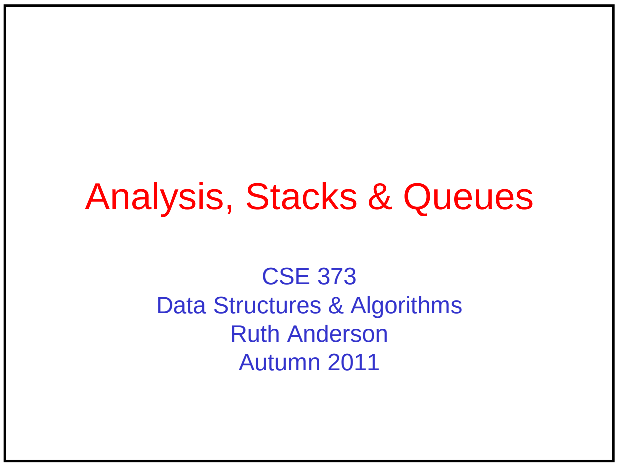
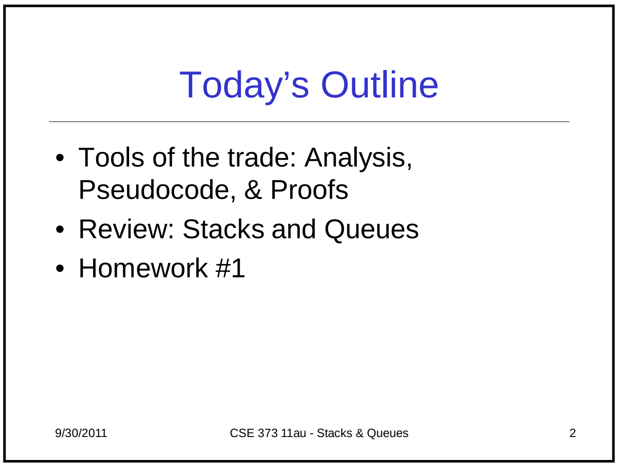


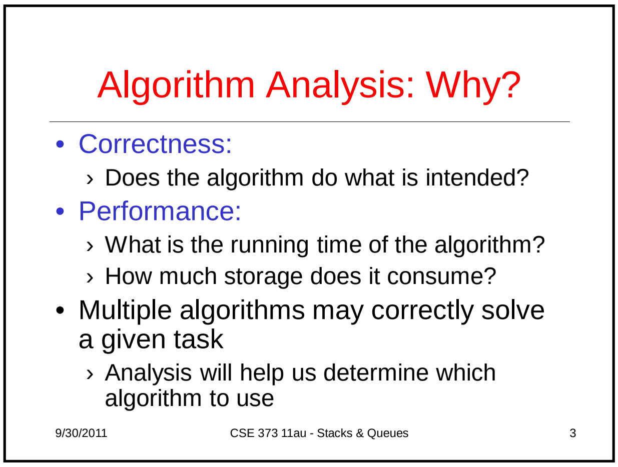
Analysis, Stacks & Queues

CSE 373
Data Structures & Algorithms
Ruth Anderson
Autumn 2011



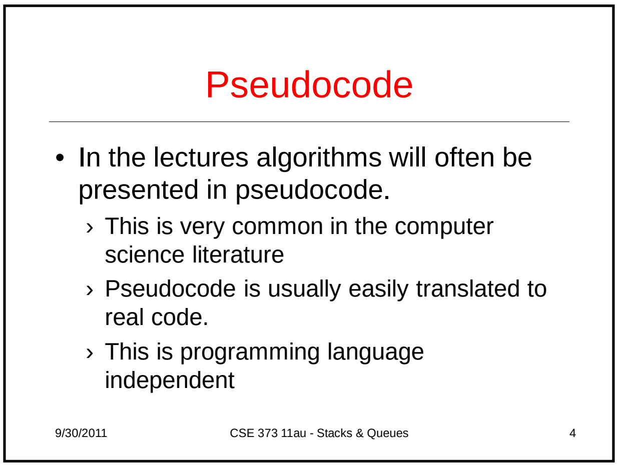
Today's Outline

- Tools of the trade: Analysis, Pseudocode, & Proofs
- Review: Stacks and Queues
- Homework #1



