- **Typical Orders**
  - Often people characterize the complexity as *constant* O(1), *logarithmic* O(log N), *linear* O(N), *polynomial* O(N^k), *exponential* O(2^N). Here is a table of some typical cases. This uses logarithms to base 2, but these are simply proportional to logarithms in other base.

Lists and Sequences

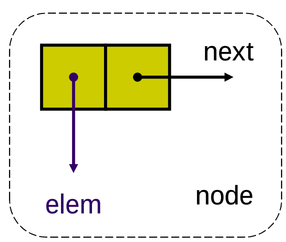


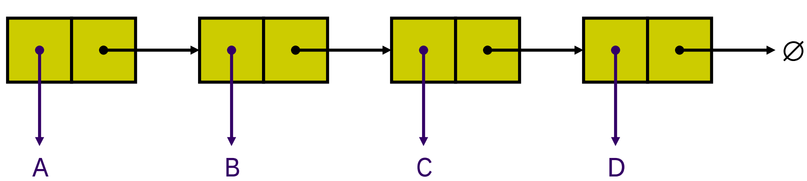
Important Concepts

- Singly linked list
- Position ADT and List ADT
- Doubly linked list
- Sequence ADT
- Implementations of the sequence ADT
- Iterators

Singly Linked List

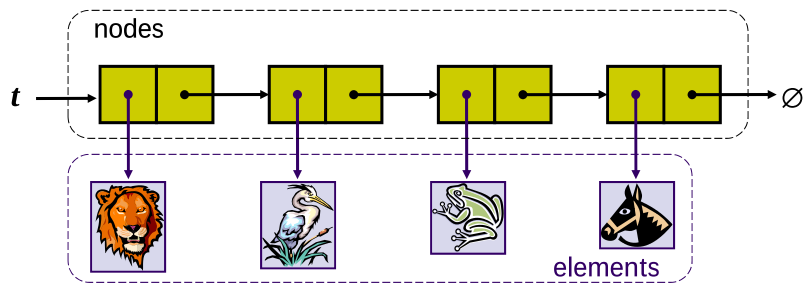
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node





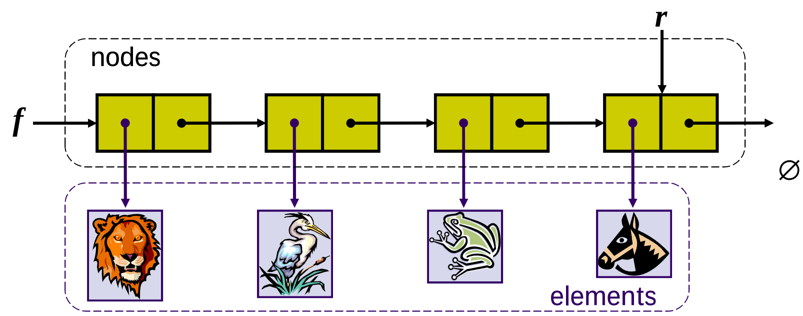
Stack with a Singly Linked List

- 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



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



Position ADT

<<interface>>

- The Position ADT models the notion of place within a data structure where a single object is stored
- It gives a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list
- Just one method:
  - object element(): returns the element stored at the position

List ADT

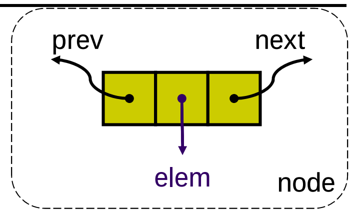
<<interface>>

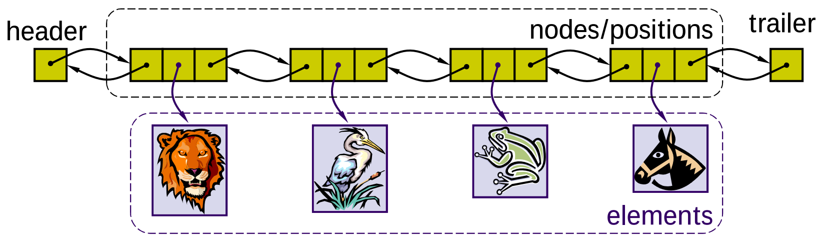
- The List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  - **size(), isEmpty()**
- Query methods:
  - **isFirst(p), isLast(p)**

Accessor methods:
  - **first(), last()**
  - **before(p), after(p)**
- Update methods:
  - **replaceElement(p, o), swapElements(p, q)**
  - **insertBefore(p, o), insertAfter(p, o),**
  - **insertFirst(o), insertLast(o)**
  - **remove(p)**

