Stacks, Queues, Deques

- Abstract Data Types
- Stacks
- Queues
- Deques

Abstract Data Types (ADT)

An Abstract Data Type is an abstraction of a data structure.

The ADT specifies:

- what can be stored in the ADT
- what operations can be done on/by the ADT.
- It specifies what each operation does, but NOT how it does it

In Java an ADT can be expressed by an interface.

An ADT is realized/implemented by a concrete data structure.

A concrete data structure implements an ADT in the same way as, in Java, a class implements an interface.

Abstract Data Types (ADTs)

- Specify precisely the operations that can be performed

-The implementation is HIDDEN and can easily change

EXAMPLE

Phone Book ADT:

Objects of type: Phone Book Entry (name, phone, e-mail)

Operations: find, add, remove

Stacks, Queues, and Deques

ADT Stack

- Implementation with Arrays
- Implementation with Singly Linked List

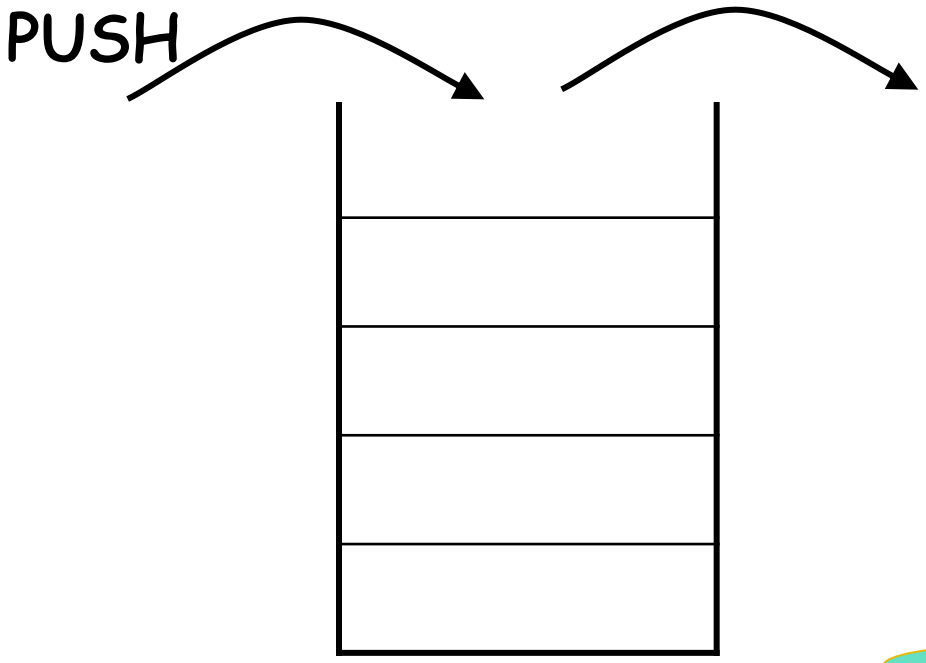
ADT Queue

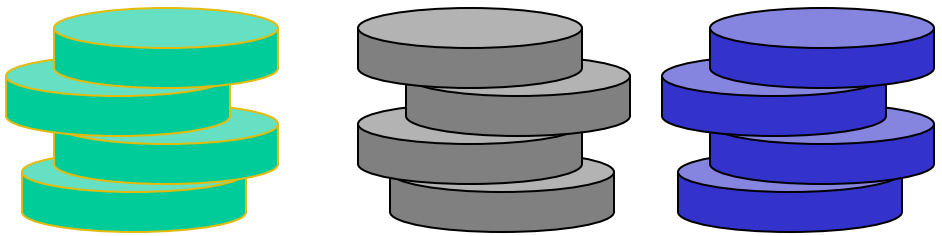
- Implementation with Arrays
- Implementation with Singly Linked List

ADT Double Ended Queues

- Implementation with doubly Linked List

Stacks





The Stack Abstract Data Type

- Main methods:
  - push(element): Inserts element onto top of stack
  - object pop(): Removes and returns the last element inserted (the one on the top); if the stack is empty, returns null

- Support methods:
  - integer size(): Returns the number of elements stored in stack
  - boolean isEmpty(): Returns a boolean indicating if stack is empty.
  - object top(): Returns the top element of the stack, without removing it; if the stack is empty, returns null

ADT for Stack: our own Stack Interface in Java

- Java interface corresponding to our Stack ADT
- Assumes null is returned from top() and pop() when stack is empty
- Different from the built-in Java class java.util.Stack

```
public interface Stack<E> {
   int size();
   boolean isEmpty();
   E top();
   void push(E element);
   E pop();
}
```

Concrete data structure: our own Array-based Stack in Java

concrete data structure

```
public class ArrayStack<E>
      implements Stack<E> {
                    Stack ADT
   // holds the stack elements
   private E[ ] S;

   // index to top element
   private int top = -1;

   // constructor
   public ArrayStack(int capacity) {
       S = (E[ ]) new Object[capacity];
    }
```

```
   public E pop() {
     if isEmpty()
            return null;
       E temp = S[top];
       // facilitate garbage collection:
       S[top] = null;
       top = top – 1;
       return temp;
    }
   //other methods of Stack interface
   public int size() { return (top + 1);}
   public boolean isEmpty() { return (top== -1);}
   public void push (E element) {  /* code here*/ }
   public E top() {/* code here*/ }
}
```

Example Use in Java

```
import net.datastructures.*;// here you can find
                            // interface Stack<E> and
                            // class ArrayStack<E>

public class ReverseAlgorithms {

  public static void intReverse (Integer[] a, int num) {
    Stack<Integer> s;
    s = new ArrayStack<Integer>(num);
    for (int i=0; i<num; i++) s.push(a[i]);
    for (int i=0; i<num; i++) a[i]=s.pop();
  }

  public static void floatReverse (Float[] f, int num) {
    Stack<Float> fs;
    fs = new ArrayStack<Float>(num);
    for (int i=0; i<num; i++) fs.push(f[i]);
    for (int i=0; i<num; i++) f[i]=fs.pop();
    }
```

```
public static void main(String[] args) {

 Integer list[]={4,1,2,3};
 intReverse(list,list.length);

 //… code that prints list  …

 Float fractions[]={0f,0.25f,0.5f,0.75f,1f};
 floatReverse(fractions,fractions.length);

 //… code that prints fractions …
 }

}
```

Output:

[3, 2, 1, 4]

[1.0, 0.75, 0.5, 0.25, 0.0]

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

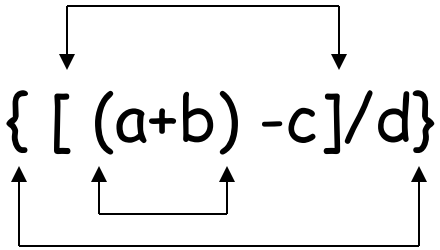
Parenthesis Matching

Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”

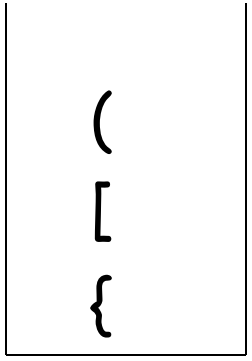
- correct: ( )(( )){([( )])}
- correct: (( ))(( )){([( )])}
- incorrect:)(( )){([( )])}
- incorrect: ({[ ])}
- incorrect: (

Parenthesis Matching

Checking for balanced parenthesis



{ [a+b) -c]/d}



{ [(a+b) -c]/d}

When we read an open parenthesis input=')'
pop ch='(' from stack and check they have matching types

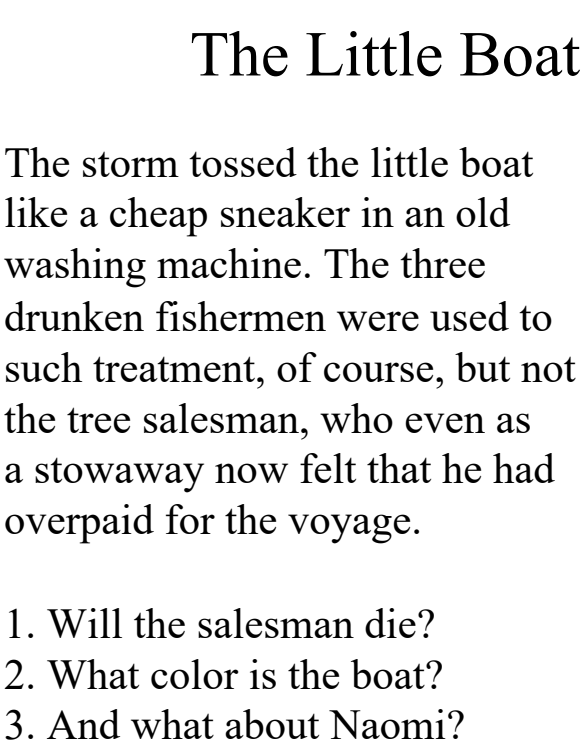
Parenthesis Matching (Java)

```
public static boolean isMatched(String expression) {
  final String opening = "({["; // opening delimiters
  final String closing = ")}]"; // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<Character>( );
  // above could use another concrete data structure: ArrayStack<Character>()
  for (char c : expression.toCharArray( )) {
    if (opening.indexOf(c) != −1) // this is a left delimiter
      buffer.push(c);
   else if (closing.indexOf(c) != −1) { // this is a right delimiter
     if (buffer.isEmpty( )) // nothing to match with
       return false;
     if (closing.indexOf(c) != opening.indexOf(buffer.pop( )))
       return false; // mismatched delimiter
     }
   }
  return buffer.isEmpty( ); // were all opening delimiters matched?
}
```