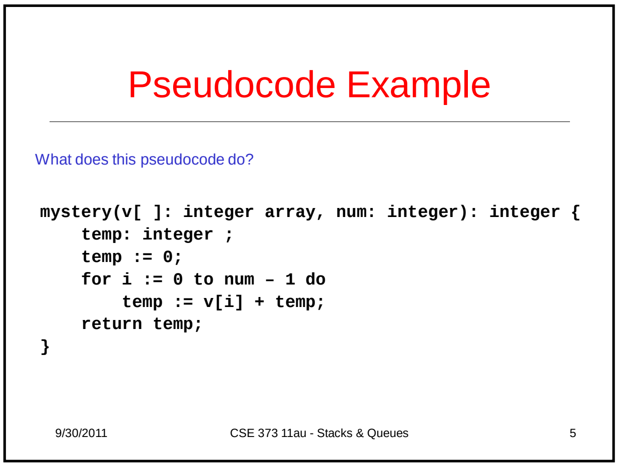
Algorithm Analysis: Why?

- Correctness:
  - › Does the algorithm do what is intended?
- Performance:
  - › What is the running time of the algorithm?
  - › How much storage does it consume?
- Multiple algorithms may correctly solve a given task
  - › Analysis will help us determine which algorithm to use



Pseudocode

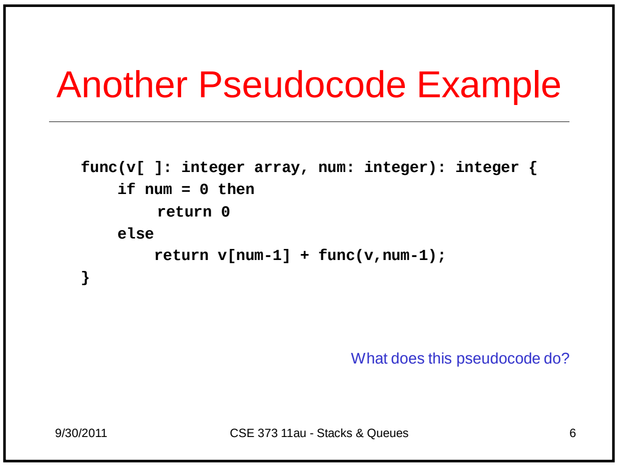
- In the lectures algorithms will often be presented in pseudocode.
  - › This is very common in the computer science literature
  - › Pseudocode is usually easily translated to real code.
  - › This is programming language independent



Pseudocode Example

What does this pseudocode do?

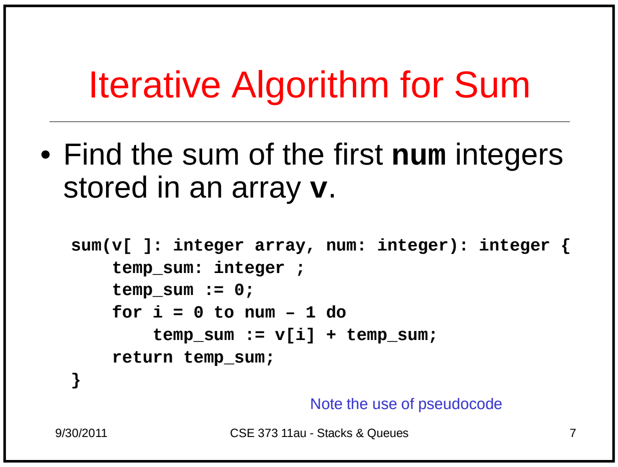
```
mystery(v[ ]: integer array, num: integer): integer {
    temp: integer ;
    temp := 0;
    for i := 0 to num – 1 do
        temp := v[i] + temp;
    return temp;
}
```



Another Pseudocode Example

```
func(v[ ]: integer array, num: integer): integer {
    if num = 0 then
        return 0
    else
        return v[num-1] + func(v,num-1);
}
```

What does this pseudocode do?