Doubly Linked List

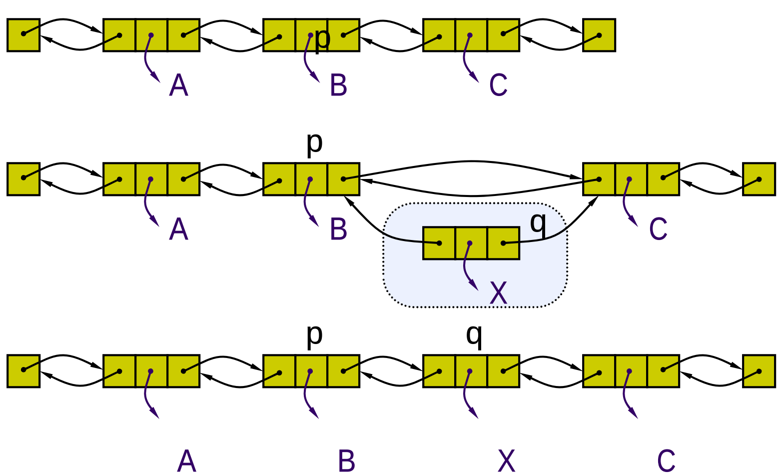
- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special ***trailer*** and ***header*** nodes





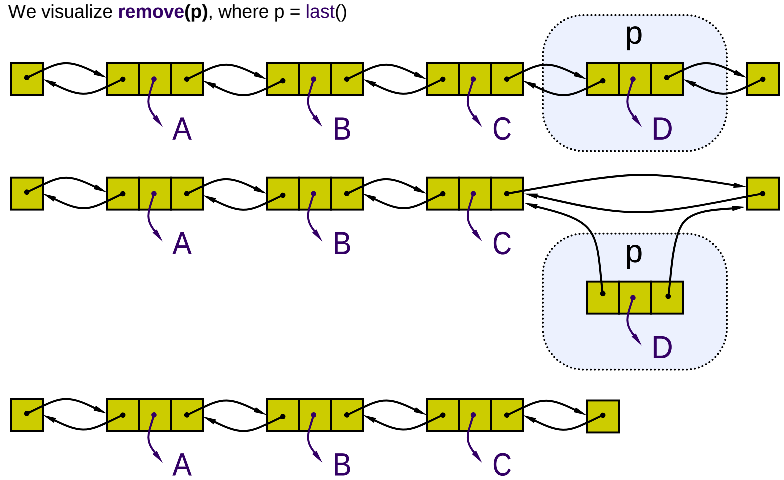
Insertion

- We visualize operation **insertAfter(p, X)**, which returns position q



Deletion

- We visualize **remove(p)**, where p = last()



Performance

- In the implementation of the List ADT by means of a doubly linked list
  - The space used by a list with n elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the operations of the List ADT run in $O(1)$ time
  - Operation element() of the Position ADT runs in $O(1)$ time

Sequence ADT
<<interface>>

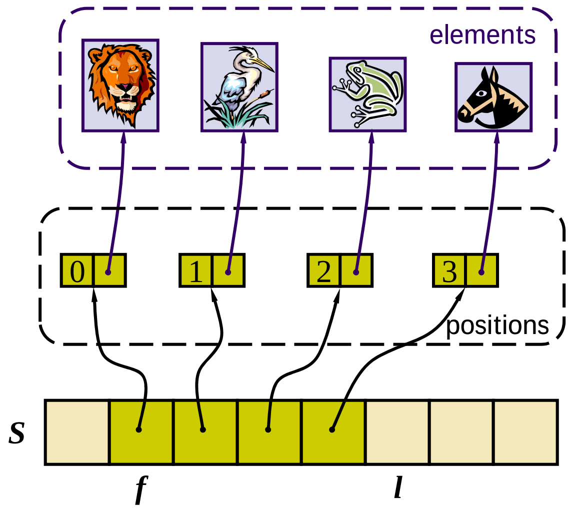
- The Sequence ADT is the union of the Vector and List ADTs
- Elements accessed by
  - Rank, or
  - Position
- Generic methods:
  - **size(), isEmpty()**
- Vector-based methods:
  - **elemAtRank(r), replaceAtRank(r, o), insertAtRank(r, o), removeAtRank(r)**
- List-based methods:
  - **first(), last(), before(p), after(p), replaceElement(p, o), swapElements(p, q), 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

