INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

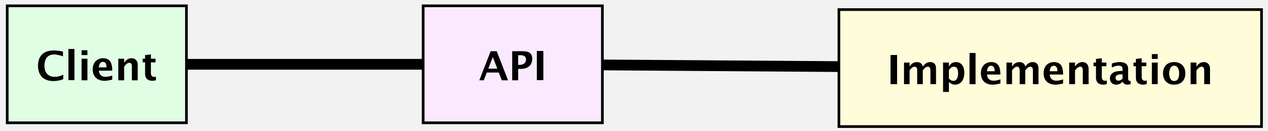
Rutgers University

2. Stacks and Queues

- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics (optional)*
- *Iterators (optional)*
- *Applications*

Client, implementation, API

Separate client and implementation via API.



API: operations that characterize the behavior of a data type.

Client: program that uses the API operations.

Implementation: code that implements the API operations.

Benefits.

- Design: create modular, reusable libraries.
- Performance: substitute faster implementations.

Ex. Stack, queue, bag, priority queue, symbol table, union-find, ...

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

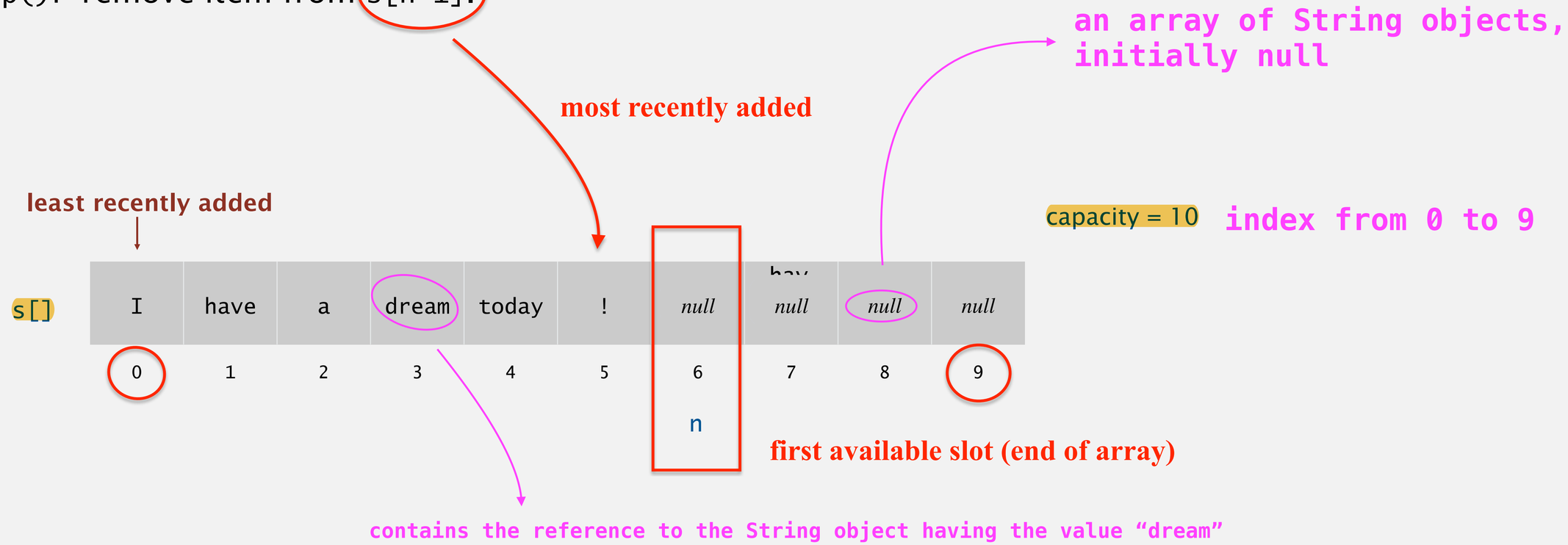
Rutgers University

2. Stacks and Queues

- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics*
- *Iterators (optional)*
- *Applications*

Fixed-capacity stack: array implementation

- Use array `s[]` to store `n` items on stack.
- `push()`: add new item at `s[n]`.
- `pop()`: remove item from `s[n-1]`.



Defect. Stack overflows when `n` exceeds `capacity`. [stay tuned]

Fixed-capacity stack: array implementation


```
public class FixedCapacityStackOfStrings
{
   private String[] s;
   private int n = 0;

   public FixedCapacityStackOfStrings(int capacity)
   {  s = new String[capacity];  }

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

   public void push(String item)
   {  s[n++] = item;  }

   public String pop()
   {  return s[--n];  }
}
```

post-increment operator:
use as index into array;
then increment n

increase the size by 1, after the array access

decrease the size by 1, before the array access

pre-decrement operator:
decrement n;
then use as index into array

Stack considerations

Overflow and underflow.

- Underflow: throw exception if pop() called on an empty stack.
- Overflow: use "resizing array" for array implementation. [stay tuned]

client programs need to decide the capacity before creating an instance of Stack

Null items. We allow null items to be added.

Duplicate items. We allow an item to be added more than once.

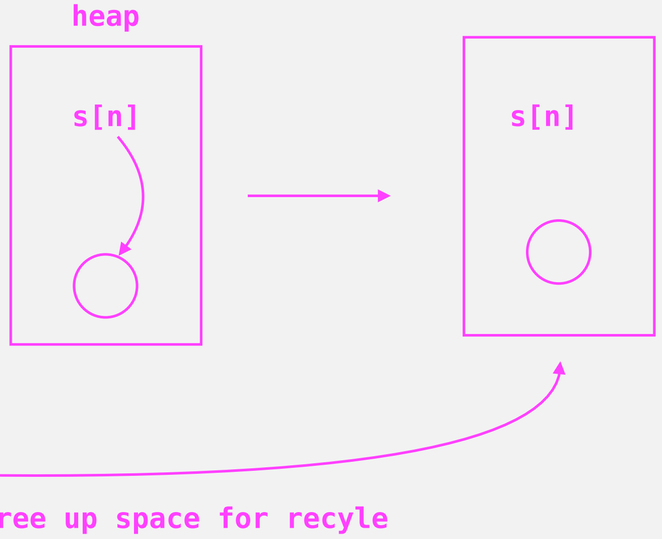
Loitering. Holding a reference to an object when it is no longer needed.

```
public String pop()
{  return s[--n];  }
```

loitering


```
public String pop()
{
   String item = s[--n];
   s[n] = null;
   return item;
}
```

no loitering



Stacks: quiz

How to implement a fixed-capacity stack with an array?

A. least recently added — push and pop?

B. most recently added — why not? — push and pop?

C. *Both A and B.*

D. *Neither A nor B.*

Answer = A

Function calls

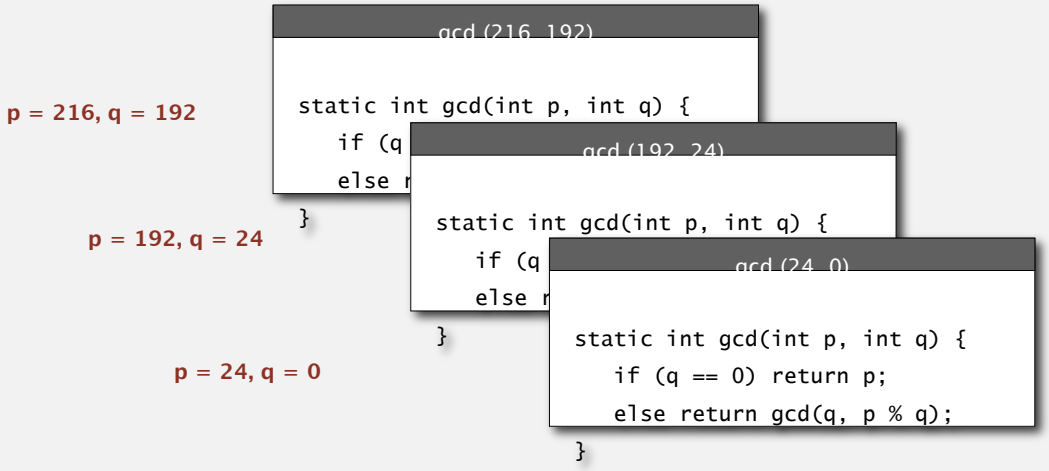
Optional content

How a compiler implements a function.

- Function call: push local environment and return address.
- Return: pop return address and local environment.

Recursive function. Function that calls itself.

Note. Can always use an explicit stack to remove recursion.



Arithmetic expression evaluation

Goal. Evaluate infix expressions.

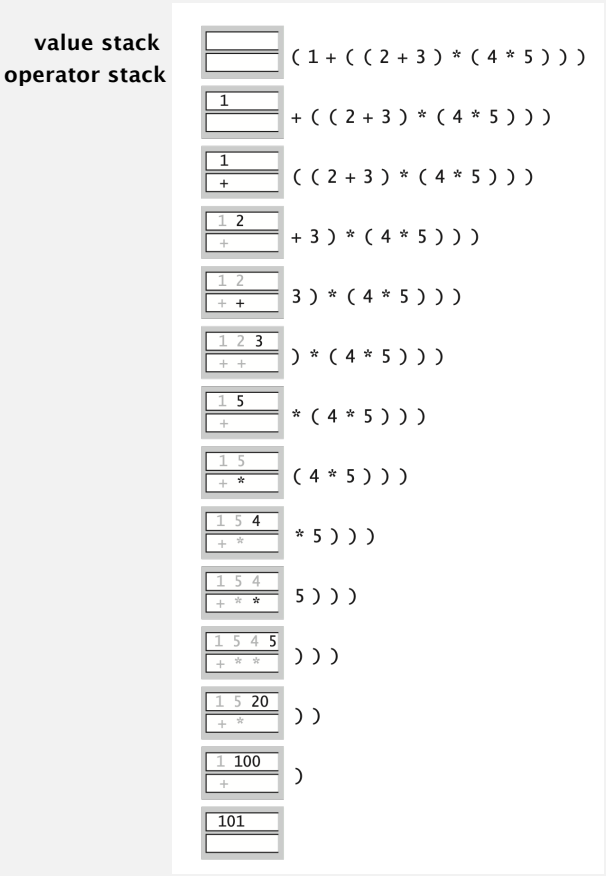
```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

operand operator

Two-stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parenthesis: ignore.
- Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the value stack.

Context. An interpreter!



Arithmetic expression evaluation

```
public class Evaluate
{
   public static void main(String[] args)
   {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty()) {
         String s = StdIn.readString();
         if      (s.equals("("))    /* noop */ ;
         else if (s.equals("+"))    ops.push(s);
         else if (s.equals("*"))    ops.push(s);
         else if (s.equals(")"))
         {
            String op = ops.pop();
            if      (op.equals("+")) vals.push(vals.pop() + vals.pop());
            else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
         }
         else vals.push(Double.parseDouble(s));
      }
      StdOut.print
   }
}
```

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

Correctness

Q. Why correct?

A. When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

as if the original input were:

```
( 1 + ( 5 * ( 4 * 5 ) ) )
```

Repeating the argument:

```
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

Extensions. More ops, precedence order, associativity.