HTML Tag Matching

- For fully-correct HTML, each <name> should pair with a matching </name>

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



HTML Tag Matching (Java)

```
public static boolean isHTMLMatched(String html) {
  Stack<String> buffer =  new LinkedStack<>( );
  int j = html.indexOf('<'); // find first '<' character (if any)
  while (j != −1) {
    int k = html.indexOf('>', j+1); // find next '>' character
    if (k == −1)
      return false; // invalid tag
    String tag = html.substring(j+1, k); // strip away < >
    if (!tag.startsWith("/")) // this is an opening tag
      buffer.push(tag);
    else { // this is a closing tag
      if (buffer.isEmpty( ))
        return false; // no tag to match
      if (!tag.substring(1).equals(buffer.pop( )))
        return false; // mismatched tag
    }
    j = html.indexOf('<', k+1); // find next '<' character (if any)
  }
  return buffer.isEmpty( ); // were all opening tags matched?
}
```

Implementing a Stack with an Array

The stack consists of an N-element array S and an integer variable t, the index of the top element in array S.


```
Algorithm size():
        return t +1
Algorithm isEmpty():
        return (t < 0)
Algorithm top():
        if isEmpty() then return null
        return S[t]
```

```
Algorithm push(obj):
        if size() = N then
            ERROR
        t ← t + 1
        S[t] ← obj
```

```
Algorithm pop():
        if isEmpty() then
            return null
        e ← S[t]
        S[t] ← null
        t ← t-1
        return e
```

Performance and Limitations

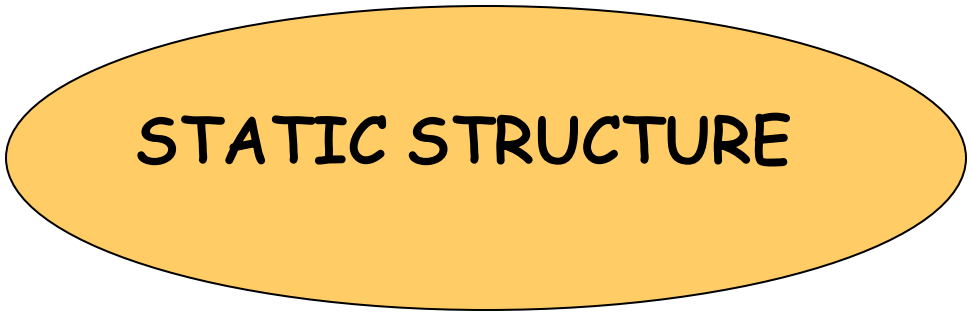
- Performance

	Time
size()	O(1)
isempty()	O(1)
top()	O(1)
push(obj)	O(1)
pop()	O(1)

Space: O(N)

N = size of the Array

- Limitations

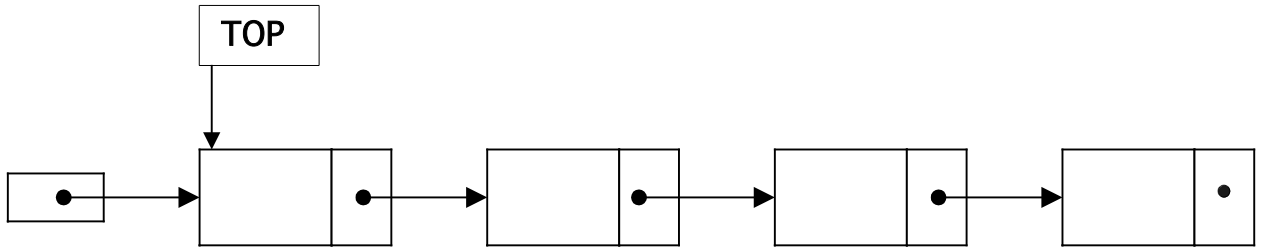


Stacks with Extendible Arrays

- Stacks can be implemented with extendible arrays.

- We will study extendible arrays when we learn the List ADT.

Implementing a Stack with a Singly Linked List

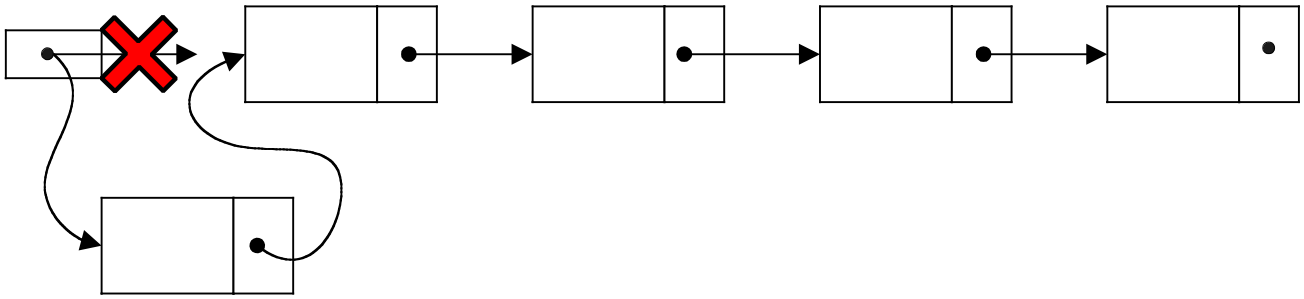


size = 4

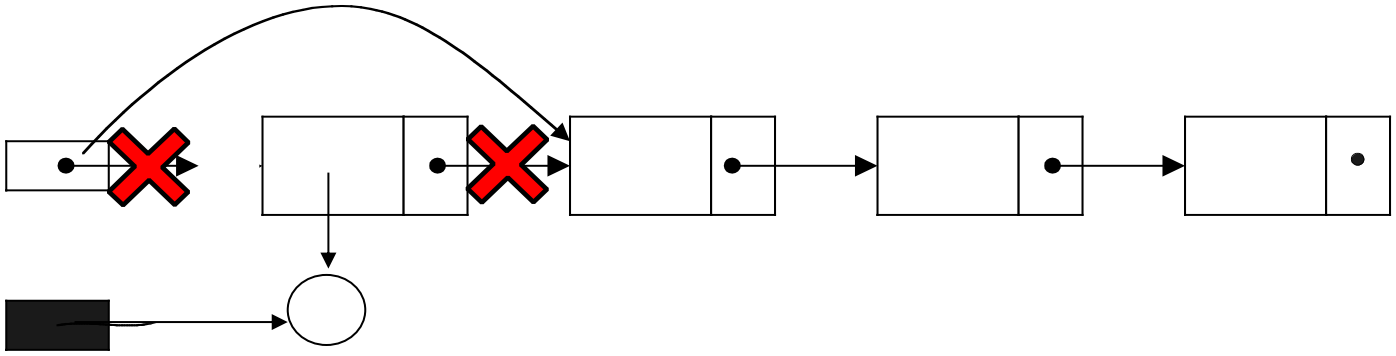
-Singly linked list plus a variable containing the current size of the list

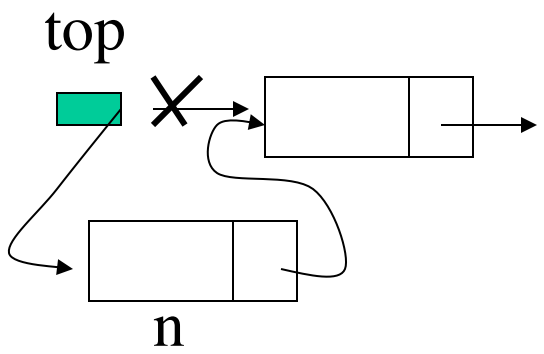
DYNAMIC STRUCTURE

PUSH: Add at the front

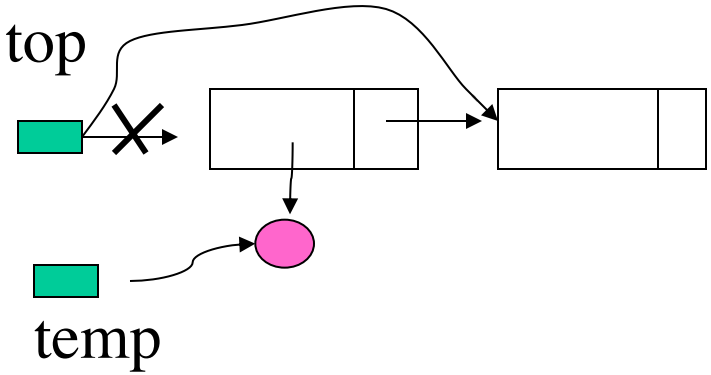


POP: Take the first






```
Algorithm push(obj):
    n ← new Node
    n.setElement(obj)
    n.setNext(top)
    top ← n
    size++
```




```
Algorithm pop():
  if isEmpty() then
                ERROR
  temp ← top.getElement()
  top ←  top.getNext()
  size- -
  return temp
```

Performance

Time:

size()	O(1)
isEmpty()	O(1)
top()	O(1)
push(obj)	O(1)
pop()	O(1)

Space: Variable

Slide by Matt Stallmann
included with permission.