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

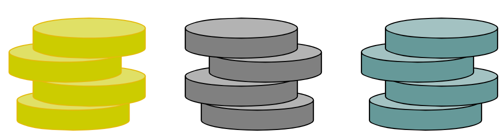
Operation	Array	List
size, isEmpty	1	1
atRank, rankOf, elemAtRank	1	***n***
first, last, before, after	1	1
replaceElement, swapElements	1	1
replaceAtRank	1	***n***
insertAtRank, removeAtRank	***n***	***n***
insertFirst, insertLast	1	1
insertAfter, insertBefore	***n***	1
remove	***n***	1

Iterators

- An iterator abstracts the process of scanning through a collection of elements
- Methods of the ObjectIterator ADT (<<interface>>):
  - **object object()**
  - **boolean hasNext()**
  - **object nextObject()**
  - **reset()**
- Extends the concept of Position by adding a traversal capability
- Implementation with an array or singly linked list
- An iterator is typically associated with an another data structure
- We can augment the Stack, Queue, Vector, List and Sequence ADTs with method:
  - **ObjectIterator elements()**
- Two notions of iterator:
  - snapshot: freezes the contents of the data structure at a given time
  - dynamic: follows changes to the data structure

06/03/2017 Rev1.0 CST8288 - OOP II Data Structures and Algorithms in Java, (Second Edition, 2001), by Michael T. Goodrich and Roberto Tamassia 33

Stacks



Important Concepts

- The Stack ADT
- Applications of Stacks
- Array-based implementation
- Growable array-based stack

Abstract Data Types (ADTs)

- An abstract data type (ADT or <<interface>>) 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

The Stack ADT

<<interface>>

- 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

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

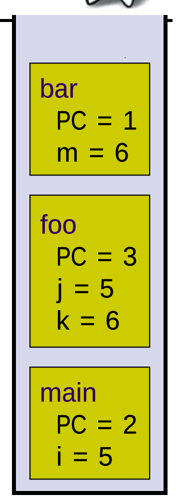
Applications of Stacks

- 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

```
main() {
  int i = 5;
  foo(i);
  }
foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }
bar(int m) {
  …
  }
```

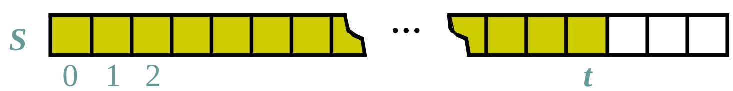


Array-based Stack

- 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 $t + 1$

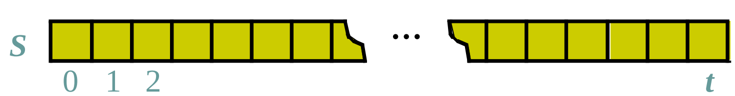
Algorithm *pop*()
  if *isEmpty*() **then**
    throw *EmptyStackException*
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$



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
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$



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

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

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;
}
```


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;
   }
```

Queues

The Queue ADT

<<interface>>

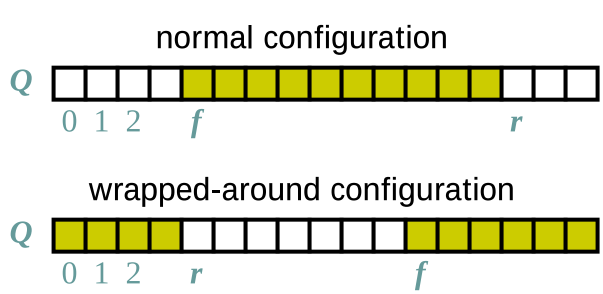
- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the **first-in first-out** scheme
- **Insertions are at the rear** of the queue and **removals are at the front** of the queue
- Main queue operations:
  - **enqueue(object)**: inserts an element at the end of the queue
  - **object 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

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

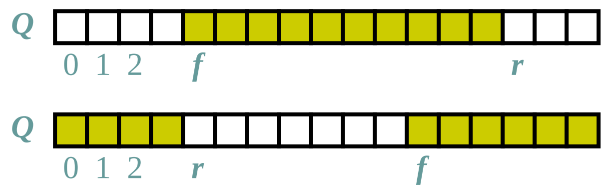


Queue Operations

- We use the modulo operator (remainder of division)

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

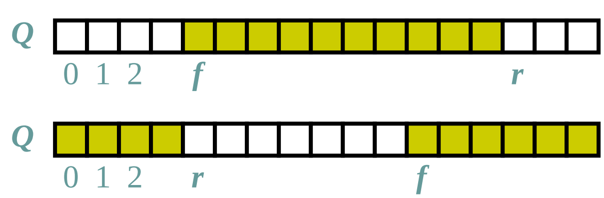
Algorithm ***isEmpty***()
  return $(f = r)$



Queue Operations (cont.)