Introduction to Linked Lists

Simplest singly-linked data structure – Linked List

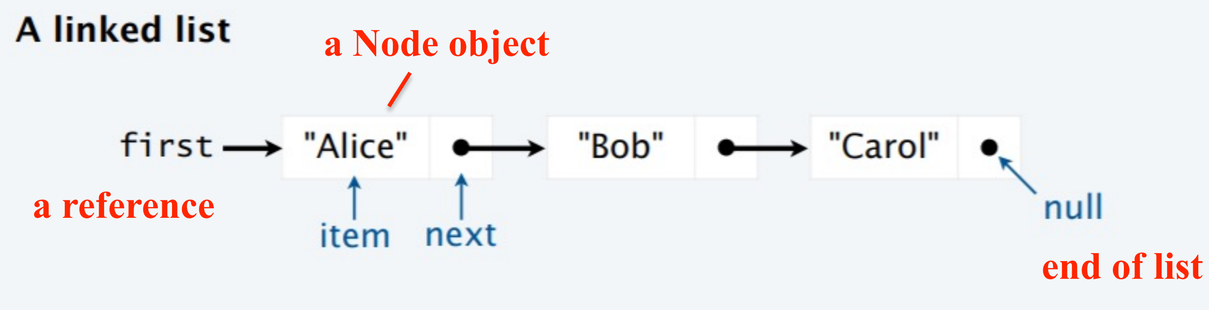
Linked list

- A recursive data structure.
- Def. A *linked list* is null or a reference to a *node*.
- Def. A *node* is a data type that contains a reference to a node.
- Unwind recursion: A linked list is a sequence of nodes.

Representation

- Use a private nested class Node to implement the node abstraction.
- For simplicity, start with nodes having two values: a String and a Node.

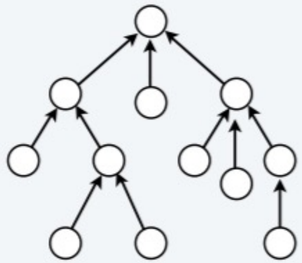
```
private class Node
{
   private String item;
   private Node next;
}
```



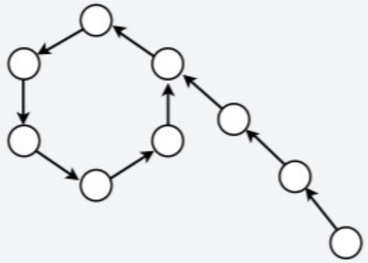
Singly Linked Data Structures

Linked list (this lecture)

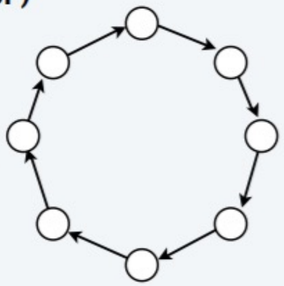
Tree



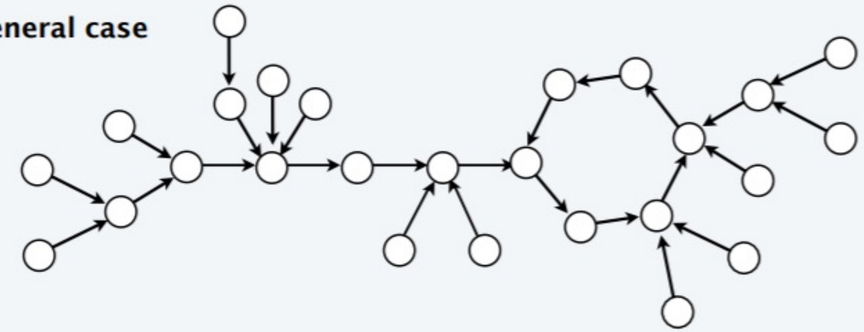
Rho



Circular list (TSP)



General case



Multiply linked structures: many more possibilities!

From the point of view of a particular object, all of these structures look the same.

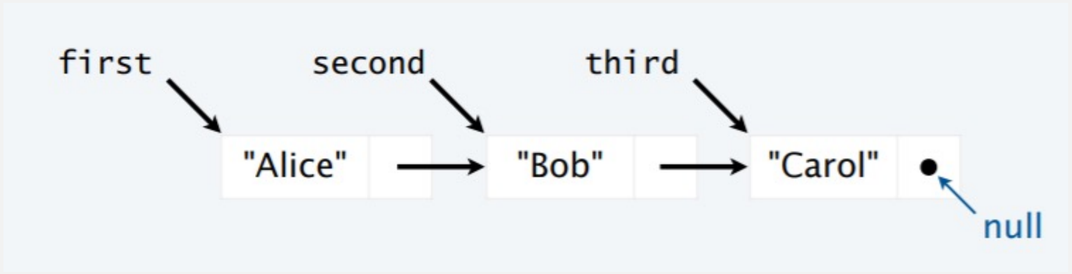
Building a Linked List

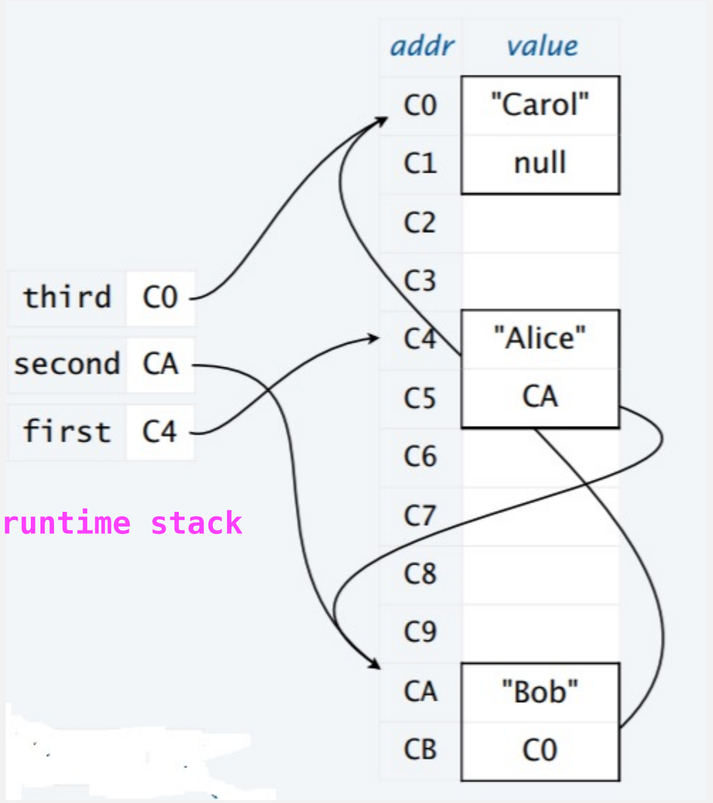
```
Node third  = new Node();
third.item  = "Carol";
third.next  = null;

Node second = new Node();
second.item = "Bob";
second.next = third;

Node first  = new Node();
first.item  = "Alice";
first.next  = second;
```

one node so far

the reference to the next node

the reference to the next node
first is the head node





scattered memory addresses
in the heap

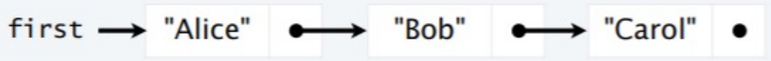
List Processing Code

Standard operations for processing data structured as a singly-linked list

- Add a node at the beginning
- Remove and return the node at the beginning
- Add a node at the end (requires a reference to the last node)
- Traverse the list (visit every node, in sequence).

List Processing code – Remove and Return First Item

Goal. Remove and return the first item in a linked list `first`.



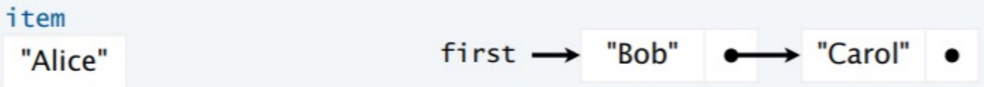

```
item = first.item;
```

keep the ref. of the first item


```
first = first.next;
```

"remove" the first item by changing the ref.


```
return item;
```



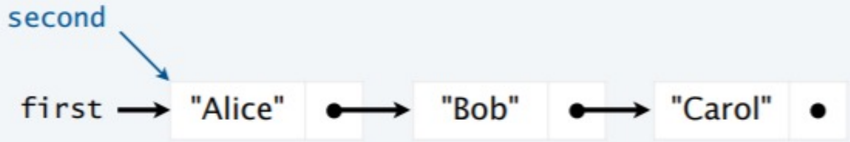
List Processing Code – Add a new node at the beginning

Goal. Add item to a linked list first.

add item to head of list

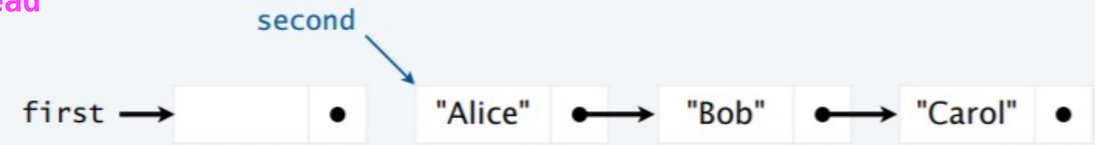

```
Node second = first;
```

keep the ref. to the old head



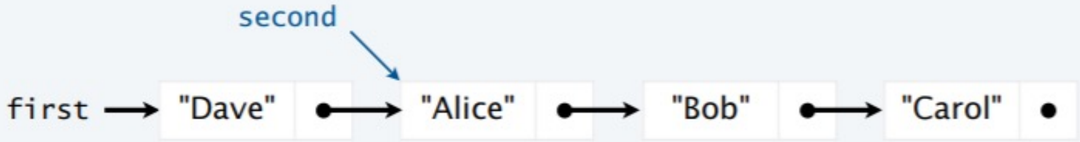

```
first = new Node();
```

new node is the head




```
first.item = item;
first.next = second;
```

point to the old head



List processing code – Traverse a List

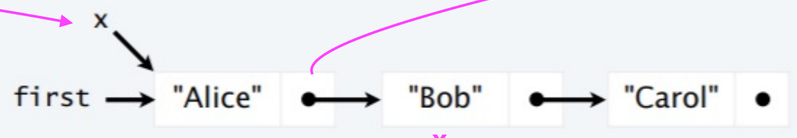
Goal. Visit every node on a linked list `first`.

traverse the list sequentially

```
Node x = first;
while (x != null)
{
   StdOut.println(x.item);
   x = x.next;
}
```

start at the head of list

the reference to the next node



move to the next node

StdOut

```
Alice
Bob
Carol
```

we don't know how many nodes, so use a while loop

Pop quiz1 on Linked Lists

Q. What is the effect of the following code (not-so-easy question)?

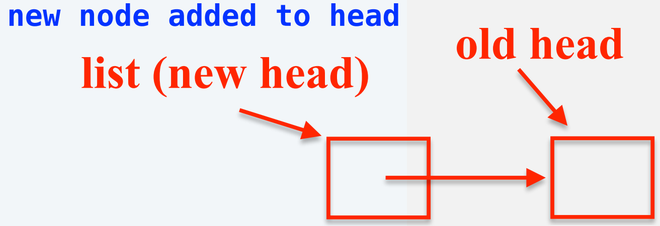
```
...
Node list = null;
while (!StdIn.isEmpty())
{
   Node old = list;
   list = new Node();
   list.item = StdIn.readString();
   list.next = old;
}
for (Node t = list; t != null; t = t.next)
   StdOut.println(t.item);
...
```

save the reference to the head

head tail

traverse the linked list

new node added to head



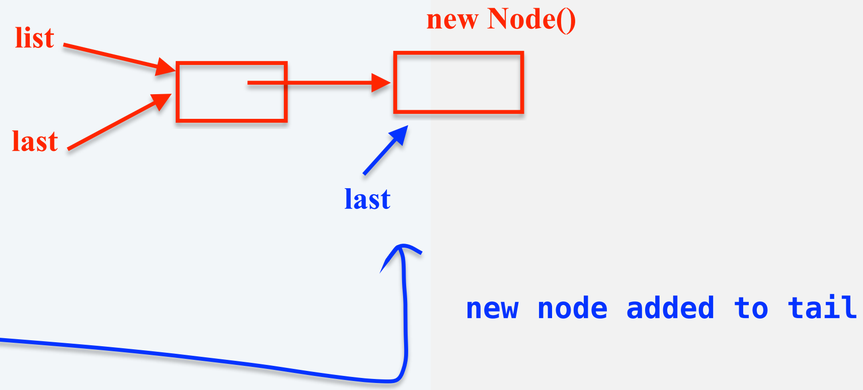
Pop quiz 2 on Linked Lists

Q. What is the effect of the following code (not-so-easy question)?

```
...
Node list = new Node();
list.item = StdIn.readString();
Node last = list;
while (!StdIn.isEmpty())
{
   last.next = new Node();
   last = last.next;
   last.item = StdIn.readString();
}
...
```

only node in the list so far

make the new node as the tail node



new node added to tail

Stacks and queues: quiz

How to implement a stack with a singly linked list?