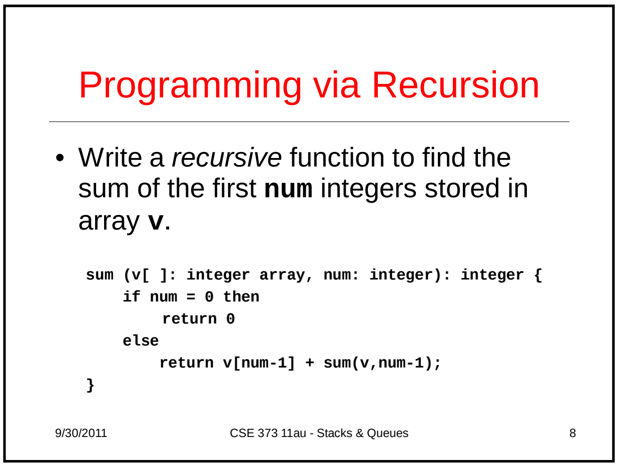


Iterative Algorithm for Sum

- Find the sum of the first **num** integers stored in an array **v**.

```
sum(v[ ]: integer array, num: integer): integer {
    temp_sum: integer ;
    temp_sum := 0;
    for i = 0 to num – 1 do
        temp_sum := v[i] + temp_sum;
    return temp_sum;
}
```

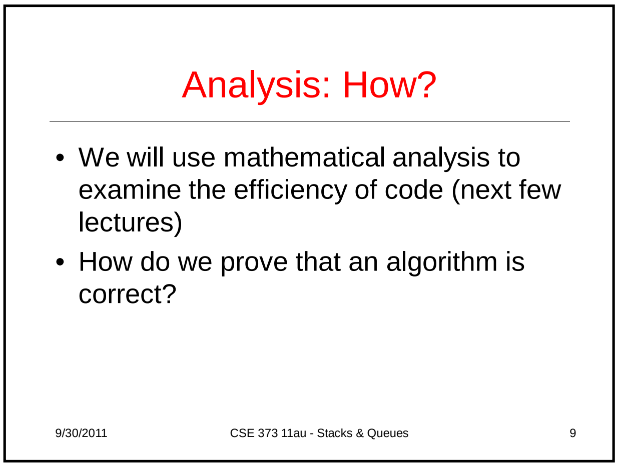
Note the use of pseudocode



Programming via Recursion

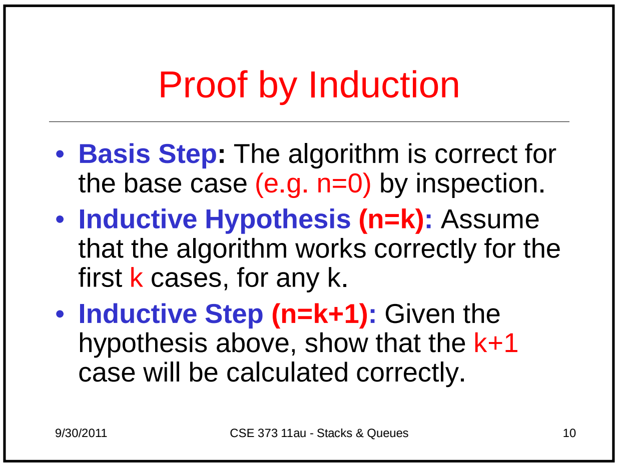
- Write a *recursive* function to find the sum of the first **num** integers stored in array **v**.

```
sum (v[ ]: integer array, num: integer): integer {
    if num = 0 then
        return 0
    else
        return v[num-1] + sum(v,num-1);
}
```