Evaluating Arithmetic Expressions

$14 - 3 * 2 + 7 = (14 - (3 * 2)) + 7$

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group
evaluated from left to right
Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and perform higher and *equal* precedence operations.

Algorithm for Evaluating Expressions

Slide by Matt Stallmann included with permission.

Two stacks:

- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

Algorithm doOp()

```
x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )
```

Algorithm repeatOps(refOp):

```
while ( valStk.size() > 1 ∧
        prec(refOp) ≤
        prec(opStk.top())
    doOp()
```

Algorithm EvalExp()

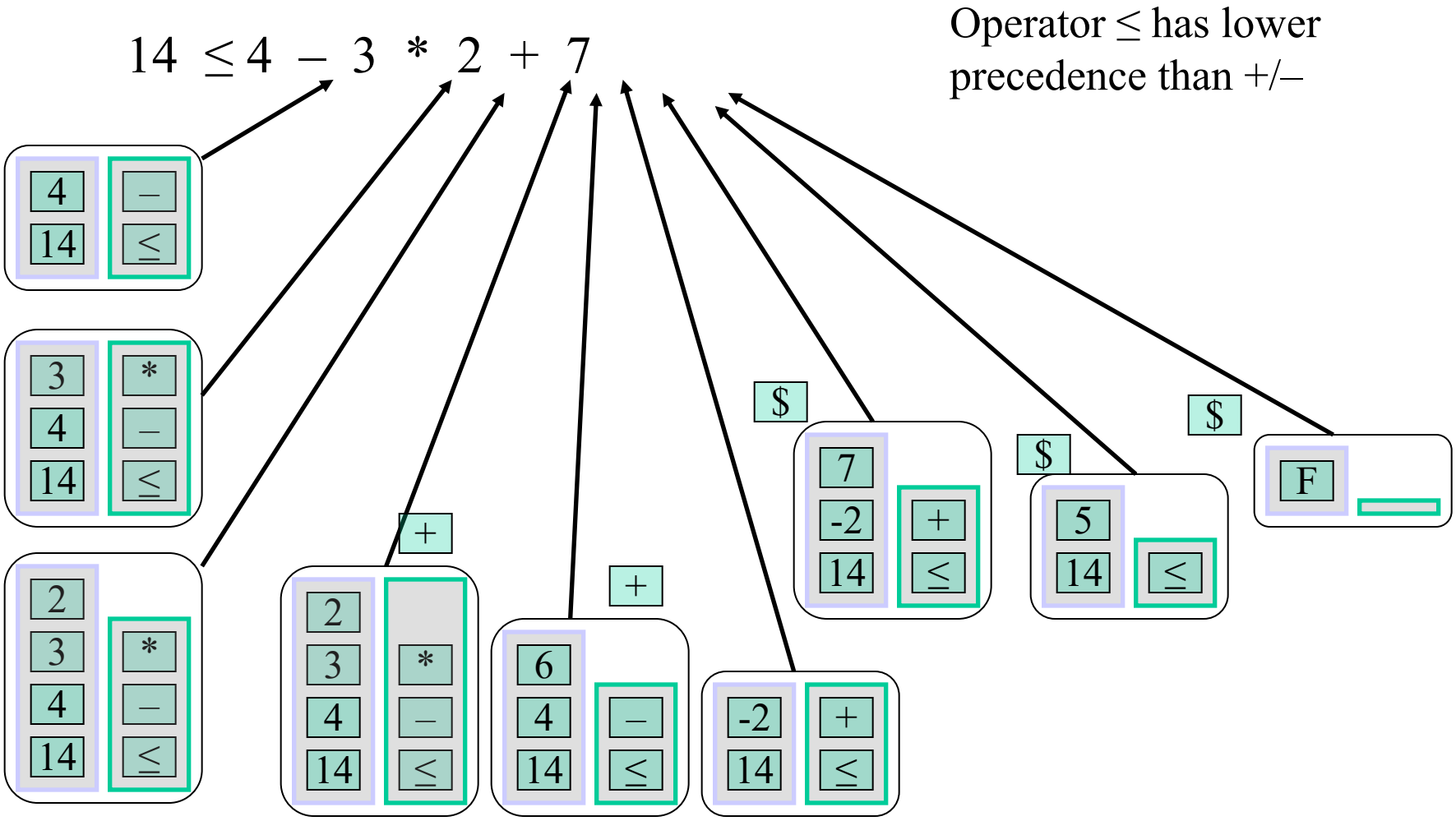
Input: a stream of tokens representing an arithmetic expression (with numbers)