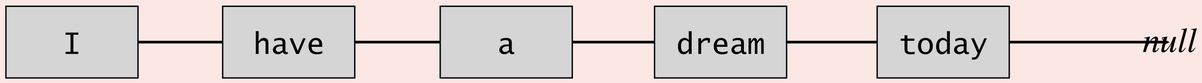
Last in First out (LIFO) -> remove most recently added first

A.

least recently added

why not?

how to push and pop?

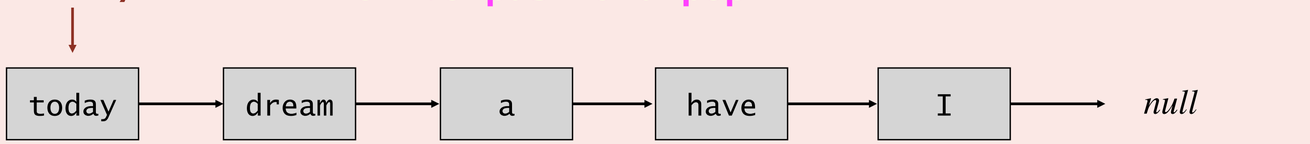


most recently added

B.

most recently added

how to push and pop?



least recently added

C. *Both A and B.*

D. *Neither A nor B.*

Answer = B

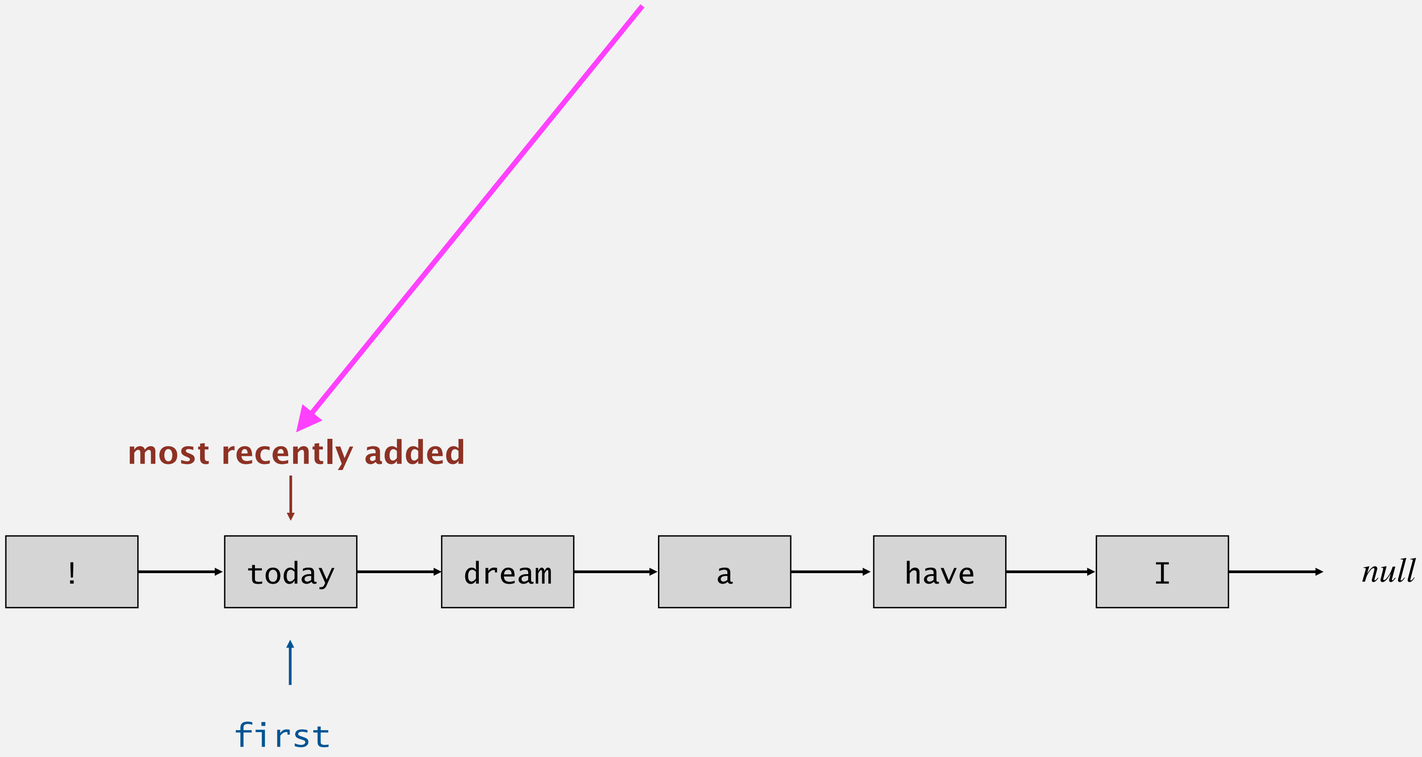
Stack: linked-list implementation

Last in First out

LO 2.1, 2.2

- Maintain pointer `first` to first node in a singly linked list.
- Push new item before `first`.
- Pop item from `first`.

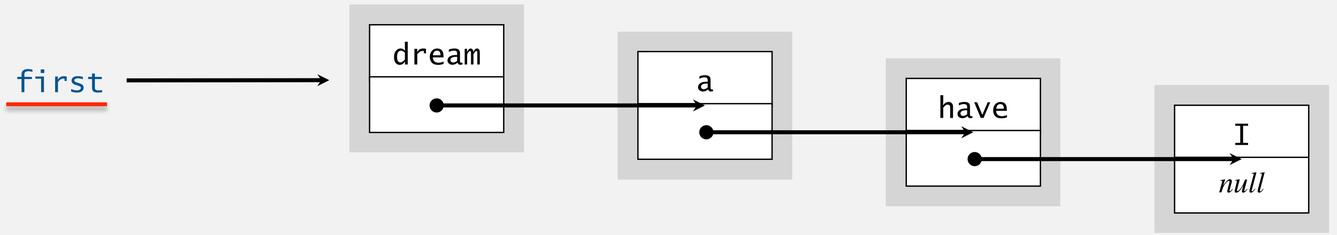
–> add and remove operations on one end only



Stack pop: linked-list implementation

remove a node

singly linked list

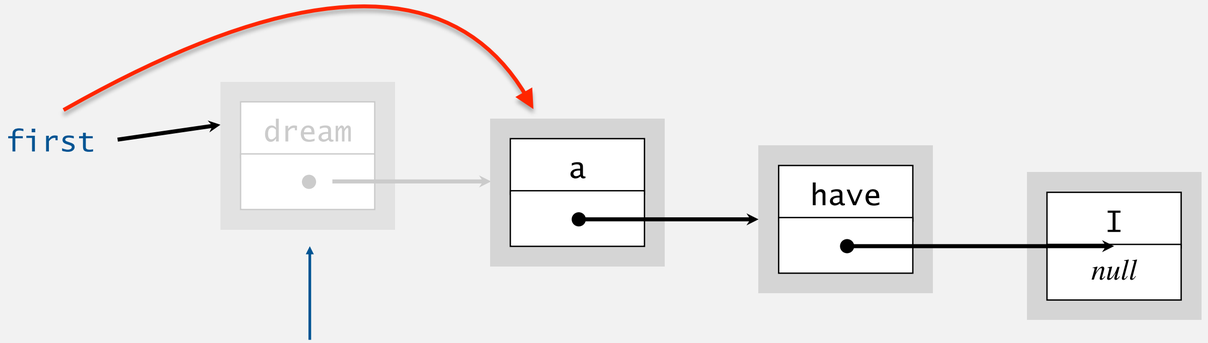


save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```



garbage collector reclaims memory

when no remaining references

return saved item

```
return item;
```

inner class

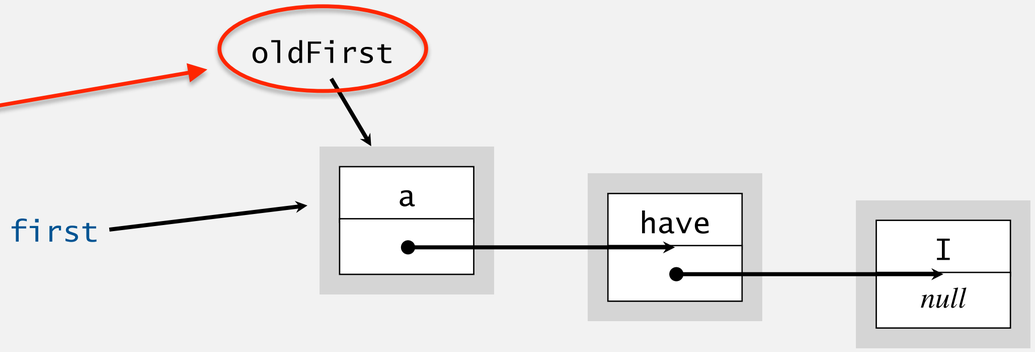
```
private class Node
{
   private String item;
   private Node next;
}
```

