COMP20003 Workshop Week 4

It's about Complexity and … A1

1. Complexity Analysis
2. Another ADT: Stack
3. Yet Another ADT: Queue

LAB:

- Finish Assignment 1 OR do Week 4 Extras

Concepts: Big-O: asymptotic evaluation of running time (or space)

$f(n) = O(\ g(n)\ )$:

- **Def:** There are constants $c>0$ and $n_0 \geq 0$ so that $f(n) \leq c.g(n)$ for all $n > n_0$
- *Underlying meaning*: f(n) grows slower than, or as the same rate as, g(n)

$f(n) = O(\ g(n)\ )$ is *equivalent* to:

- ✓ $f(n) \leq c.g(n)$ for some constant c and all large n
- ✓ $c.g(n)$ is *one of* the upper bounds of f(n) for all large n
- ✓ $f(n)= O(\ f(n)\ )$
- ✓ $f(n)= O(h(n))$ for any function h(n) that grows faster than f(n) or g(n)

BE CAREFUL! Any of:

- $f(n) \leq g(n)$
- $f(n) \leq c.g(n)$
- $f(n) \ll g(n)$ (sign << denotes: grows slower than)

implies $f(n)= O(\ g(n)\ )$, but the *reverse is incorrect*. For example, $f(n)= O(\ g(n)\ )$ does NOT imply that $f(n) \leq g(n)$.

Example: find g(n) so that $f(n)= O(\ g(n)\ )$ if

- $f(n)= 7n$
- $f(n)= 2n^2 + 5n + 1$

Concepts: from Big-O to Big-Ω and Big-θ

$f(n) = \Omega(g(n)) \Leftrightarrow g(n) = O(f(n))$

- **Def:** There are constants $c>0$ and $n_0 \geq 0$ so that $f(n) \geq cf(n)$ for all $n > n_0$

→ $f(n) \geq c.\ g(n)$ (for some constant c and all large n)

$f(n) = \theta(g(n)) \Leftrightarrow f(n) = O(g(n))$ AND $f(n) = \Omega(g(n))$

→ f(n) and g(n) *grow at the same rate*

→ f(n) is sandwiched between $c_1.g(n)$ and $c_2.g(n)$: $c_1.g(n) \leq f(n) \leq c_2.g(n)$

(for some constants c_1, c_2 and all large n)

Example: find some Ω and θ for:

- $f(n)= 7n$
- $f(n)= 2n^2 + 5n + 1$

Do W4.4 Now

Application 1: Finding Complexity of C codes (and algorithms in general)

Rules from lectures/Skiena:

- Each simple operation takes exactly one time step.
- Each memory access takes exactly one time step.

So practically:

A finite number of assignments, arithmetic and logical expressions is just $\theta(1)$.
“finite” means “not depending on input size”

Examples: are they $\theta(1)$?

```
a=  (b+c)*d  -  x;
if  (  a+b  >  c<<10  )
    a=  x  +  a*b;
```

```
for  (i=0;  i<1000;  i++)  {
    a=  a*(b+c)*d  -  x;
}
```

Application 1: Find complexity of algorithms, including C codes

Rules from lectures/Skiena:

- Loops and subroutines/functions are not considered simple operations. Instead, they are the composition of many single-step operations.

```
1   for (i=0; i<n; i++)
2       sum = sum + i;
```

Method 1

- Loop body is θ(1), and *always* repeats n times
- So, it's θ(n x 1) = θ(n)

Method 2

- Loop body is constant time, equivalent to 1
- Line 1 has i running from 0 to n-1
- So, total time is

$$\sum_{i=0}^{n-1} 1 = n$$

hence it's θ(n)

Application 1: Find complexity of algorithms, including C codes

```
for (i=0; i<n; i++) {
    if (x==A[i]) return i;
}
return -1;
```

Method 1:

Method 2:

Application 1: Find complexity of algorithms, including C codes

```
1   for (i=1; i<n; i++) {
2       if (x==A[i]) return i;
3   }
4   return -1;
```