Output: the value of the expression

```
while there's another token z
    if isNumber(z) then
        valStk.push(z)
    else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

Algorithm on a Example Expression

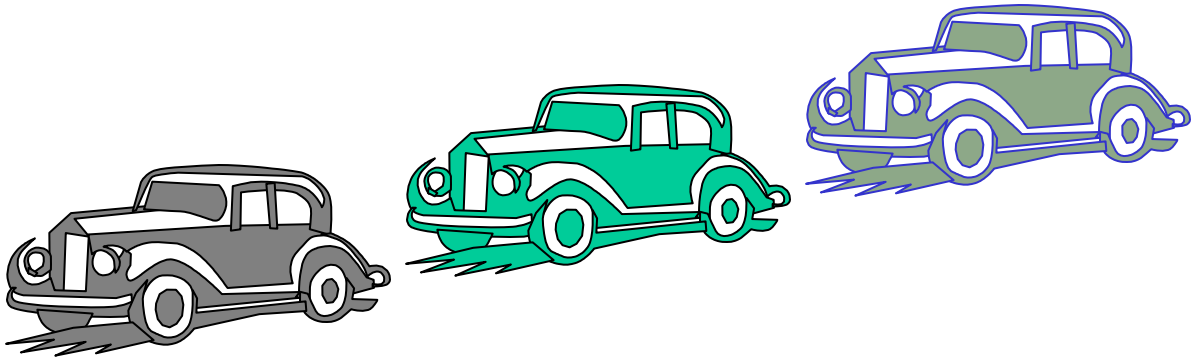
Slide by Matt Stallmann included with permission.



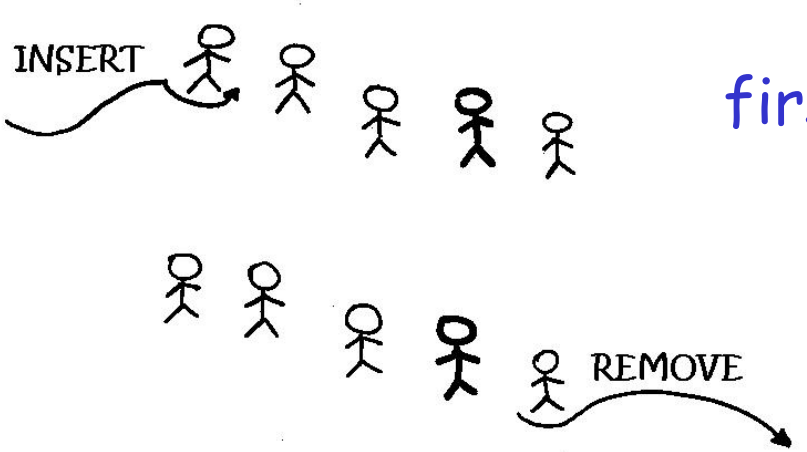
Stacks, Queues, Deques

- Abstract Data Types
- Stacks
- Queues
- Deques

The Queue



The Queue



first-in-first-out (FIFO)

Elements are inserted at the **rear** (enqueued) and removed from the **front** (dequeued)

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

The Queue Abstract Data Type

- Fundamental methods:
  - **enqueue(o):** Insert object o at the rear of the queue
  - **dequeue():** Remove the object from the front of the queue and return it; if the queue is empty return null.

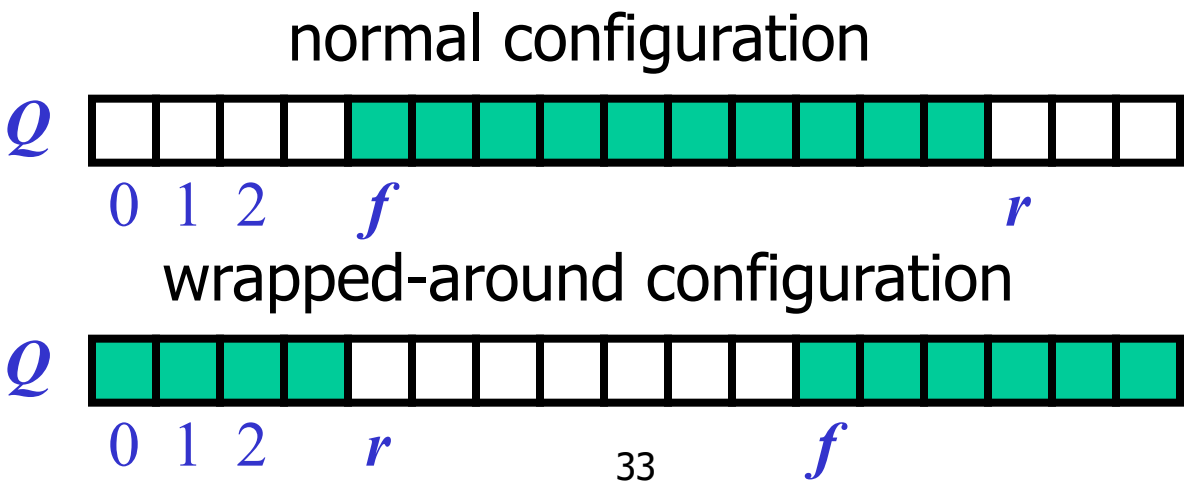
- Support methods:
  - **size():** Return the number of objects in the queue
  - **isEmpty():** Return a boolean value that indicates whether the queue is empty
  - **first():** Return, but do not remove, the front object in the queue; if the queue is empty return null.

Example

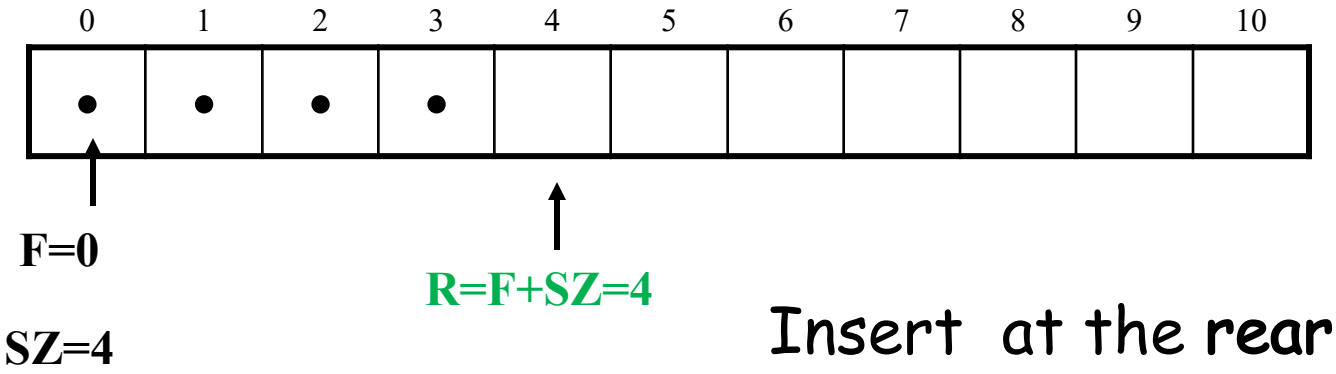
Operation		*Output*	*Q*
enqueue(5)	–	(5)	
enqueue(3)	–	(5, 3)	
dequeue()	*5*	(3)	
enqueue(7)	–	(3, 7)	
dequeue()	*3*	(7)	
first()	*7*	(7)	
dequeue()	*7*	()	
dequeue()	*null*	()	
isEmpty()	*true*	()	
enqueue(9)	–	(9)	