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

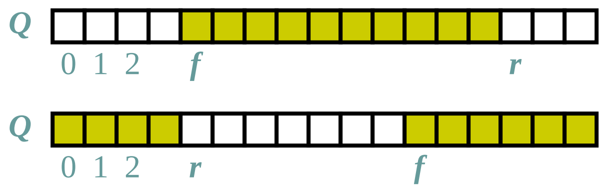
Algorithm ***enqueue***(o)
  if *size*() $= N - 1$ **then**
    throw *FullQueueException*
  else
    $Q[r] \leftarrow o$
    $r \leftarrow (r + 1) \bmod N$



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
    $o \leftarrow Q[f]$
    $f \leftarrow (f + 1) \bmod N$
    return o



Growable Array-based Queue

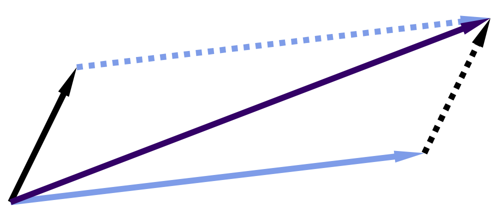
- In an enqueue operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- Similar to what we did for an array-based stack
- The enqueue operation has amortized running time
  - $O(n)$ with the incremental strategy
  - $O(1)$ with the doubling strategy

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;
}
```


Vectors



Important Concepts

- The Vector ADT
- Array-based implementation

The Vector ADT

<<interface>>

- The Vector ADT extends the notion of array by storing a sequence of arbitrary objects
- An element can be accessed, 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)

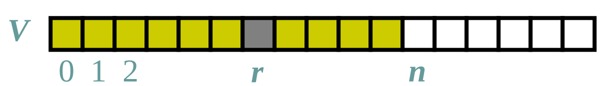
- Main vector operations:
  - **object elemAtRank(integer r)**: returns the element at rank r without removing it
  - **object replaceAtRank(integer r, object o)**: replace the element at rank with o and return the old element
  - **insertAtRank(integer r, object o)**: insert a new element o to have rank r
  - **object removeAtRank(integer r)**: removes and returns the element at rank r
- Additional operations **size()** and **isEmpty()**

Applications of Vectors

- Direct applications
  - Sorted collection of objects (elementary database)
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

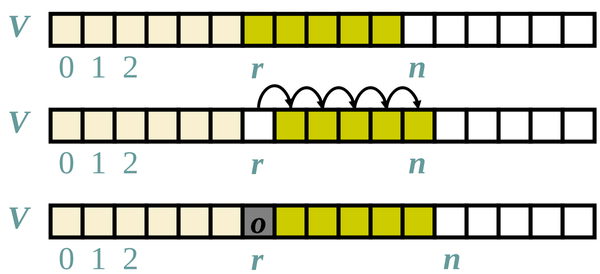
Array-based Vector

- Use an array ***V*** of size ***N***
- A variable ***n*** keeps track of the size of the vector (number of elements stored)
- Operation ***elemAtRank***(***r***) is implemented in ***O***(1) time by returning ***V***[***r***]



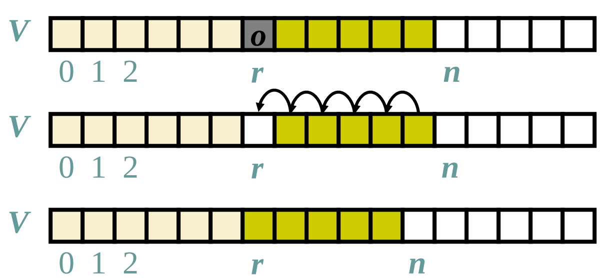
Insertion

- In operation ***insertAtRank***(***r, o***), we need to make room for the new element by shifting forward the ***n*** – ***r*** elements ***V***[***r***], …, ***V***[***n*** – 1]
- In the worst case (***r*** = 0), this takes ***O***(***n***) time



Deletion

- In operation ***removeAtRank***(***r***), we need to fill the hole left by the removed element by shifting backward the ***n*** – ***r*** – 1 elements ***V***[***r*** + 1], …, ***V***[***n*** – 1]
- In the worst case (***r*** = 0), this takes ***O***(***n***) time



Performance

- In the array based implementation of a Vector
  - The space used by the data structure is ***O***(***n***)
  - ***size***, ***isEmpty***, ***elemAtRank*** and ***replaceAtRank*** run in ***O***(1) time
  - ***insertAtRank*** and ***removeAtRank*** run in ***O***(***n***) time
- If we use the array in a circular fashion, ***insertAtRank***(0) and ***removeAtRank***(0) run in ***O***(1) time
- In an ***insertAtRank*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one