Stack push: linked-list implementation

add a new node

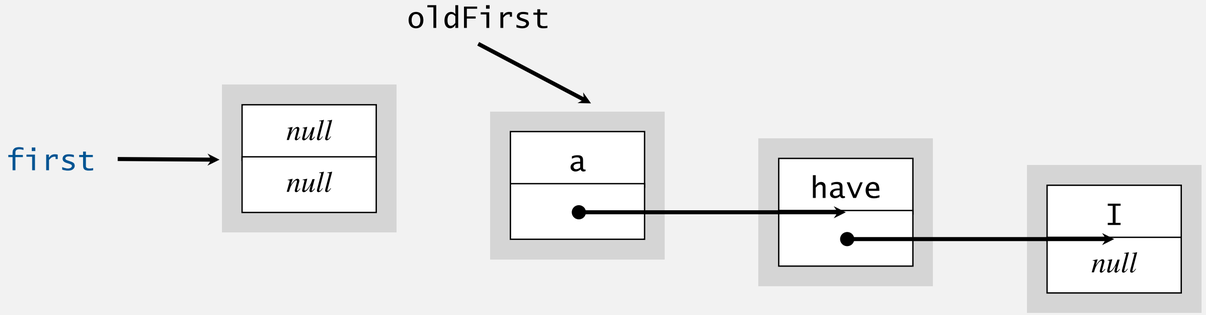
save a link to the list

```
Node oldFirst = first;
```



create a new node for the beginning

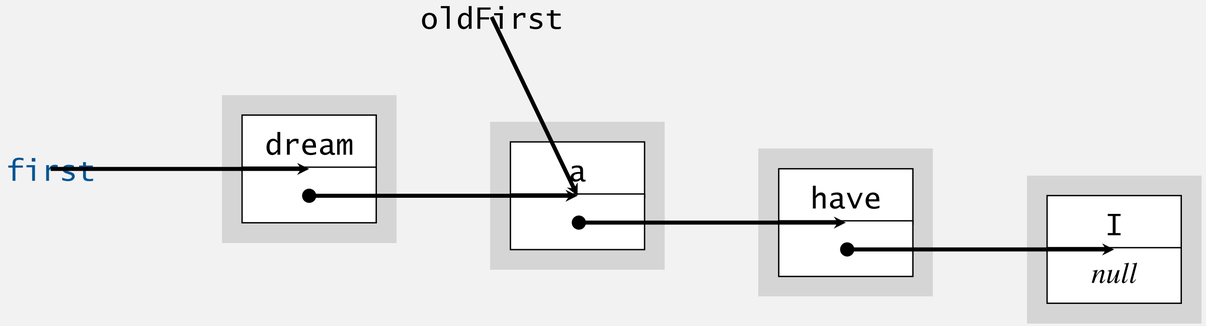
```
first = new Node();
```



initialize the instance variables in the new Node

```
first.item = "dream";
first.next = oldFirst;
```

link the new head to list



inner class

```
private class Node
{
   private String item;
   private Node next;
}
```

Stack: linked-list implementation


```
public class LinkedStackOfStrings
{
   private Node first = null;

   private class Node
   {
      private String item;
      private Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node oldFirst = first;
      first = new Node();
      first.item = item;
      first.next = oldFirst;
   }
   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

outer class

used only in outer class LinkedStackOfStrings

private inner class

(access modifiers for instance variables of such a class don't matter)

no Node constructor defined ⇒

Java supplies default no-argument constructor

instance variables are set to default values
— numeric types to 0 or 0.0
— booleans to false
— reference types to null

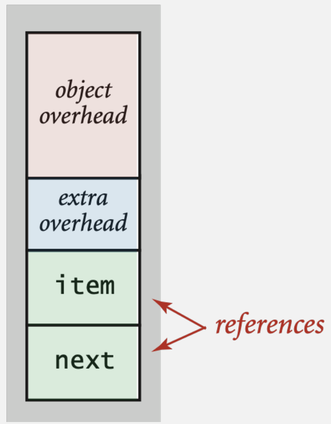
Stack: linked-list implementation performance

Proposition. Every operation takes $O(1)$ time.

Proposition. A stack with n items has n Node objects and uses $\sim 40\,n$ bytes.

inner class

```
private class Node
{
   String item;
   Node next;
}
```



16 bytes (object overhead)

8 bytes (inner class extra overhead)

8 bytes (reference to String)

8 bytes (reference to Node)

40 bytes per stack Node

Remark. This counts only the memory for the stack itself.

(not the memory for the strings, which the client owns)

item contains the reference to the string

2. Stacks and Queues

- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics*
- *Iterators (optional)*
- *Applications (optional)*

Stack: resizing-array implementation to address the overflow and underflow problems

LO 2.3

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

- `push()`: increase size of array `s[]` by 1.
- `pop()`: decrease size of array `s[]` by 1.

Too expensive.

- Need to copy all items to a new array, for each operation.
- Array accesses to add first n items = $n + (2 + 4 + 6 + \ldots + 2(n-1)) \sim n^2$.

infeasible for large n

1 array access per push

$2(k-1)$ array accesses to expand to size k (ignoring cost to create new array)

copying 1 item involves 1 read and 1 write
e.g., increase the size from 2 to 3,
need 2 read and 2 write (k=2, 2*k = 4)

Challenge. Ensure that array resizing happens infrequently.

Stack: resizing-array implementation

LO 2.3

Q. How to grow array?

A. If array is full, create a new array of twice the size, and copy items.

"repeated doubling"

```
public ResizingArrayStackOfStrings()
{  s = new String[1]; }

public void push(String item)
{
   if (n == s.length) resize(2 * s.length);
   s[n++] = item;
}

private void resize(int capacity)
{
   String[] copy = new String[capacity];
   for (int i = 0; i < n; i++)
      copy[i] = s[i];
   s = copy;
}
```

$1 + 2 + 4 + \ldots + n = 2^{(k+1)} - 1$

$n = 2^k$

Array accesses to add first $n = 2^i$ items. $n + (2 + 4 + 8 + \ldots + n) \sim 3n$.

feasible for large n

copying 1 item involves 1 read and 1 write
e.g., increase the size from 2 to 4,
need 2 read and 2 write (k=4, 2*(k/2) = 4)

2n - 2

1 array access per push

k array accesses to double to size k (ignoring cost to create new array)

Stack: resizing-array implementation

LO 2.3

Q. How to shrink array?

First try.

- `push()`: double size of array `s[]` when array is full.
- `pop()`: halve size of array `s[]` when array is one-half full.

Too expensive in worst case.

- Consider alternating sequence of push and pop operations when array is full.
- Each operation takes $O(n)$ time.

thrashing

Stack: resizing-array implementation

LO 2.3

Q. How to shrink array?

Efficient solution.

- `push()`: double size of array `s[]` when array is full.
- `pop()`: halve size of array `s[]` when array is one-quarter full.

```
public String pop()
{
   String item = s[--n];
   s[n] = null;
   if (n > 0 && n == s.length/4) resize(s.length/2);
   return item;
}
```

do not want to shrink to size 0

Invariant. Array is between 25% and 100% full.

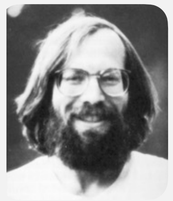
Stack resizing-array implementation: performance

LO 2.3

Proposition. Starting from an empty stack, any sequence of m push and pop operations takes $O(m)$ time.

so, on average, each of m operation takes $\Theta(1)$ time

Amortized analysis. Starting from an empty data structure, average running time per operation over a worst-case sequence of operations.



Bob Tarjan
(1986 Turing award)

	worst	amortized
construct	1	1
push	n	1
pop	n	1
size	1	1

order of growth of running time
for resizing-array stack with n items

Stack resizing-array implementation: memory usage

LO 2.3

Proposition. A `ResizingArrayStackOfStrings` uses between $\sim 8n$ and $\sim 32n$ bytes of memory for a stack with n items.

- $\sim 8n$ when full. [array length = n]
- $\sim 32n$ when one-quarter full. [array length = $4n$]

4n x 8

n is the number of items currently in the array

```
public class ResizingArrayStackOfStrings
{
   private String[] s;
   private int n = 0;