enqueue(7)	–	(9, 7)	
size*()*	*2*	(9, 7)	
enqueue(3)	–	(9, 7, 3)	
enqueue(5)	–	(9, 7, 3, 5)	
dequeue()	*9*	(7, 3, 5)	

Array-based Queue

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



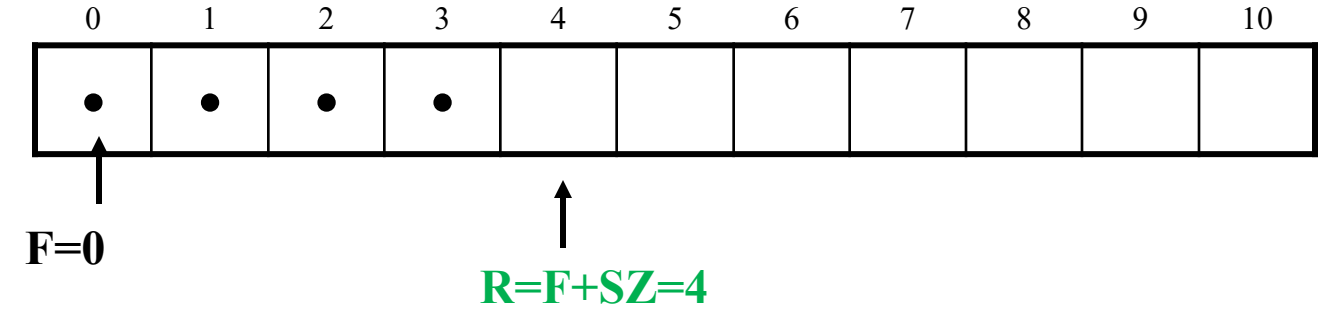
Implementing a Queue with an Array

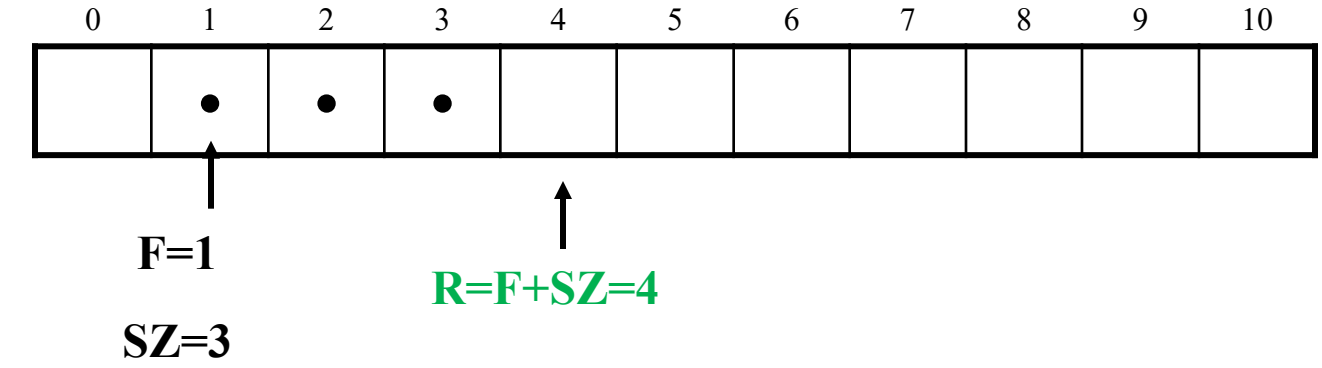


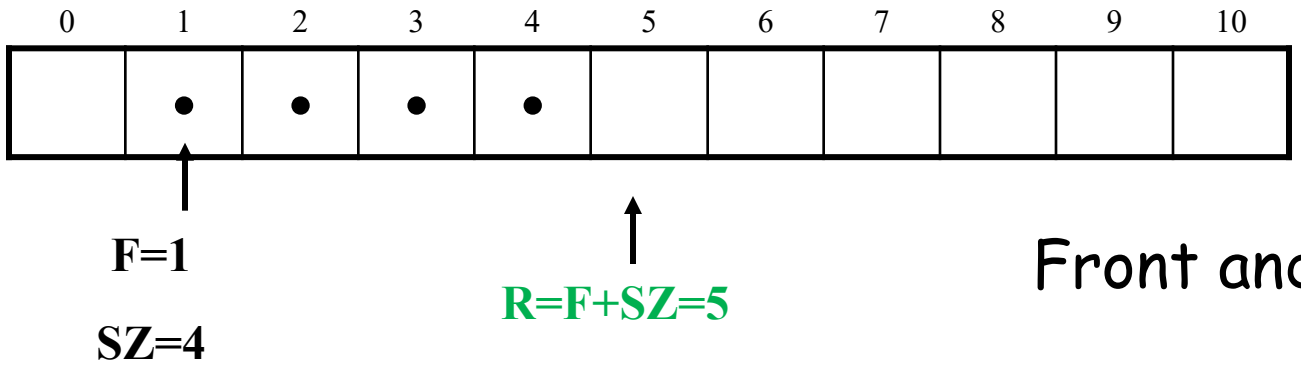
Insert at the **rear** and remove from **front**

Remove: F ← F+1; SZ ← SZ-1

Insert: Insert at F+SZ (=R);
SZ ← SZ+1 (note that R=R+1)

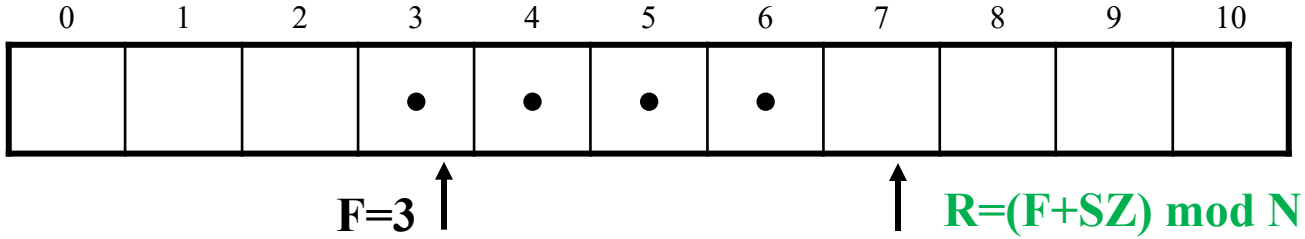






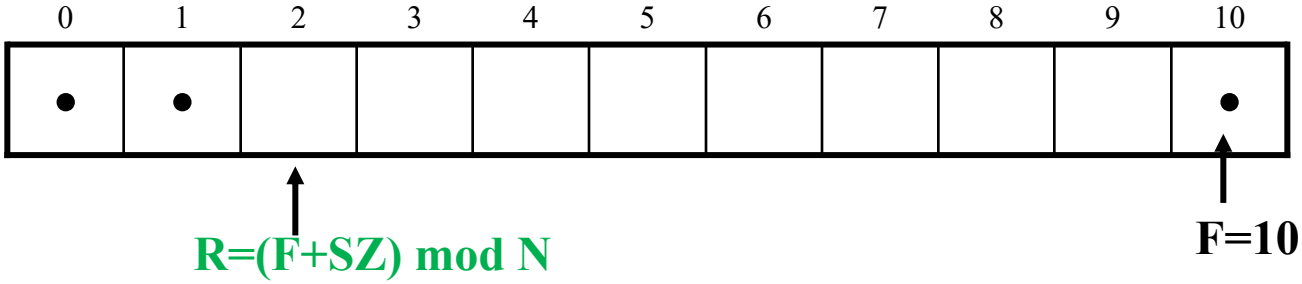
Front and rear move …

Implementing a Queue with an Array

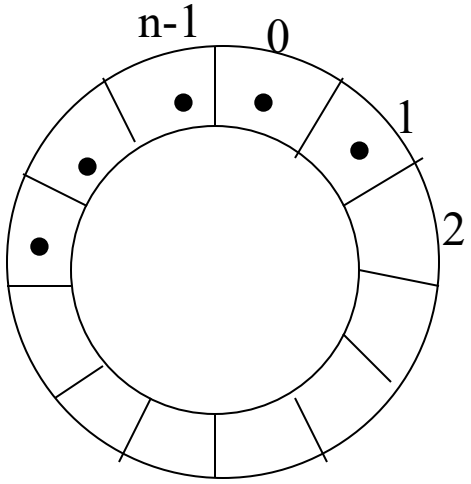


Fixed array size N=11

SZ=4



SZ=3



Remove: at F

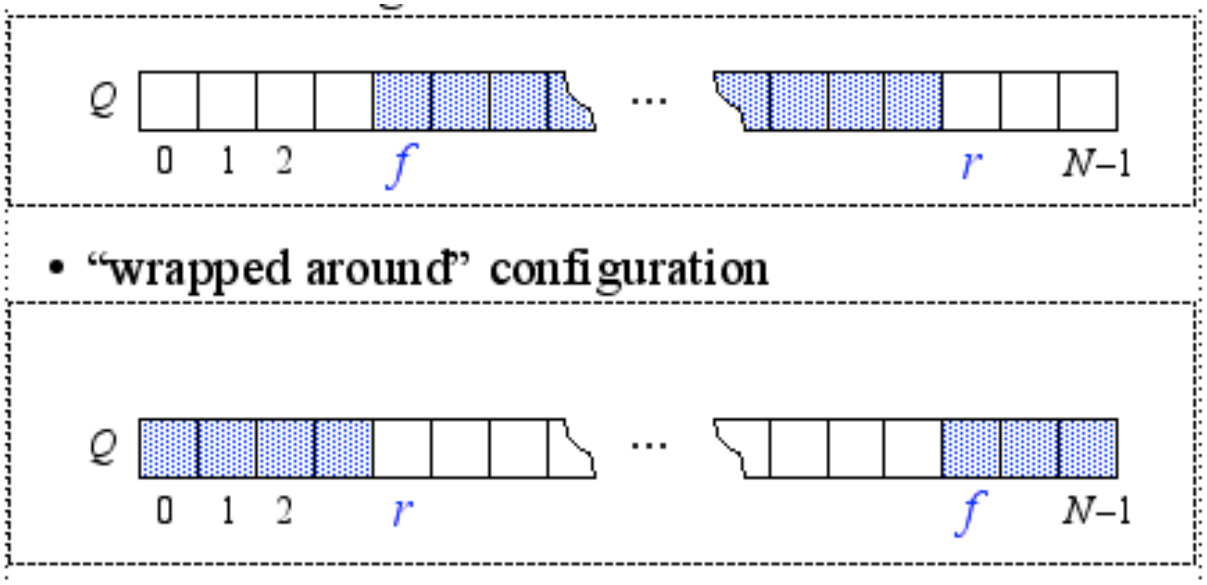
$F \leftarrow (F+1) \bmod N$; $SZ \leftarrow SZ-1$

Insert: at $R = (F+SZ) \bmod N$

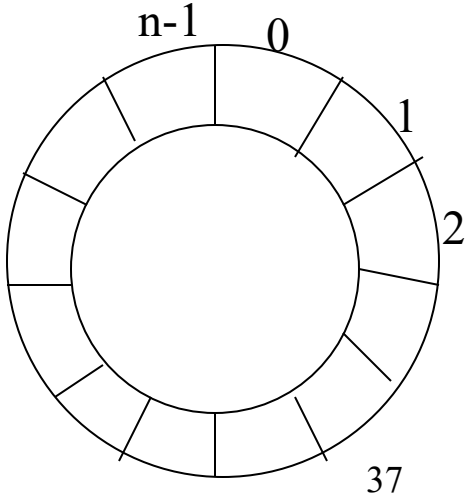
$SZ \leftarrow SZ+1$

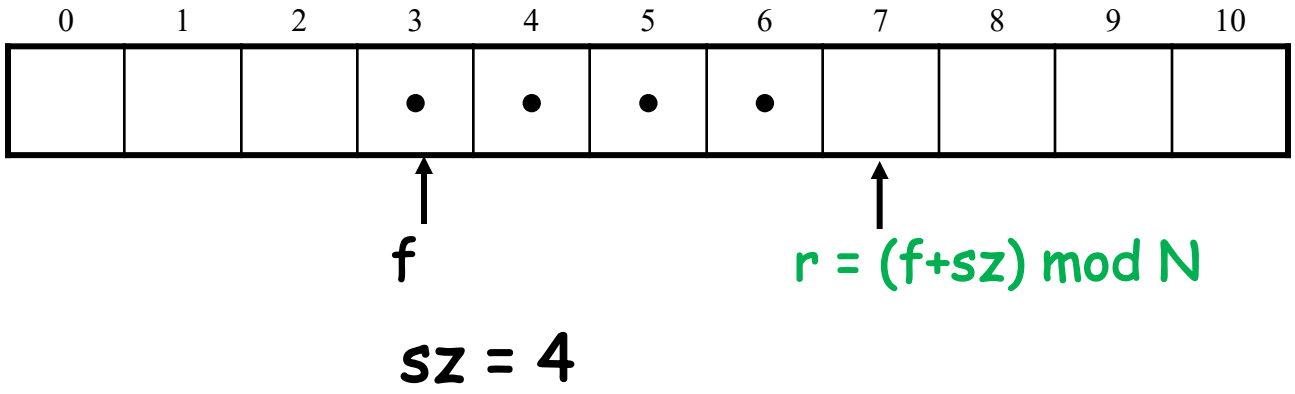
- Array in a circular fashion
- Size N fixed at the beginning

- The queue consists of an N-element array Q and two integer variables:
  - *f*, index of the front element
  - *r*, index of the element after the rear element



We waste one cell and we always leave REAR empty





Questions:

What is the status of an empty queue?

Answer: f=r and sz=0; f may be any index in 0..N-1

Initialize data structure with:
Q ← new array of size N
f ← 0; sz ←0;