Method 1:

- Loop body is θ(1) (or O(1))
- Loop runs at least 1 times, at most n times

→ So O(n), but not θ(n)

Method 2:

- Loop body (line 2) is constant time, equivalent to 1
- *the best case:* the loop runs 1 time an is O(1), so the total running time is θ(1) in the best case
- *the worst case:* The loop runs with i from 0 to n-1, so

  $T(n) = \sum_{i=0}^{n-1} 1$

  = n = θ(n) in the worst case
- *General case:* So T(n) = **Ω**(1), and T(n)= O(n), no θ available

→ This algorithm is $O(n^2)$, its best case is θ(n), worst case is $θ(n^2)$

Do W4.5 and W4.6 Now

Application 2: Comparing complexity functions using *complexity classes*

(One method) to compare f(n) and g(n) in terms of Big-O: first reduce f(n) and g(n) to their simplest form (ie. to their respective complexity classes) using:

Big-O Heuristics

- Ignore coefficient
  (but not when inside a function)
- Drop lower-order terms

Big-O Arithmetic

- $O(f) + O(g) = O(f+g)$
  $= O(\max(f, g))$
- $O(f) * O(g) = O(f*g)$

then compare the simplest forms using:

Growth order from lectures/Skiena's

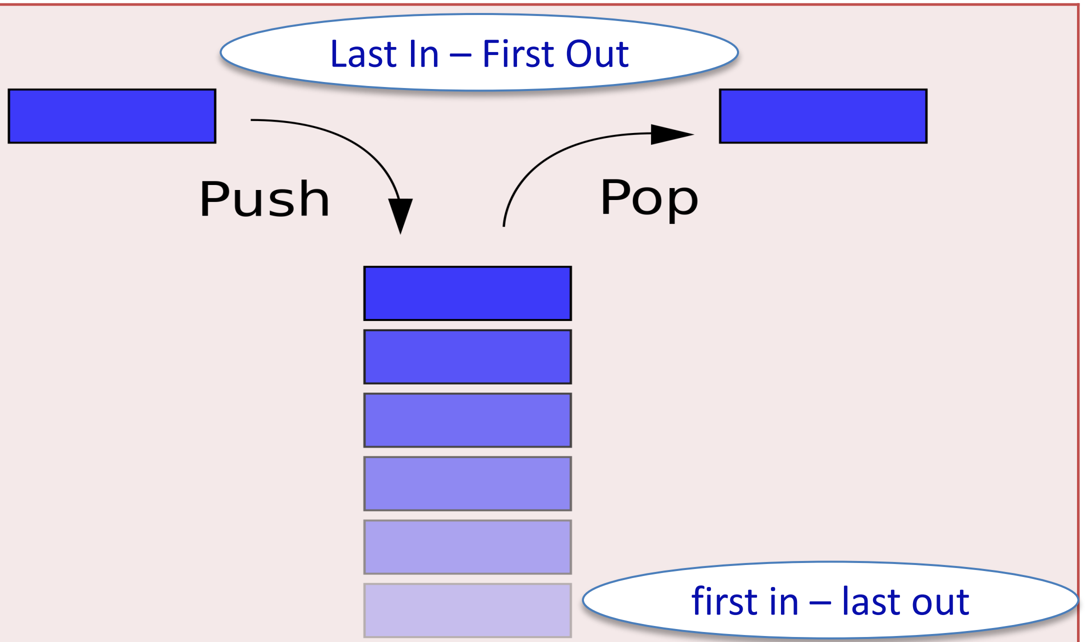
$$n! \gg 2^n \gg n^3 \gg n^2 \gg n\log n \gg n \gg \log n \gg 1$$

Notes:

- $\log_a n = \theta(\log_b n)$ for all a, b > 0
- $a^n \gg b^n$ if $a > b > 1$
- $n^a \gg n^b$ if $a > b > 0$