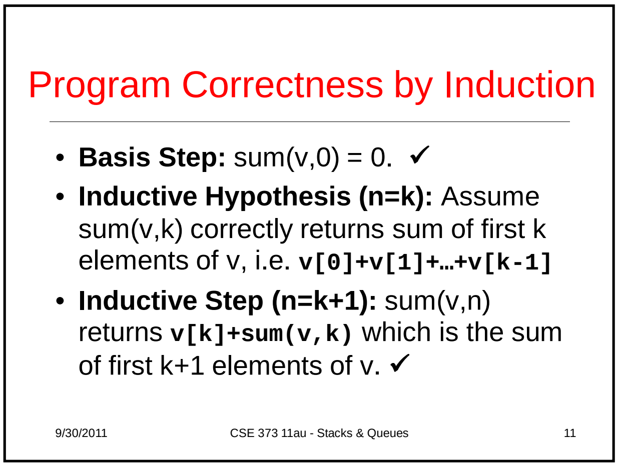
Analysis: How?

- We will use mathematical analysis to examine the efficiency of code (next few lectures)
- How do we prove that an algorithm is correct?



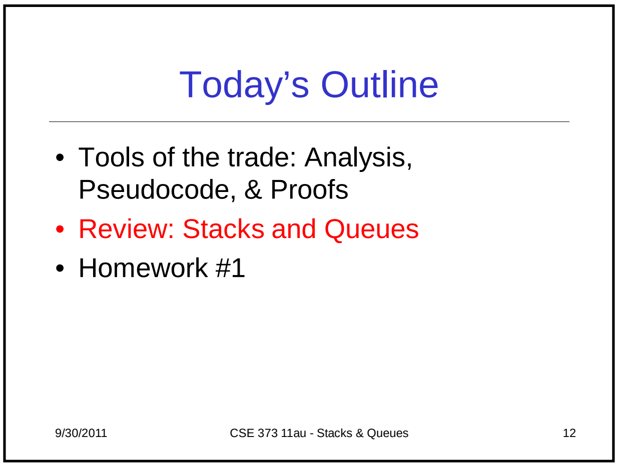
Proof by Induction

- **Basis Step:** The algorithm is correct for the base case (e.g. n=0) by inspection.
- **Inductive Hypothesis (n=k):** Assume that the algorithm works correctly for the first k cases, for any k.
- **Inductive Step (n=k+1):** Given the hypothesis above, show that the k+1 case will be calculated correctly.



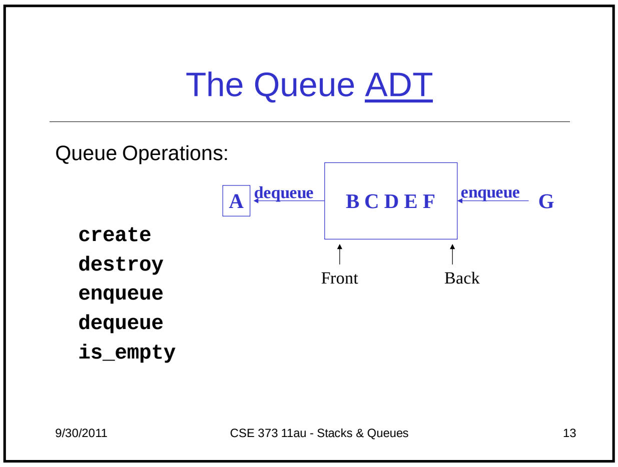
Program Correctness by Induction

- **Basis Step:** sum(v,0) = 0. ✓
- **Inductive Hypothesis (n=k):** Assume sum(v,k) correctly returns sum of first k elements of v, i.e. **`v[0]+v[1]+…+v[k-1]`**
- **Inductive Step (n=k+1):** sum(v,n) returns **`v[k]+sum(v,k)`** which is the sum of first k+1 elements of v. ✓



Today's Outline

- Tools of the trade: Analysis, Pseudocode, & Proofs
- Review: Stacks and Queues
- Homework #1