```
Algorithm size():
        return sz

Algorithm isEmpty():
        return (sz==0)

Algorithm first():
        if isEmpty() then
            return null
        return Q[f]
```

```
Algorithm dequeue():
        if isEmpty() then
            return null
        temp ← Q[f]
        Q[f] ← null
        f ← (f + 1) mod N
        sz ← sz - 1
        return temp

Algorithm enqueue(o):
        if sz = N - 1 then
            throw IllegalStateException
        else
            r ← (f+sz) mod N
            Q[r] ← o
            sz ← sz + 1
```

Performance

Time:

size()	O(1)
isEmpty()	O(1)
first()	O(1)
enqueue(o)	O(1)
dequeue()	O(1)

Space: O(sz)

Our Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Assumes that first() and dequeue() return null if queue is empty

```
public interface Queue<E> {
  int size();
  boolean isEmpty();
  E first();
  void enqueue(E e);
  E dequeue();
}
```

Array-based Implementation

concrete data structure for Queue based on arrays

Queue ADT

```
1  /** Implementation of the queue ADT using a fixed-length array. */
2  public class ArrayQueue<E> implements Queue<E> {
3    // instance variables
4    private E[ ] data;                    // generic array used for storage
5    private int f = 0;                    // index of the front element
6    private int sz = 0;                   // current number of elements
7
8    // constructors
9    public ArrayQueue( ) {this(CAPACITY);} // constructs queue with default capacity
10   public ArrayQueue(int capacity) {     // constructs queue with given capacity
11     data = (E[ ]) new Object[capacity]; // safe cast; compiler may give warning
12   }
13
14   // methods
15   /** Returns the number of elements in the queue. */
16   public int size( ) { return sz; }
17
18   /** Tests whether the queue is empty. */
19   public boolean isEmpty( ) { return (sz == 0); }
20
```

Array-based Implementation (2)

```
21    /** Inserts an element at the rear of the queue. */
22    public void enqueue(E e) throws IllegalStateException {
23      if (sz == data.length) throw new IllegalStateException("Queue is full");
24      int avail = (f + sz) % data.length;       // use modular arithmetic
25      data[avail] = e;
26      sz++;
27    }
28
29    /** Returns, but does not remove, the first element of the queue (null if empty). */
30    public E first( ) {
31      if (isEmpty( )) return null;
32      return data[f];
33    }
34
35    /** Removes and returns the first element of the queue (null if empty). */
36    public E dequeue( ) {
37      if (isEmpty( )) return null;
38      E answer = data[f];
39      data[f] = null;                           // dereference to help garbage collection
40      f = (f + 1) % data.length;
41      sz--;
42      return answer;
43    }
```

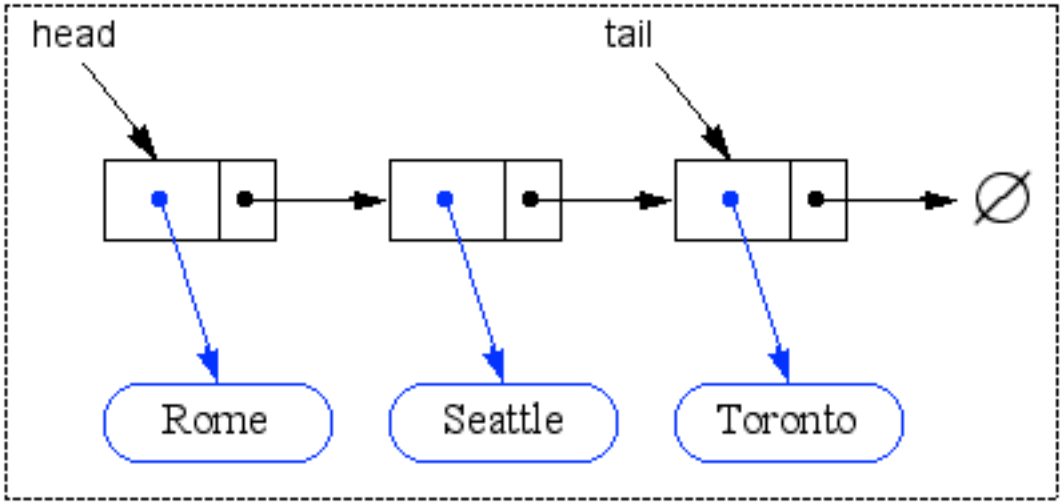
Comparison to java.util.Queue

- Our Queue methods and corresponding methods of java.util.Queue:

Our Queue ADT	**Interface java.util.Queue**	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue( )	remove( )	poll( )
first( )	element( )	peek( )
size( )	size( )	
isEmpty( )	isEmpty( )	

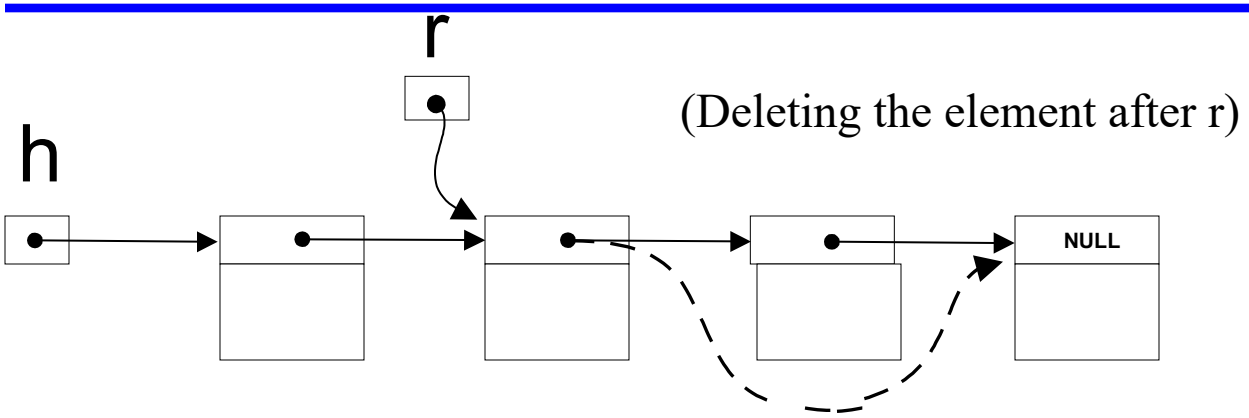
Implementing a Queue with a Singly Linked List

Nodes connected in singly linked list
We keep a pointer to the head and one to the tail



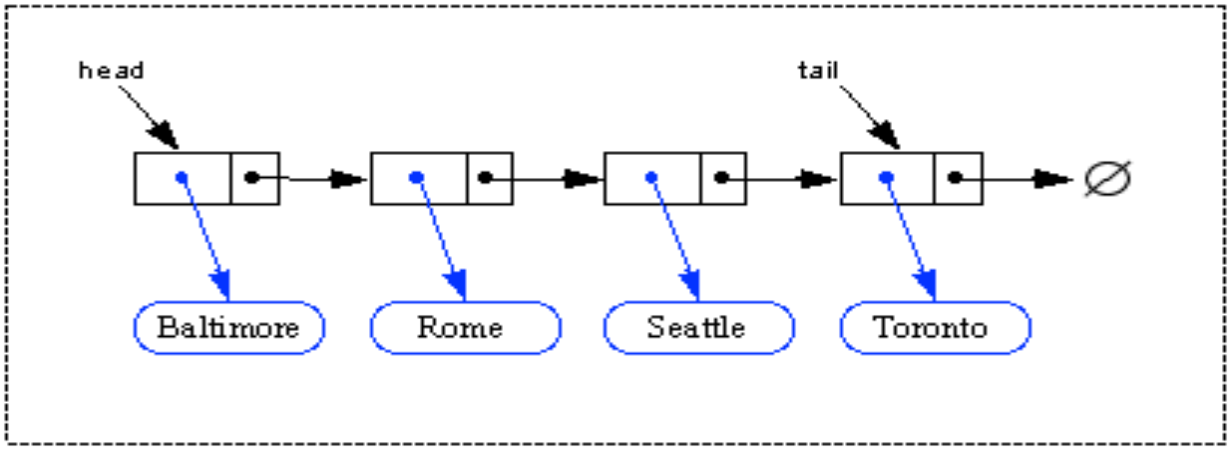
The head of the list is the front of the queue, the tail of the list is the rear of the queue. *Why not the opposite?*

Remember ....

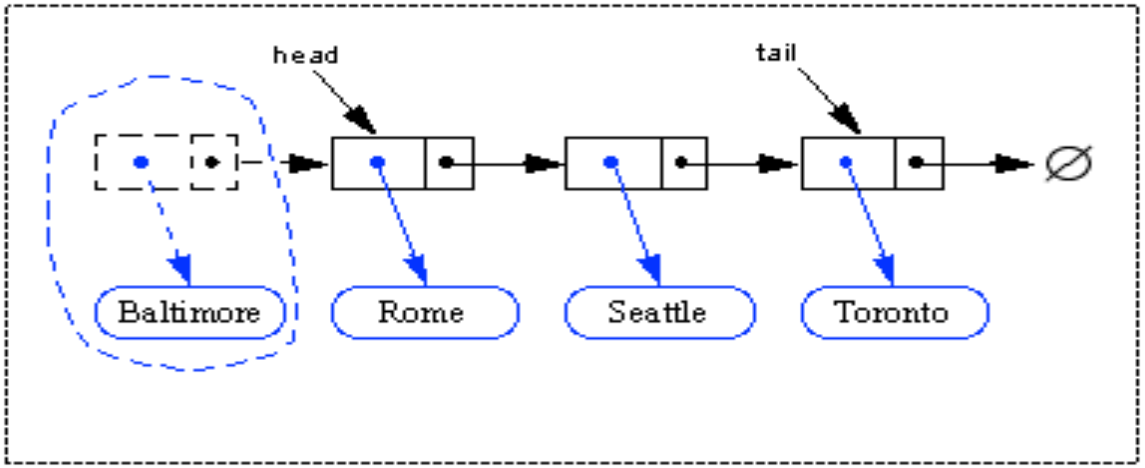


First element (easy)	h ← h.getNext()
Element after r (easy)	w ← r.getNext() r.setNext(w.getNext())
Element at r (difficult)	• Use a pointer to the preceding element, or • Exchange the content of the element at r with the contents of the element following r, and delete the element after r. (this is impossible if r points to the last element)

Removing at the Head



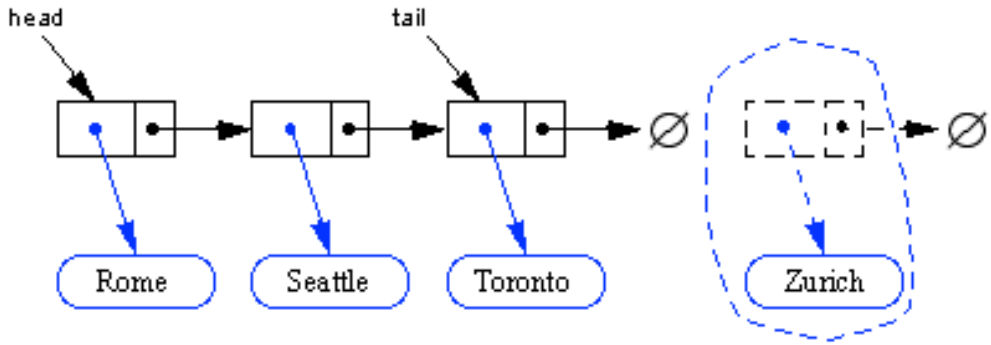
- advance head reference



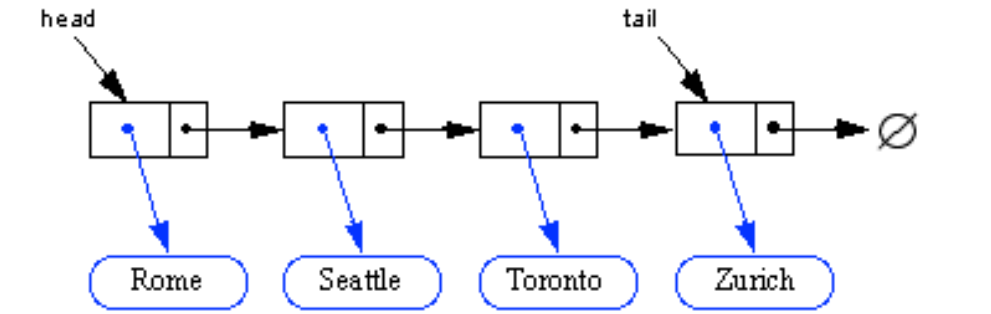
- inserting at the head is just as easy (but we won't do it !)

Inserting at the Tail

- create a new node



- chain it and move the tail reference



- how about removing at the tail?

This is very difficult; we won't do it as it would take O(n).

Queue implementation using Singly Linked Lists

concrete data structure for Queue based on linked list

Queue ADT