   ⋮
}
```

8 bytes × array length

an array of references

4 bytes

Remark. This counts only the memory for the stack itself.

(not the memory for the strings, which the client allocates)

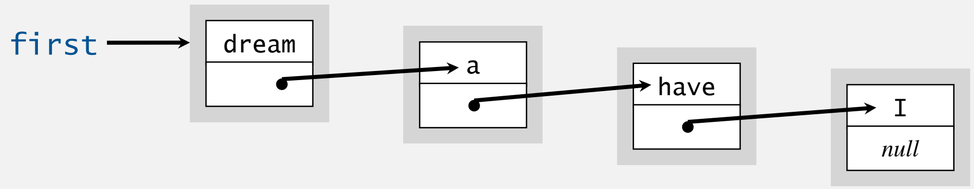
Stack implementations: resizing array vs. linked list

LO 2.4, 2.5, 2.6, 2.7, 2.8, 2.9

Tradeoffs. Can implement a stack with either resizing array or linked list. Which is better?

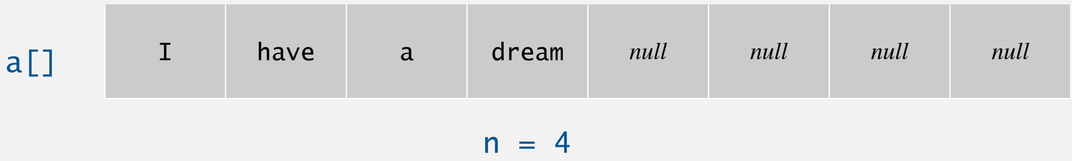
Linked-list implementation.

- Stronger performance guarantee: $O(1)$ worst case.
- More memory. overhead of objects



Resizing-array implementation.

- Weaker performance guarantee: $O(1)$ amortized.
- Less memory.
- Better use of cache. consecutive memory space



INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

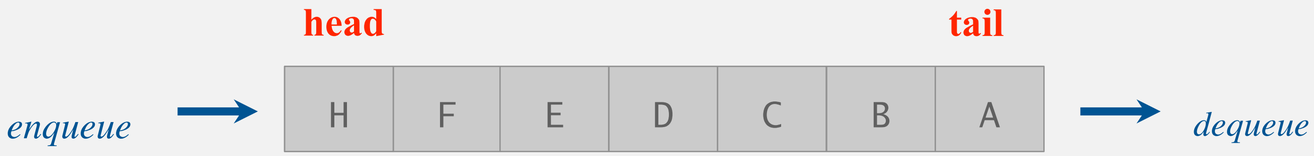
Rutgers University

2. Stacks and Queues

- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics (optional)*
- *Iterators (optional)*
- *Applications*

Queue of strings API

First in First out (FIFO)



public class QueueOfStrings	
QueueOfStrings()	*create an empty queue*
void enqueue(String item)	*add a new string to queue*
String dequeue()	*remove and return the string least recently added*
boolean isEmpty()	*is the queue empty?*
int size()	*number of strings on the queue*

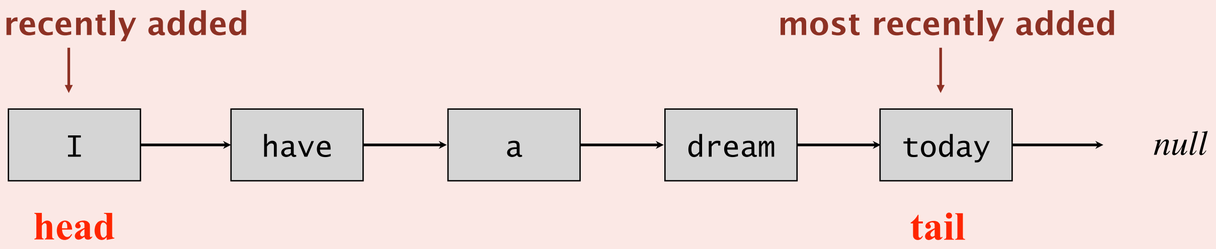
Performance goal. Every operation takes $\Theta(1)$ time.

Stacks and queues: quiz

How to implement a queue with a singly linked linked list?

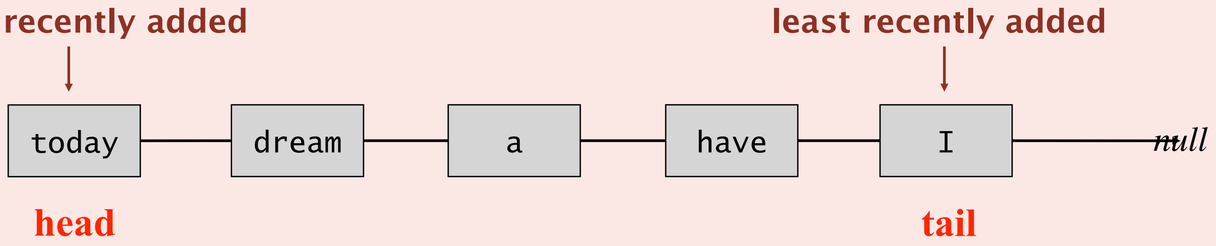
First in First out -> remove least recently added first

A.



how to enqueue and dequeue?

B.



how to enqueue and dequeue?

C. *Both A and B.*

Answer = A

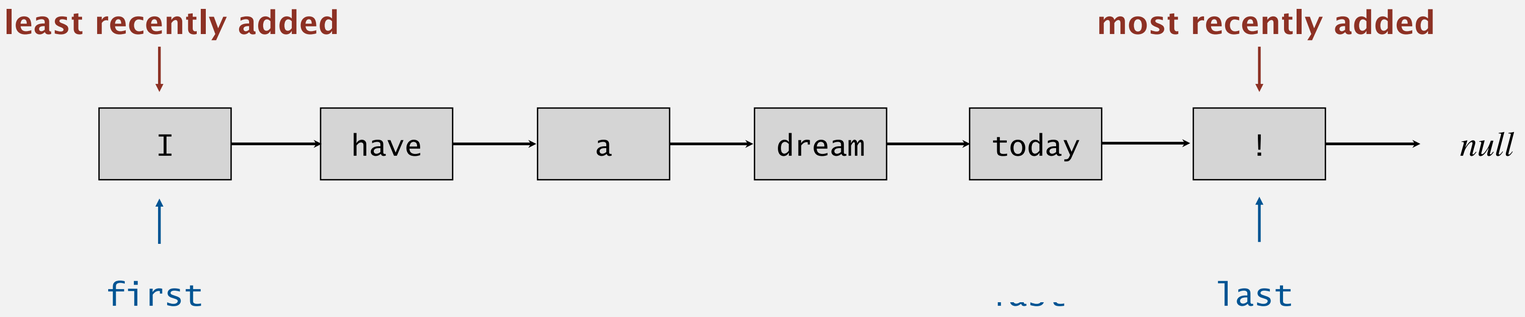
D. *Neither A nor B.*

Queue: linked-list implementation

LO 2.1

- Maintain one pointer `first` to first node in a singly linked list.
- Maintain another pointer `last` to last node.
- Dequeue from `first`.
- Enqueue after `last`.

-> operations performed on both ends



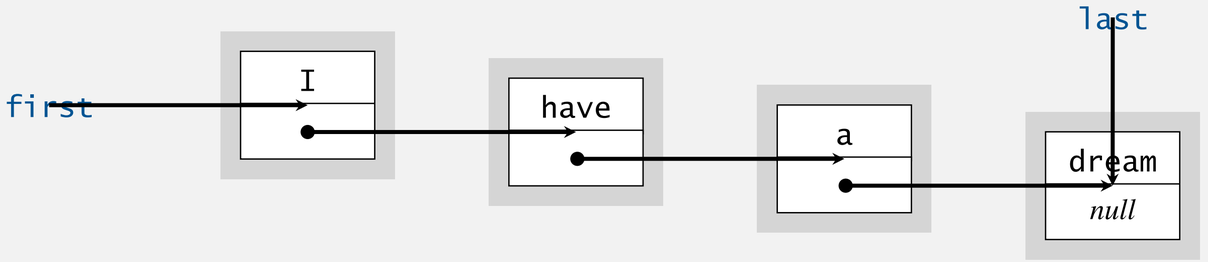
Queue dequeue: linked-list implementation

LO 2.1

Code is identical to pop().

remove a node from head

singly linked list

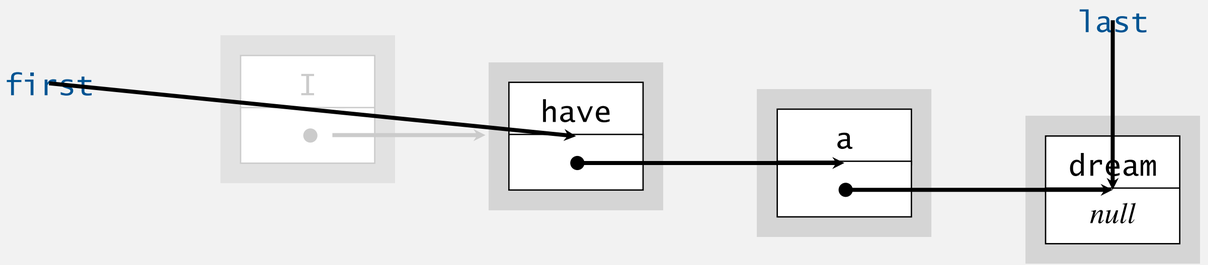


save item to return

```
String item = first.item;
```

delete first node

```
first = first.next;
```



return saved item

```
return item;
```

inner class

```
private class Node
{
   private String item;
   private Node next;
}
```

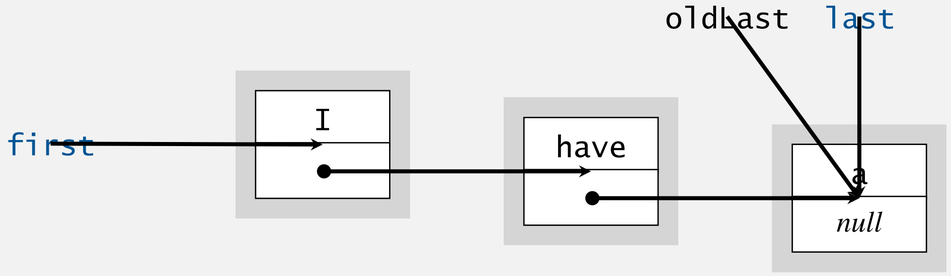
Queue enqueue: linked-list implementation

LO 2.1

add a new node to tail

save a link to the list

```
Node oldLast = last;
```

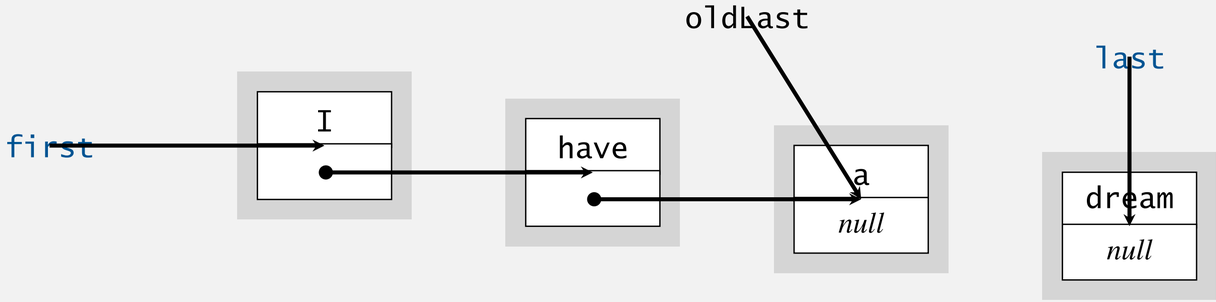


inner class

```
private class Node
{
   private String item;
   private Node next;
}
```

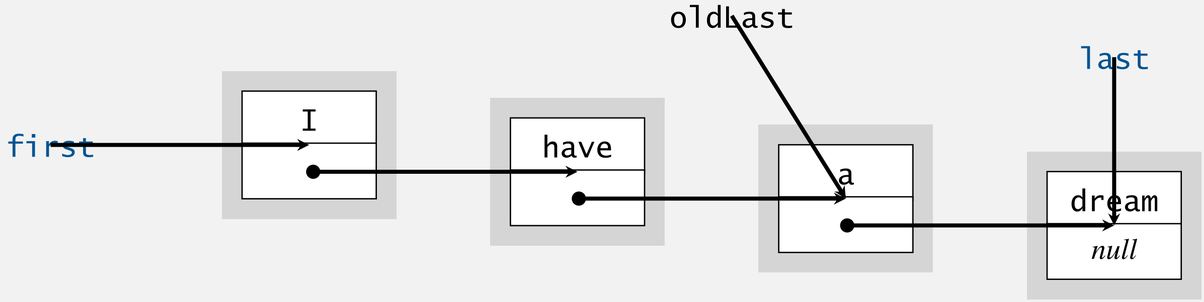
create a new node at the end

```
last = new Node();
last.item = "dream";
```



link together

```
oldLast.next = last;
```



Queue: linked-list implementation

LO 2.2

```
public class LinkedQueueOfStrings
{
   private Node first, last;
   private class Node
   {  /* same as in LinkedStackOfStrings */  }
   public boolean isEmpty()
   {  return first == null;  }
   public void enqueue(String item)
   {
      Node oldLast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldLast.next = last;
   }

   public String dequeue()
   {
      String item = first.item;
      first       = first.next;
      if (isEmpty()) last = null;
      return item;
   }
}
```

0 node

special cases

corner cases to deal with empty queue

null

0 node

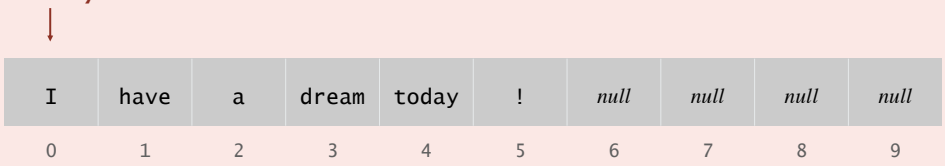
1 node

Stacks and queues: quiz

LO 2.1

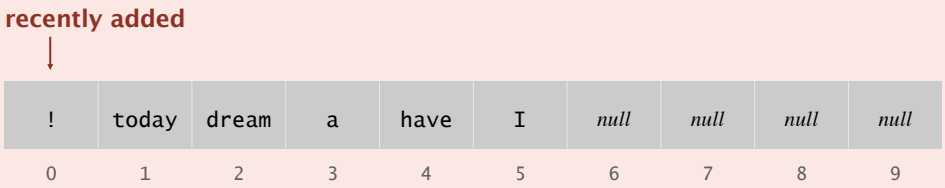
How to implement a fixed-capacity queue with an array?

A. least recently added



B.

most recently added



C. *Both A and B.*

D. Neither A nor B.

Queue: resizing-array implementation

LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.

Q. How to resize?

https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/ResizingArrayQueue.java

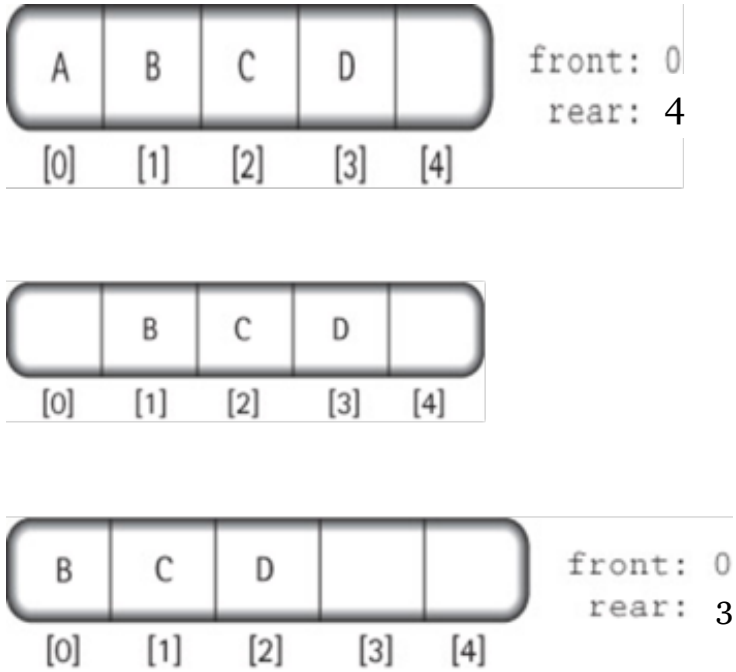
Queue Implementation

- Array-based implementation
- Must remember the “rear” and the “front” position
- Must keep track of the “current size”