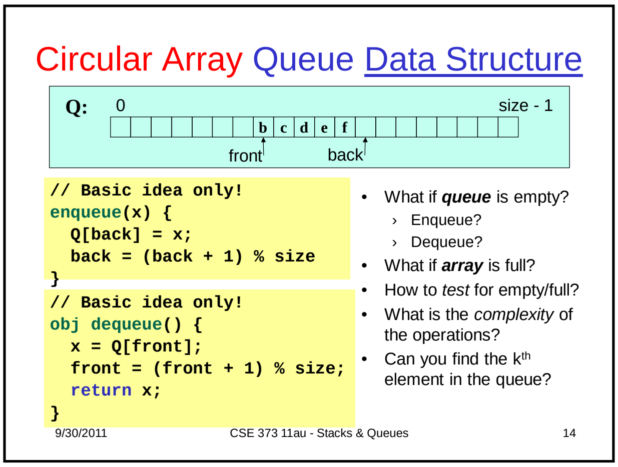


The Queue ADT

Queue Operations:

create
destroy
enqueue
dequeue
is_empty

A
dequeue
B C D E F
enqueue
G
Front
Back



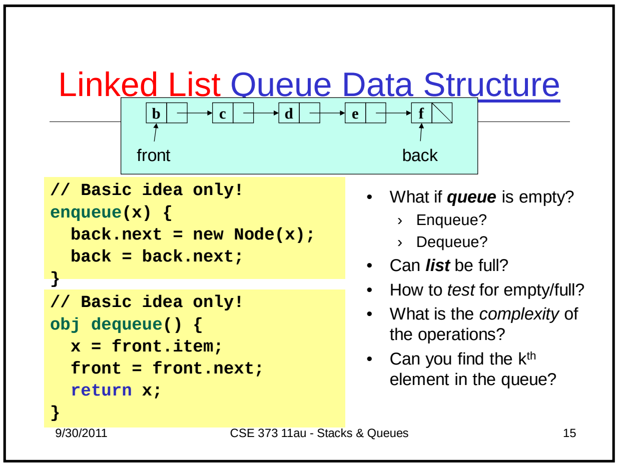
Circular Array Queue Data Structure

Q:
0
size - 1
b
c
d
e
f
front
back

```
// Basic idea only!
enqueue(x) {
  Q[back] = x;
  back = (back + 1) % size
}
```

```
// Basic idea only!
obj dequeue() {
  x = Q[front];
  front = (front + 1) % size;
  return x;
}
```

- What if ***queue*** is empty?
  - › Enqueue?
  - › Dequeue?
- What if ***array*** is full?
- How to *test* for empty/full?
- What is the *complexity* of the operations?
- Can you find the k^{th} element in the queue?



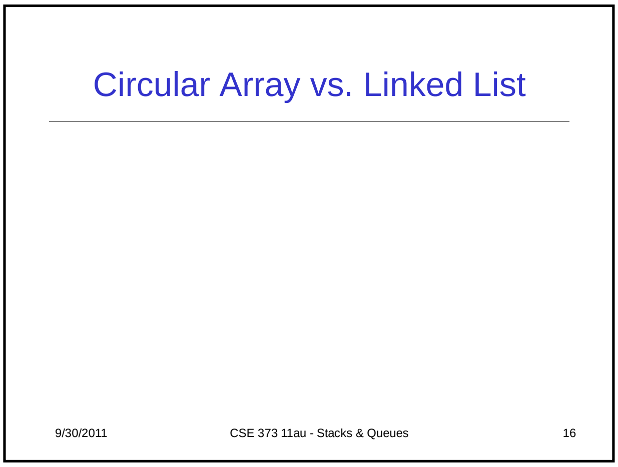
Linked List Queue Data Structure

b
c
d
e
f
front
back

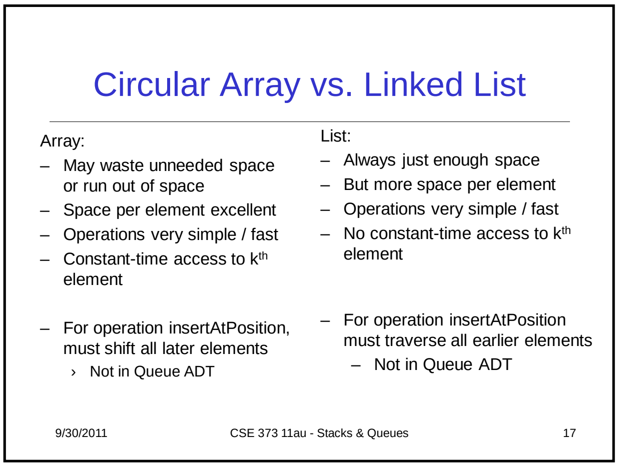
```
// Basic idea only!
enqueue(x) {
  back.next = new Node(x);
  back = back.next;
}
```

```
// Basic idea only!
obj dequeue() {
  x = front.item;
  front = front.next;
  return x;
}
```