```
public class LinkedQueue<E> implements Queue<E> {

    private SinglyLinkedList<E> list = new SinglyLinkedList<>();  // an empty  list

    public LinkedQueue() { }     // new queue relies on the initially empty list

    public void enqueue(E element) { list.addLast(element); }

    public E dequeue() { return list.removeFirst(); }

    public int size() { return list.size(); }

    public E first() { return list.first(); }

    public boolean isEmpty() { return list.isEmpty(); }
```

ADAPTING an existing class SinglyLiskedList<E>

(See lecture on singly linked lists where this class is defined)

Performance

Time:

size()	O(1)
isEmpty()	O(1)
front()	O(1)
enqueue(o)	O(1)
dequeue()	O(1)

Space: O(n) when storing n elements

To calculate, need to verify the big-Oh of each method you call. Looking at previous page, the corresponding operations used for singly linked list are all O(1).
(Review lecture about singly linked lists)

Choosing Queue implementation

If we know in advance a reasonable upper bound for the number of elements in the queue, then use

⇨ ARRAYS (fixed size N, as seen here)

Otherwise

⇨ LISTS

⇨ Extendible ARRAYS (when full increases the size; as will see later)

Example: Palindromes

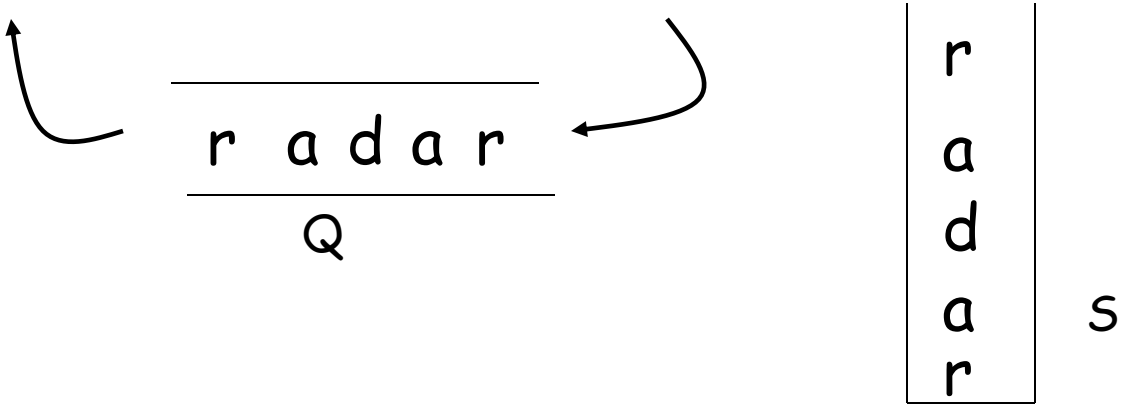
Problem: Decide if a string is a palindrome using one queue and one stack

"RADAR"

"WAS IT A CAR OR A CAT I SAW"

Read the line into a stack and into a queue

Compare the outputs of the queue and the stack



MADAM, I' M ADAM

LONELY TYLENOL

NEVER ODD OR EVEN

NO LEMON NO MELON

TOO BAD I HID A BOOT

O STONE BE NOT SO

RACE FAST SAFE CAR

DO GEESE SEE GOD

NO DEVIL LIVED ON

A TOYOTA' S A TOYOTA

A more general ADT: Double-Ended Queues (Deque)

A double-ended queue, or deque, supports insertion and deletion from the front and back.

Main methods:

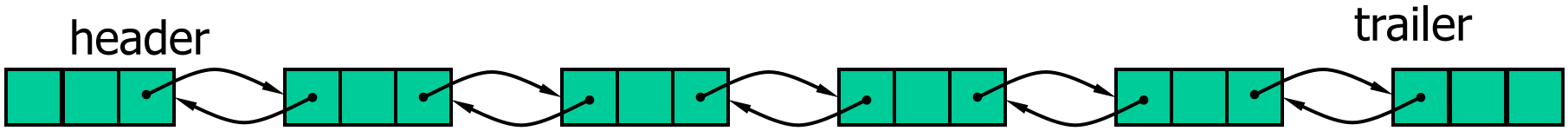
addFirst(e):	Insert e at the beginning of deque.
addLast(e):	Insert e at end of deque
removeFirst():	Removes and returns first element
removeLast():	Removes and returns last element

Support methods:

- first()
- last()
- size()
- isEmpty()