```
public class Queue<E>
{
   private E[] elements;
   private int front, rear, count;
   ....
}
```

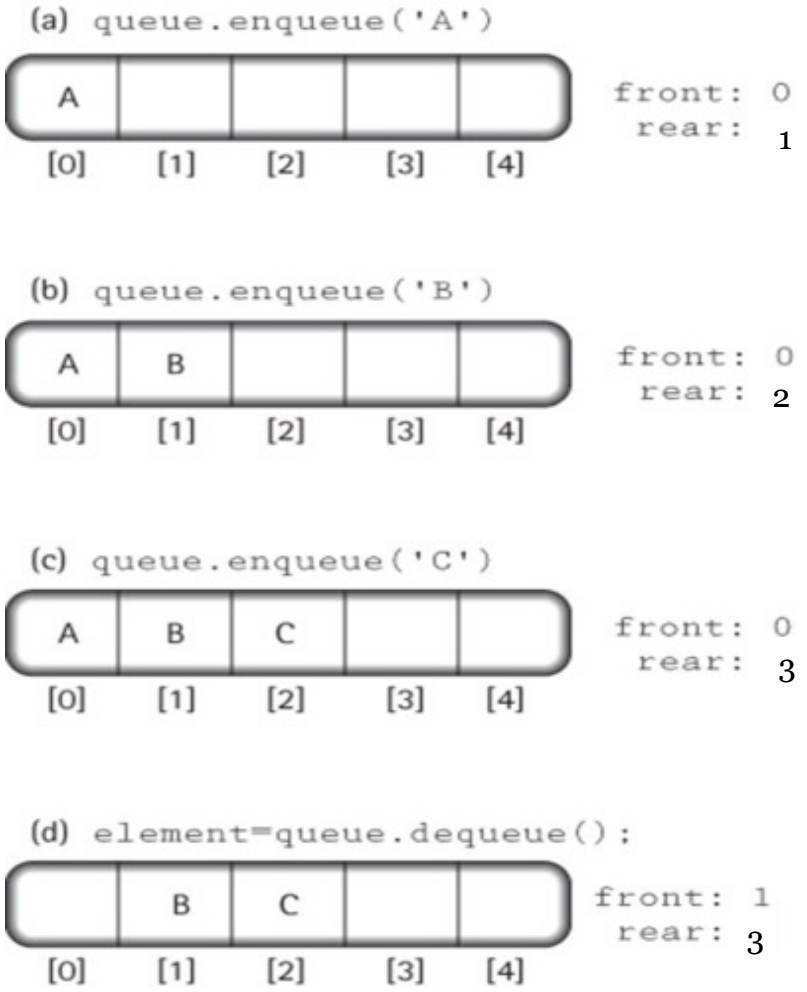
Queue Implementation—Fixed Front design

- After four calls to enqueue with arguments ‘A’, ‘B’, ‘C’, and ‘D’:
- dequeue the front element
- Move every element in the queue up one slot
- The dequeue operation is inefficient, so we do not use this approach



Floating Front Design

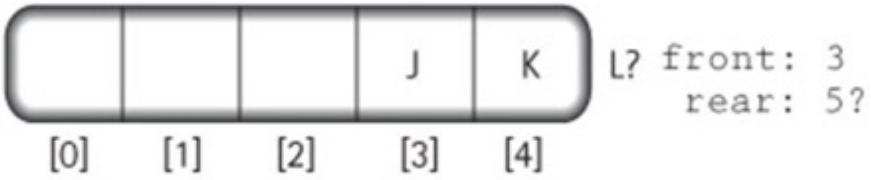
- enqueue(), add 1 to rear
- dequeue(), add 1 to front
- Trouble when hit the end of the array



Wrap around with Floating Front design

- Make the array circular
- Front and rear move forward in a circular fashion
- Works fine except that an empty and full queue both have front == rear
- Solve that by adding a count

(a) There is no room at the end of the array



(b) Using the array as a circular structure, we can wrap the queue around to the beginning of the array

A General Queue Class

```
public class Queue
{
   private Object [] elements;
   private int front, rear, count;

   public Queue ( int size )
   {
      elements = new Object[size];
      front = rear = count = 0;
   }
   ....
}
```

enqueue (add) operation

```
public void enqueue ( Object x )
{
    elements[rear] = x;
    rear = (rear + 1) % elements.length;
    count++;
}
```

wrap around the index

dequeue (remove)

```
public Object dequeue()
{
   Object x = elements[front];
   front = (front + 1) % elements.length;
   count--;
   return x;
}
```

wrap around the index

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

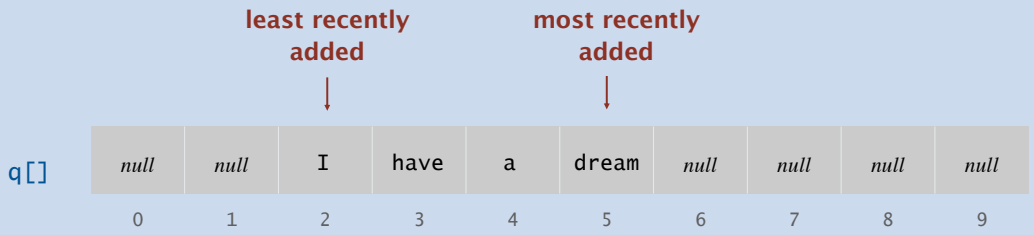
Goal. Implement a queue using a resizing array so that, starting from an empty queue, any sequence of any sequence of m enqueue and dequeue operations takes $O(m)$ time.

Need to be careful to copy queue items to beginning of queue of twice the size (wrap-around could be an issue)

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

Goal. Implement a queue using a resizing array so that, starting from an empty queue, any sequence of any sequence of m enqueue and dequeue operations takes $O(m)$ time.



QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.

enqueue today

`q[]`

null	*null*	I	have	a	dream	*null*	*null*	*null*	*null*
0	1	2	3	4	5	6	7	8	9

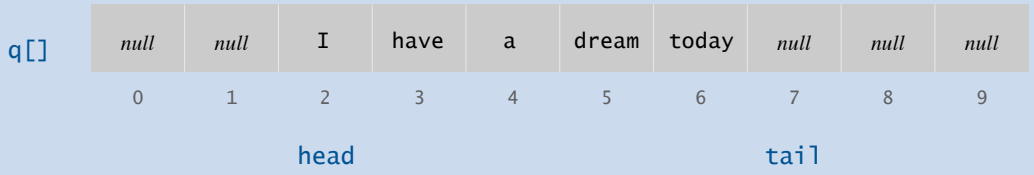
head tail

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.

enqueue !

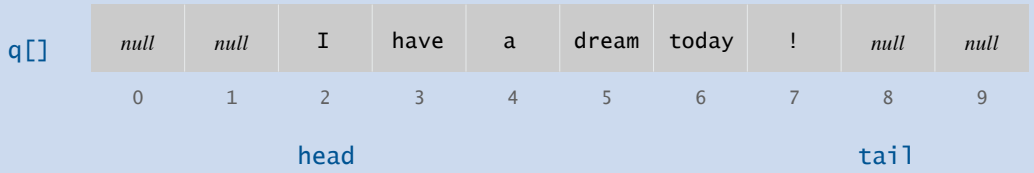


QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.

dequeue

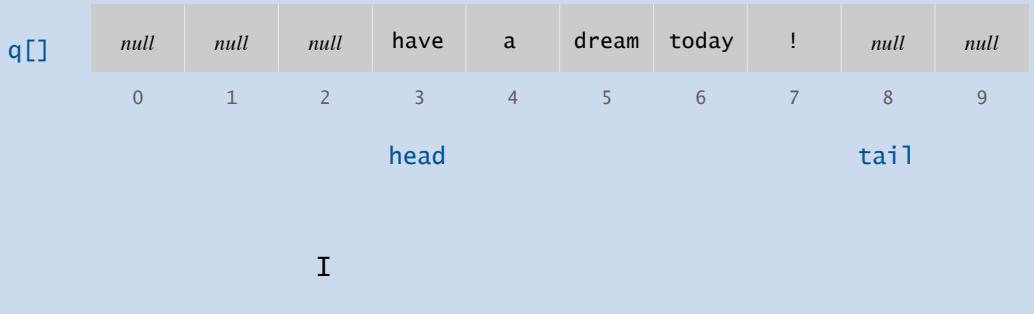


QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.

dequeue

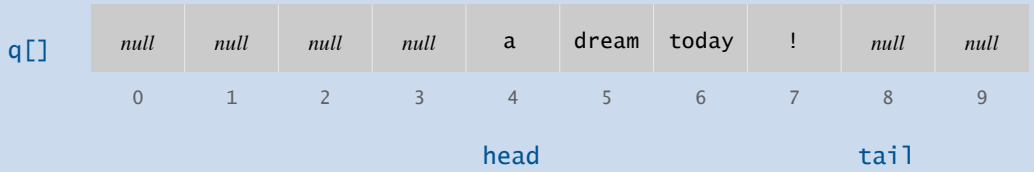


QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.

enqueue I



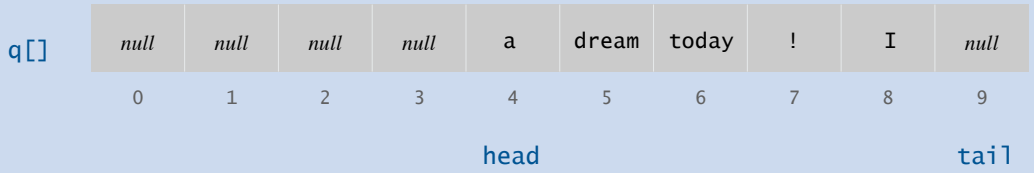
have

QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.

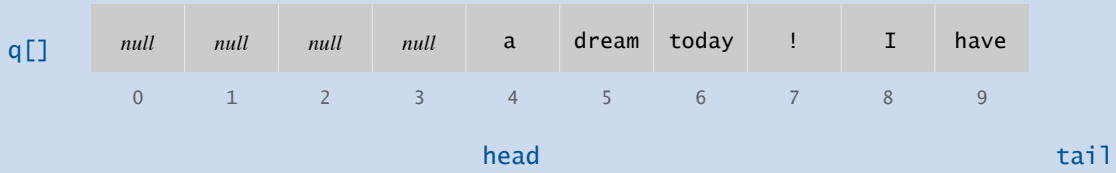
enqueue have



QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.

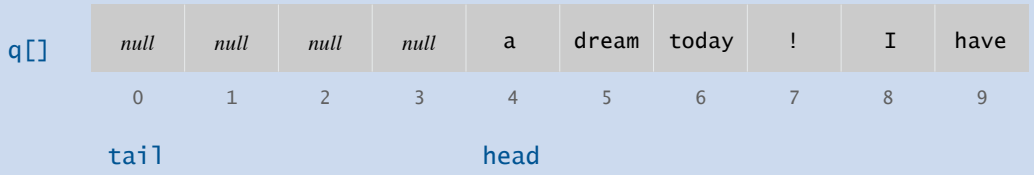


QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.

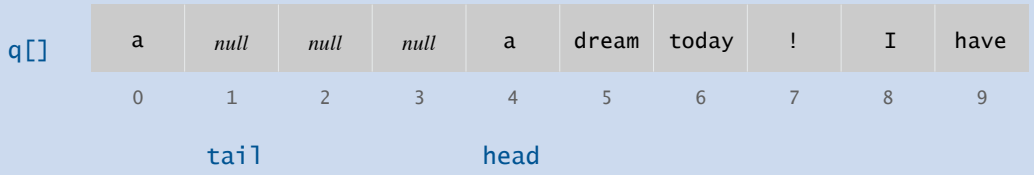
enqueue a



QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.



QUEUE: RESIZING-ARRAY IMPLEMENTATION

LO 2.1

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.

Q. How to resize?

`q[]`

a	*null*	*null*	*null*	a	dream	today	!	I	have
0	1	2	3	4	5	6	7	8	9

`tail` `head`

Stack Implementation Summary

LO 2.9

	Pop()	Push()	isEmpty()	Size()	notes
Fixed Array	O(1)	O(1)	O(1)*	O(1)*	Stack size is limited
Resizable Array	O(1)	O(1) on average**	O(1)*	O(1)*	Stack size is unlimited
Linked List	O(1)	O(1)	O(1)*	O(1)*	Stack size is unlimited

- * Assumes O(1) extra memory to hold the number of elements in the stack
- ** based on amortized analysis (see videos/book for details)

Queue Implementation Summary

LO 2.9

	enqueue()	dequeue()	isEmpty()	Size()
Fixed Array	O(1)*	O(1)*	O(1)**	O(1)**
Resizable Array	O(1)	O(1) on average***	O(1)**	O(1)**
Linked List	O(1)	O(1)	O(1)**	O(1)**

- * Assumes pointers to head and tail (circular array)
- ** Assumes O(1) memory to store size of the queue
- *** based on amortized analysis (see videos/book for details)

Implementation Decisions

Arrays and Linked List are some of the most fundamental memory organization methods.

Arrays. Organizes memory as a contiguous block of memory

Linked List. Organizes a linked collection of memory blocks that are drawn from free heap space.

Question. When is it appropriate to select one or the other

Answer. In general,

- if the application stack/queue space requirements are known in advance, an array is a sufficient structure.
  - E.g., use a stack to evaluate a postfix expression
- If the application stack/queue space requirements are not known in advance, it is appropriate to use a ~~queue~~ linked list structure.
  - Use a queue to process data packets flowing through a hub, where data traffic can vary based on the time of the day.

in a router on the Internet

2. Stacks and Queues

- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics (optional)*
- *Iterators (optional)*
- *Applications (optional)*

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfInts`, `StackOfApples`, `StackOfOranges`, ...

Solution in Java: generics.

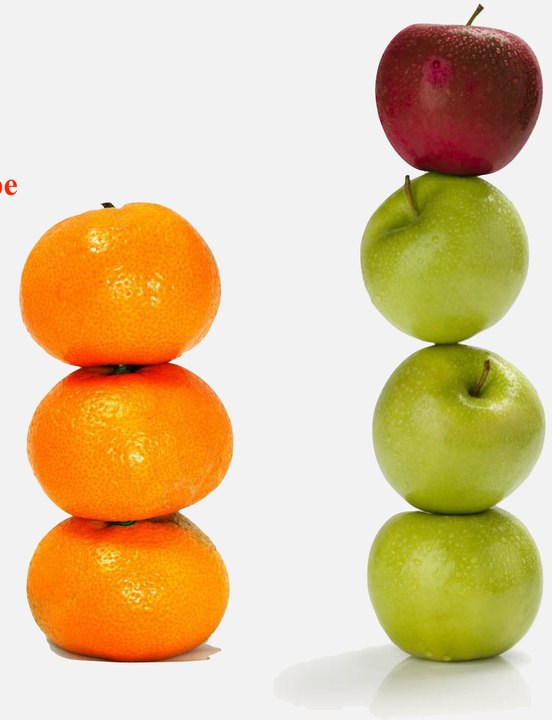
type parameter

(use syntax both to specify type and to call constructor)

compiler will check the type