- Examples?
- Do W4.3 Now

Another ADT: Stack (LIFO) = W4.7

	
http://www.123rf.com/stock-photo/tyre.html	https://simple.wikipedia.org/wiki/Stack_(data_structure)
Stack Operations	**push**: add an element into stack **pop**: remove an element from stack **create**: create a new, empty stack **delete**: delete (free all associated memory) **isEmpty**: check if stack is empty, or **size**: return number of elements in stack

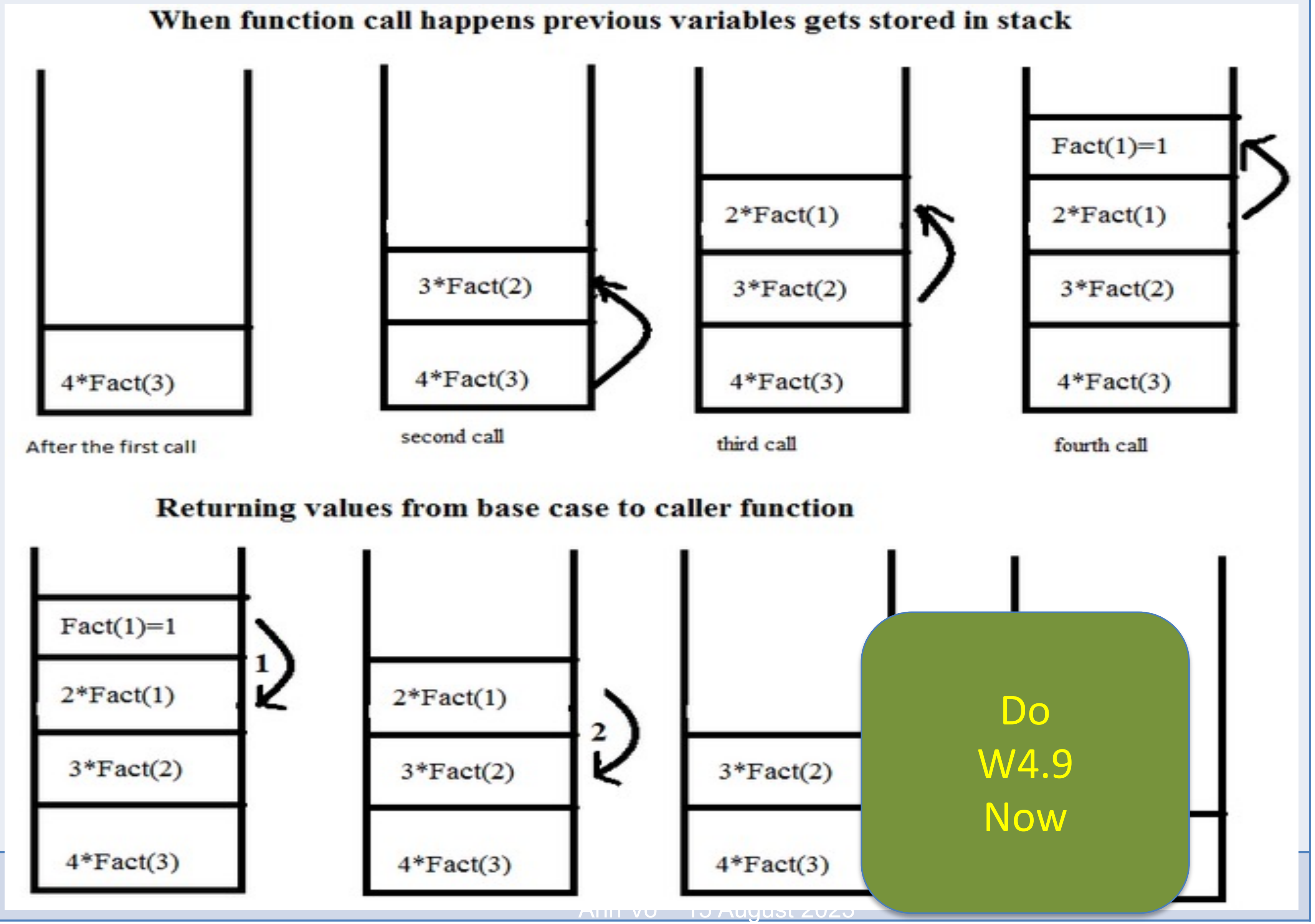
Example: stack in function calls

Stack is widely used in implementation of programming systems. For example, compilers employ stacks for keeping track of function calls and execution.