Implementing Deques with Doubly Linked Lists

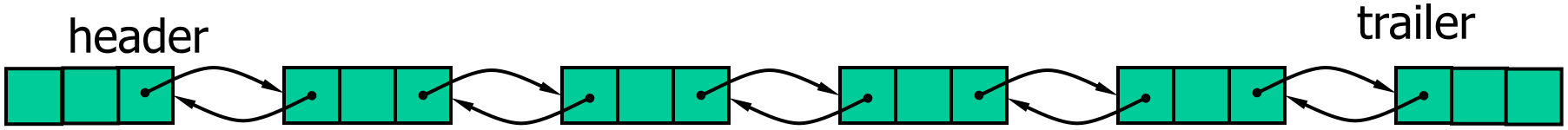
Deletions at the tail of a singly linked list cannot be done efficiently



To implement a deque, we use a doubly linked list with special header and trailer nodes

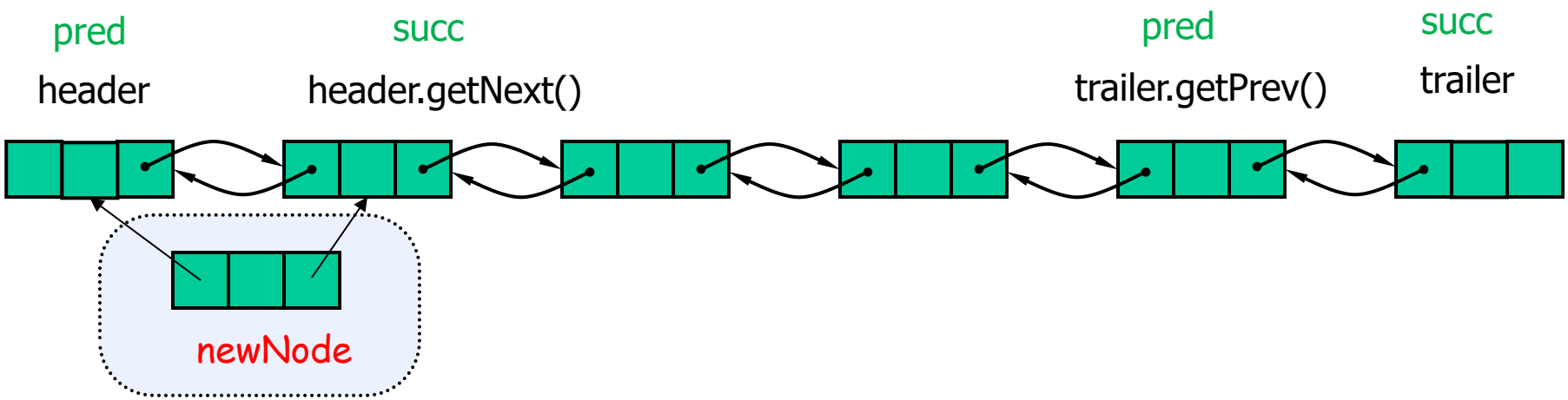
- The *header* node goes before the first list element. It has a valid next link but a null prev link.

- The *trailer* node goes after the last element. It has a valid prev reference but a null next reference.



NOTE: the header and trailer nodes are sentinel or "dummy" nodes because they do not store elements.

Insertion : addFirst(o) and addLast(o)

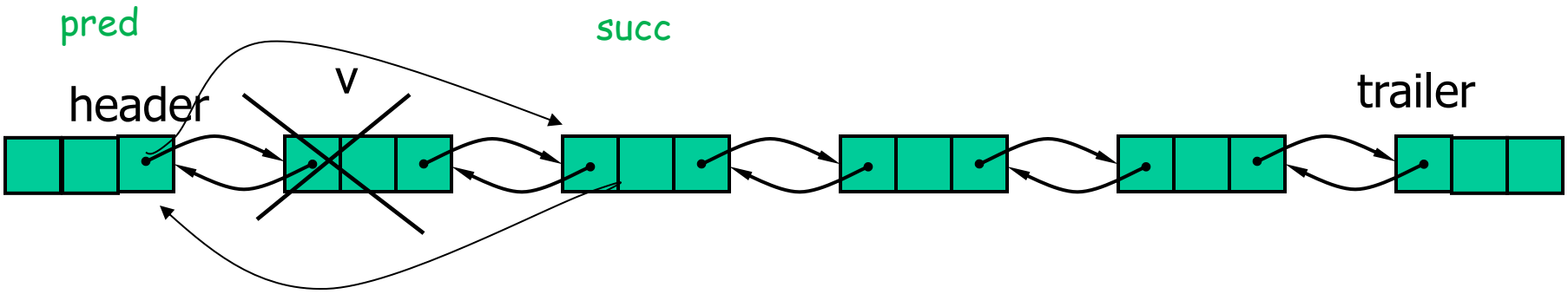


Algorithm addBetween(e,pred,succ):
newNode ← new Node;
newNode.setElement(e);
newNode.setPrev(pred);
newNode.setNext(succ);
pred.setNext(newNode);
succ.setPrev(newNode);
size ← size+1

Algorithm addFirst(e):
addBetween(e, header,header.getNext())

Algorithm addLast(e):
addBetween(e, trailer.getPrev(),trailer);

Deletion : removeFirst() and removeLast()

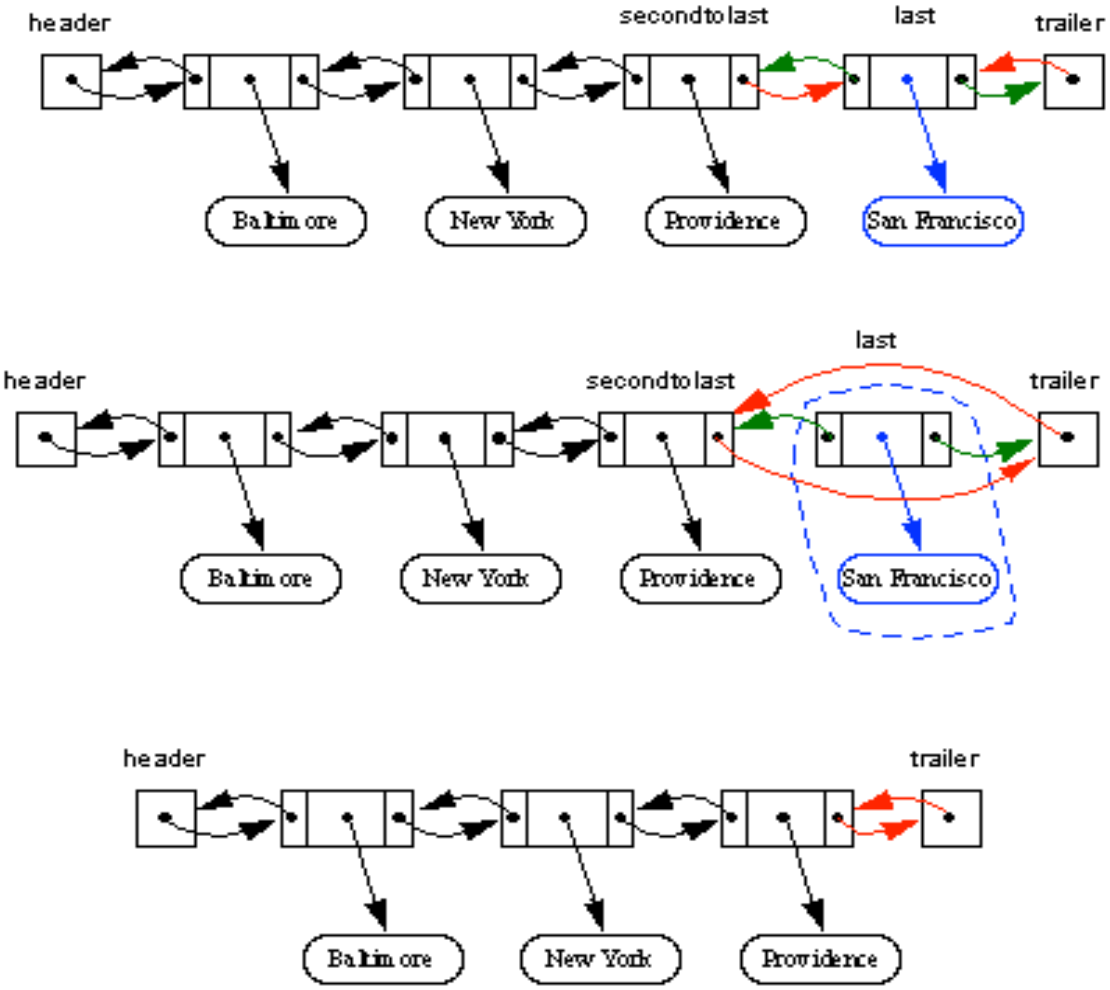



```
Algorithm remove(v):
        pred ← v.getPrev();
        succ ← v.getNext();
        newNode ← new Node;
        pred.setNext(succ);
        succ.setPrev(pred);
        size ← size-1
        return v.getElement();
```

```
Algorithm removeFirst():
    if (isEmpty()) return null;
    return remove(header.getNext());
```

```
Algorithm removeLast():
    if (isEmpty()) return null;
    return remove(trailer.getPrev());
```

Here's a visualization of the code for removeLast():



With this implementation, all methods have complexity O(1)

Performance: Deques using Doubly Linked Lists

Operation	Time
addFirst(e):	O(1)
addLast(e):	O(1)
removeFirst():	O(1)
removeLast():	O(1)
first()	O(1)
last()	O(1)
size()	O(1)
isEmpty()	O(1)

Space: O(n)
for a deque with n items

Implementing Stacks and Queues with Deques

Stacks with Deques:

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(o)	addLast(o)
pop()	removeLast()

Queues with Deques:

Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue(o)	addLast(o)
dequeue()	removeFirst()