```
Stack<Apple> stack = new Stack<Apple>();
Apple apple = new Apple();
Orange orange = new Orange();
stack.push(apple);
stack.push(orange);
...
```

compile-time error



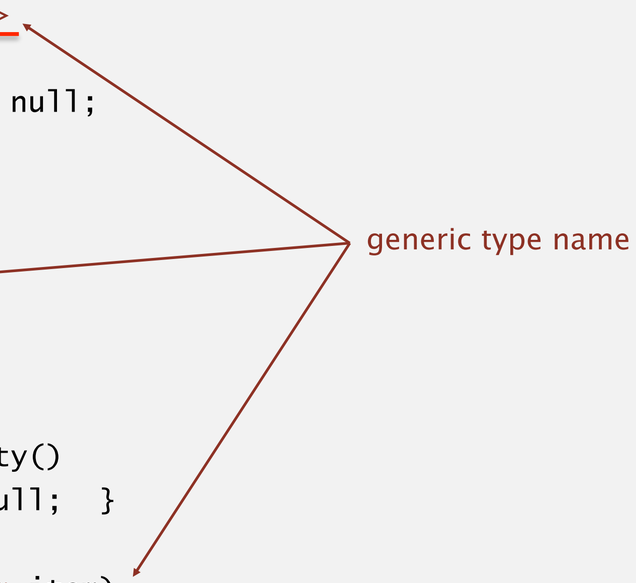
Generic stack: linked-list implementation

stack of strings (linked list)

```
public class LinkedStackOfStrings
{
   private Node first = null;
   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

generic stack (linked list)

```
public class Stack<Item>
{
   private Node first = null;

   private class Node
   {
      Item item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(Item item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public Item pop()
   {
      Item item = first.item;
      first = first.next;
      return item;
   }
}
```



Generic stack: array implementation

The way it should be.

stack of strings (fixed-length array)

```
public class FixedCapacityStackOfStrings
{
   private String[] s;
   private int n = 0;

   public …StackOfStrings(int capacity)
   {  s = new String[capacity];  }

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

   public void push(String item)
   {  s[n++] = item;  }

   public String pop()
   {  return s[--n];  }
}
```

Generic stack of strings (fixed-length array)

```
public class FixedCapacityStack<Item>
{
   private Item[] s;
   private int n = 0;

   public FixedCapacityStack(int capacity)
   {  s = new Item[capacity];  }

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

   public void push(Item item)
   {  s[n++] = item;  }

   public Item pop()
   {  return s[--n];  }
}
```

@#$*! generic array creation not allowed in Java

Generic stack: array implementation

The way it should be.

stack of strings (fixed-length array)

```
public class FixedCapacityStackOfStrings
{
   private String[] s;
   private int n = 0;

   public …StackOfStrings(int capacity)
   {  s = new String[capacity];  }

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

   public void push(String item)
   {  s[n++] = item;  }

   public String pop()
   {  return s[--n];  }
}
```

generic stack (fixed-length array)

```
public class FixedCapacityStack<Item>
{
   private Item[] s;
   private int n = 0;

   public FixedCapacityStack(int capacity)
   {  s = (Item[]) new Object[capacity]; }

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

   public void push(Item item)
   {  s[n++] = item;  }

   public Item pop()
   {  return s[--n];  }
}
```

the ugly cast

```
public class FixedCapacityStack<E> {
    private E[] s;
    private int n;

    public FixedCapacityStack(int capacity)
    {
       s = (E[]) new Object[capacity];
    }
   ...
}
```

Unchecked cast

```
~/Desktop/queues> javac -Xlint:unchecked FixedCapacityStack.java
FixedCapacityStack.java:26: warning: [unchecked] unchecked cast
        s = (Item[]) new Object[capacity];
                     ^
  required: Item[]
  found:    Object[]
  where Item is a type-variable:
    Item extends Object declared in class FixedCapacityStack
1 warning
```

Q. Why does Java require a cast (or reflection)?

Short answer. Backward compatibility.

Long answer. Need to learn about type erasure and covariant arrays.

Stacks and queues: quiz 5

Which of the following is a correct way to declare and initialize an empty stack of integers?

A. `Stack stack = new Stack<int>();`

B. `Stack<int> stack = new Stack();`

C. `Stack<int> stack = new Stack<int>();`

D. *None of the above.*

CANNOT use primitive types

need a reference type in the bracket
e.g. Integer (the wrapper class for int)

Generic data types: autoboxing and unboxing

Q. What to do about primitive types?

Wrapper type.

- Each primitive type has a "wrapper" reference type.
- Ex: `Integer` is wrapper type for `int`.

Autoboxing. Automatic cast from primitive type to wrapper type.

Unboxing. Automatic cast from wrapper type to primitive type.