- What if ***queue*** is empty?
  - › Enqueue?
  - › Dequeue?
- Can ***list*** be full?
- How to *test* for empty/full?
- What is the *complexity* of the operations?
- Can you find the k^{th} element in the queue?



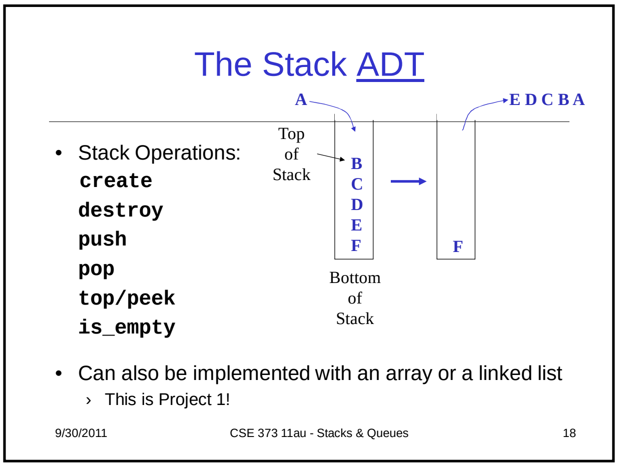
Circular Array vs. Linked List



Circular Array vs. Linked List

Array:
- – May waste unneeded space or run out of space
- – Space per element excellent
- – Operations very simple / fast
- – Constant-time access to k^{th} element
- – For operation insertAtPosition, must shift all later elements
  - › Not in Queue ADT

List:
- – Always just enough space
- – But more space per element
- – Operations very simple / fast
- – No constant-time access to k^{th} element
- – For operation insertAtPosition must traverse all earlier elements
  - – Not in Queue ADT



The Stack ADT

- Stack Operations:
  create
  destroy
  push
  pop
  top/peek
  is_empty

A
E D C B A
Top
of
Stack
B
C
D
E
F
F
Bottom
of
Stack

- Can also be implemented with an array or a linked list
  - › This is Project 1!

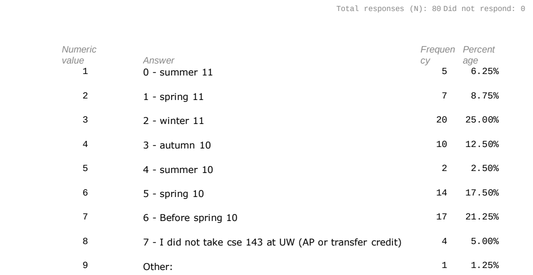
Stacks in Practice

- Function call stack
- Removing recursion
- Checking if symbols (parentheses) are balanced
- Evaluating Postfix Notation

Homework #1 – Sound Blaster!

- Reverse sound clips using a stack!
- Implement a stack interface two ways:
  - › With an array
  - › With linked list nodes (make your own nodes)
- Do NOT use LinkedList or other things from Java Collections

When did you take cse 143 (what quarter)?



Total responses (N): 80 Did not respond: 0

Numeric value	Answer	Frequency	Percentage
1	0 - summer 11	5	6.25%
2	1 - spring 11	7	8.75%
3	2 - winter 11	20	25.00%
4	3 - autumn 10	10	12.50%
5	4 - summer 10	2	2.50%
6	5 - spring 10	14	17.50%
7	6 - Before spring 10	17	21.25%
8	7 - I did not take cse 143 at UW (AP or transfer credit)	4	5.00%
9	Other:	1	1.25%