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);    // autoboxing: stack.push(Integer.valueOf(17));
int a = stack.pop();     // int a = stack.pop().intValue();
                         // auto-unboxing
```

Bottom line. Client code can use generic stack for any type of data.

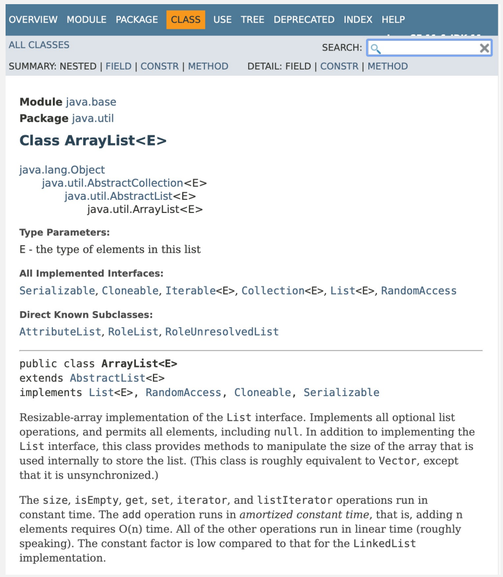
(but substantial overhead for primitive types)

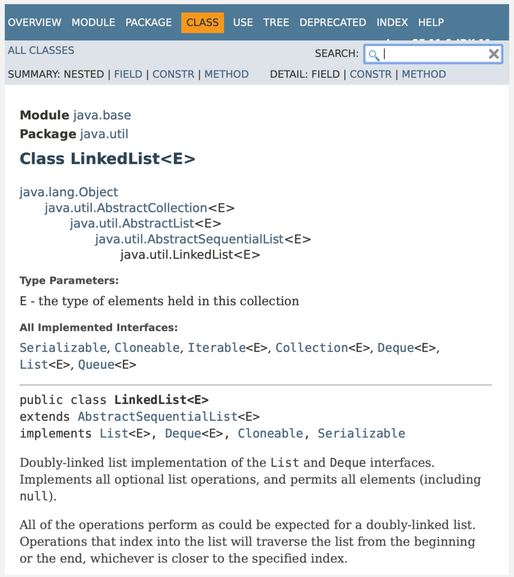
Java collections framework

Java's library of collection data types.

- `java.util.ArrayList` [resizing array] — **Initial capacity is 10**
- `java.util.LinkedList` [doubly linked list]
- `java.util.ArrayDeque` **double-ended Queue**

This class is likely to be faster than Stack when used as a stack, and faster than LinkedList when used as a queue.





This course. Implement from scratch (once).

Beyond. Basis for understanding performance guarantees.

Best practices.

- Use our `Stack` and `Queue` for stacks and queues to improve design and efficiency.
- Use Java's `ArrayList` or `LinkedList` when other ops needed (but remember that some ops are inefficient).

Stacks and queues summary

Fundamental data types.

- Value: collection of objects.
- Operations: add, remove, iterate, test if empty

Stack. Examine the item most recently added (LIFO).

Queue. Examine the item least recently added (FIFO).

Efficient implementations.

- Singly linked list.
- Resizing array.

Next time. Advanced Java (including iterators for collections).

2. Stacks and Queues

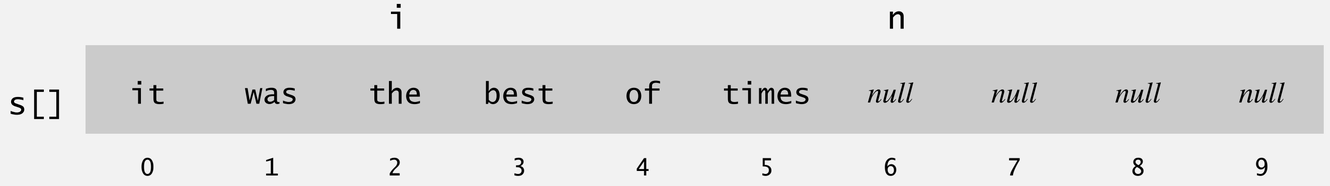
- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics*
- *Iterators (optional)*
- *Applications (optional)*

Iteration

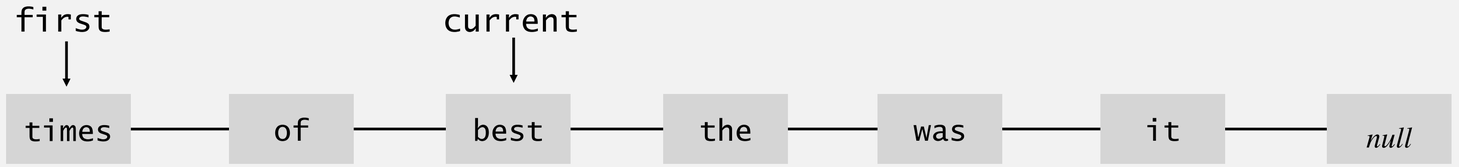
Design challenge. Allow client to iterate over the stack items, without exposing the internal representation of the stack.

traversing the list

resizing-array representation



linked-list representation



Java solution. Use a foreach loop.

Foreach loop

Java provides elegant syntax for iterating over items in a collection.

"foreach" loop (shorthand)

```
Stack<String> stack;
...

for (String s : stack)
    ...
```

equivalent code (longhand)

```
Stack<String> stack;
...

Iterator<String> i = stack.iterator();
while (i.hasNext())
{
   String s = i.next();
}  ...
}
```

To make user-defined collection support foreach loop:

- Data type must have a method named `iterator()`.
- The `iterator()` method returns an object that has two core method:
  - the `hasNext()` methods returns `false` when there are no more items
  - the `next()` method returns the next item in the collection

Iterators

To support foreach loops, Java provides two interfaces.

- `Iterator` interface: `next()` and `hasNext()` methods.
- `Iterable` interface: `iterator()` method that returns an `Iterator`.
- Both should be used with generics.

java.util.Iterator interface

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
}
```

must implement these 2 methods

java.lang.Iterable interface

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

must implement this method

Type safety.

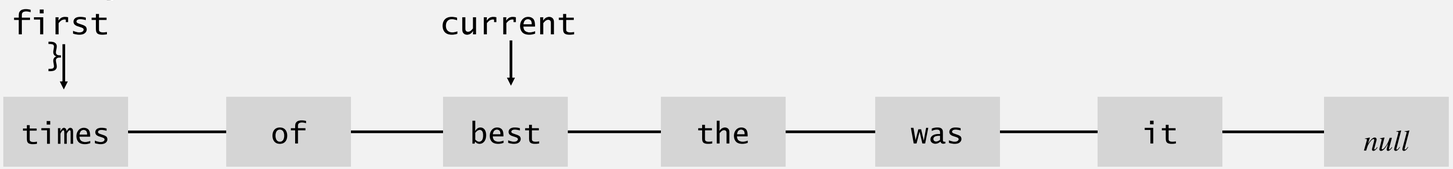
- Implementation must use these interfaces to support foreach loop.
- Client program won't compile unless implementation do.

Stack iterator: linked-list implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

   public Iterator<Item> iterator() { return new LinkedIterator(); }

   private class LinkedIterator implements Iterator<Item>
   {
      private Node current = first;

      public boolean hasNext()  {  return current != null;  }

      public Item next()
      {
         Item item = current.item;
         current   = current.next;
         return item;
      }
   }
}
```

inner class

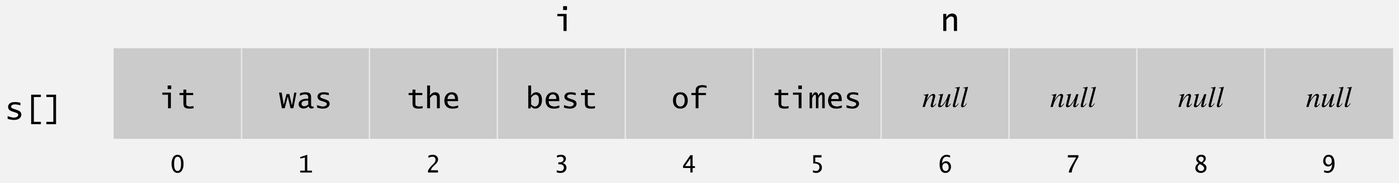
throw NoSuchElementException if no more items in iteration



Stack iterator: array implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

   public Iterator<Item> iterator()
   { return new ReverseArrayIterator(); }

   private class ReverseArrayIterator implements Iterator<Item>
   {
      private int i = n;

      public boolean hasNext() {  return i > 0;   }
      public Item next()       {  return s[--i];  }
   }

}
```

inner class



ITERATION: CONCURRENT MODIFICATION

Q. What if client modifies the data structure while iterating?

A. A fail-fast iterator throws a `java.util.ConcurrentModificationException`.

If the data structure is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will generally throw a ConcurrentModificationException.

concurrent modification

```
for (String s : stack)
   stack.push(s);
```

Q. How to detect concurrent modification?

A.

concurrency is a complex problem

Java class Vector is synchronized, it is the superclass of Stack

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

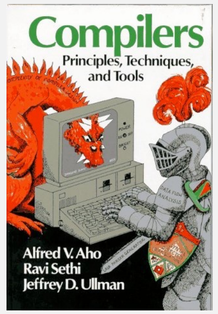
Rutgers University

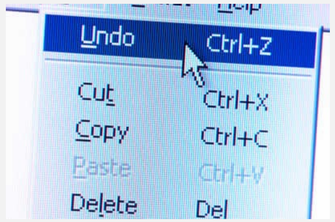
2. Stacks and Queues

- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics*
- *Iterators (optional)*
- *Applications*

Stack applications

- Java virtual machine.
- Parsing in a compiler.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Function-call stack during execution of a program.
- ...





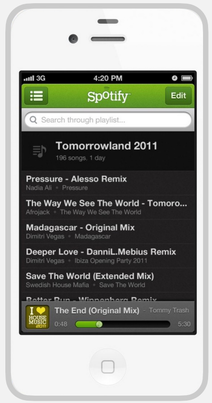
Queue applications

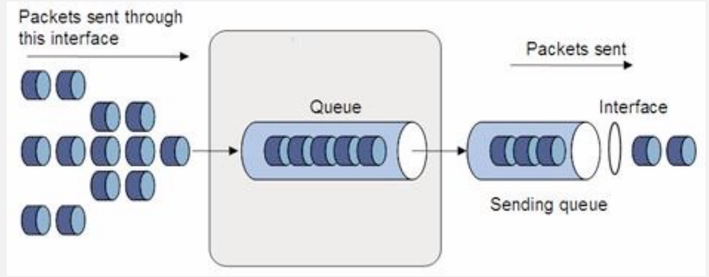
Familiar applications.

- Spotify playlist.
- Data buffers (iPod, TiVo, sound card, streaming video, …).
- Asynchronous data transfer (file I/O, pipes, sockets, …).
- Dispensing requests on a shared resource (printer, processor, …).

Simulations of the real world.

- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.





Java collections library

Optional content

List interface. `java.util.List` is API for a sequence of items.

`public interface List<Item> extends Iterable<Item>`

	`List()`	*create an empty list*
`boolean`	`isEmpty()`	*is the list empty?*
`int`	`size()`	*number of items*
`void`	`add(Item item)`	*add item to the end*
`Iterator<Item>`	`iterator()`	*iterator over all items in the list*
`Item`	`get(int index)`	*return item at given index*
`Item`	`remove(int index)`	*return and delete item at given index*
`boolean`	`contains(Item item)`	*does the list contain the given item?*
	⋮	

Implementations. `java.util.ArrayList` uses a resizing array;
`java.util.LinkedList` uses a doubly linked list.

Caveat: not all operations are efficient!

Java collections library

`java.util.Stack`.

- Supports `push()`, `pop()`, and iteration.
- Inherits from `java.util.Vector`, which implements `java.util.List` interface.

Java 1.3 bug report (June 27, 2001)

```
The iterator method on java.util.Stack iterates through a Stack from
the bottom up. One would think that it should iterate as if
it were popping off the top of the Stack.
```

status (closed, will not fix)

```
It was an incorrect design decision to have Stack extend Vector ("is-
a" rather than "has-a"). We sympathize with the submitter
but cannot fix this because of compatibility.
```

Java collections library

`java.util.Stack`.

- Supports `push()`, `pop()`, and iteration.
- Inherits from `java.util.Vector`, which implements `java.util.List` interface.

`java.util.Queue`. An interface, not an implementation of a queue.

Best practices. Use our `Stack` and `Queue` for stacks and queues;
use `java.util.ArrayList` or `java.util.LinkedList` when appropriate.

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

STACKS AND QUEUES

- *Stacks*
- *Resizing arrays*
- *Queues*
- *Generics*
- *Iterators (optional)*
- *Applications (optional)*

https://ds.cs.rutgers.edu

Slides Adopted and modified from Algorithms by Sedgewick and Wayne

Last updated on 1/8/21