Stack for :

fact(4)

```
int fact( int n ) {
  if ( n<=1 )
      return 1;
  return n*fact(n-1);
}
```

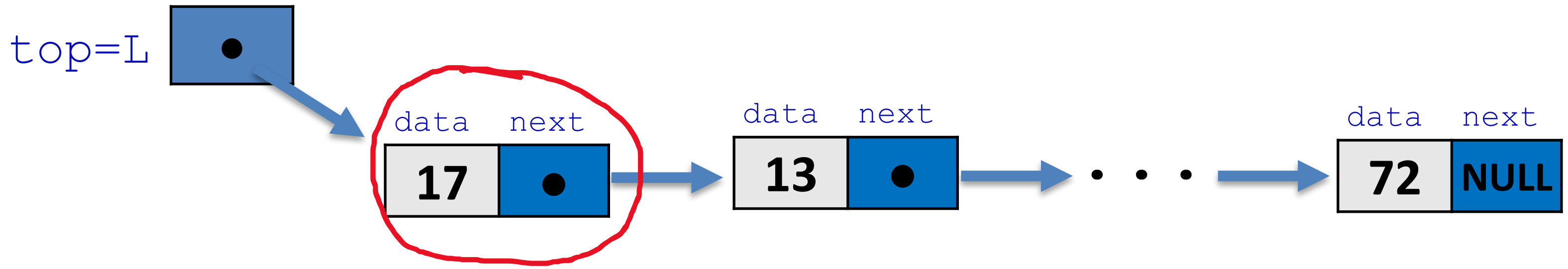


Do
W4.9
Now

Stacks: Implementation using linked lists

push and pop are O(1)

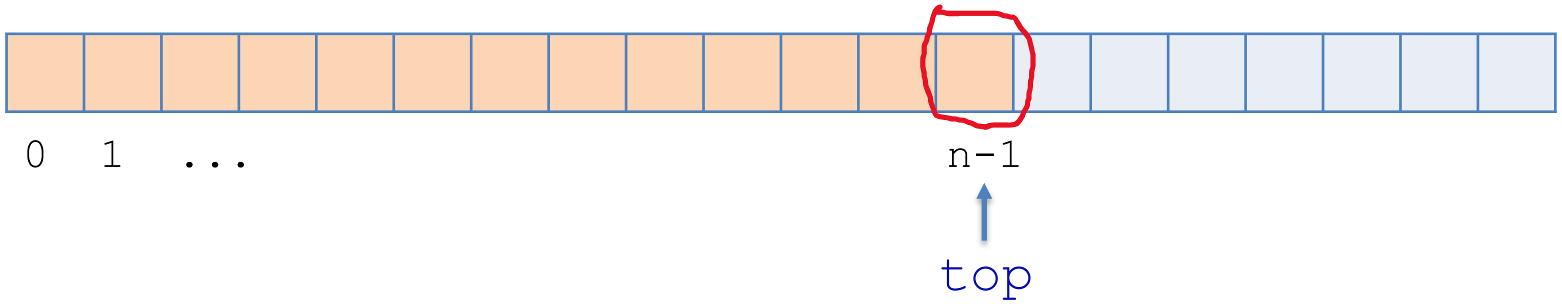
Is simpler than using array: no need to worry about size.



`push(x)=` insert `x` to the start of the list = `listPrepend : O(?)`
`pop =` remove (and return) the head of the list = `listDeleteHead : O(?)`

Stacks: Implementation using arrays

Should be straight-forward.



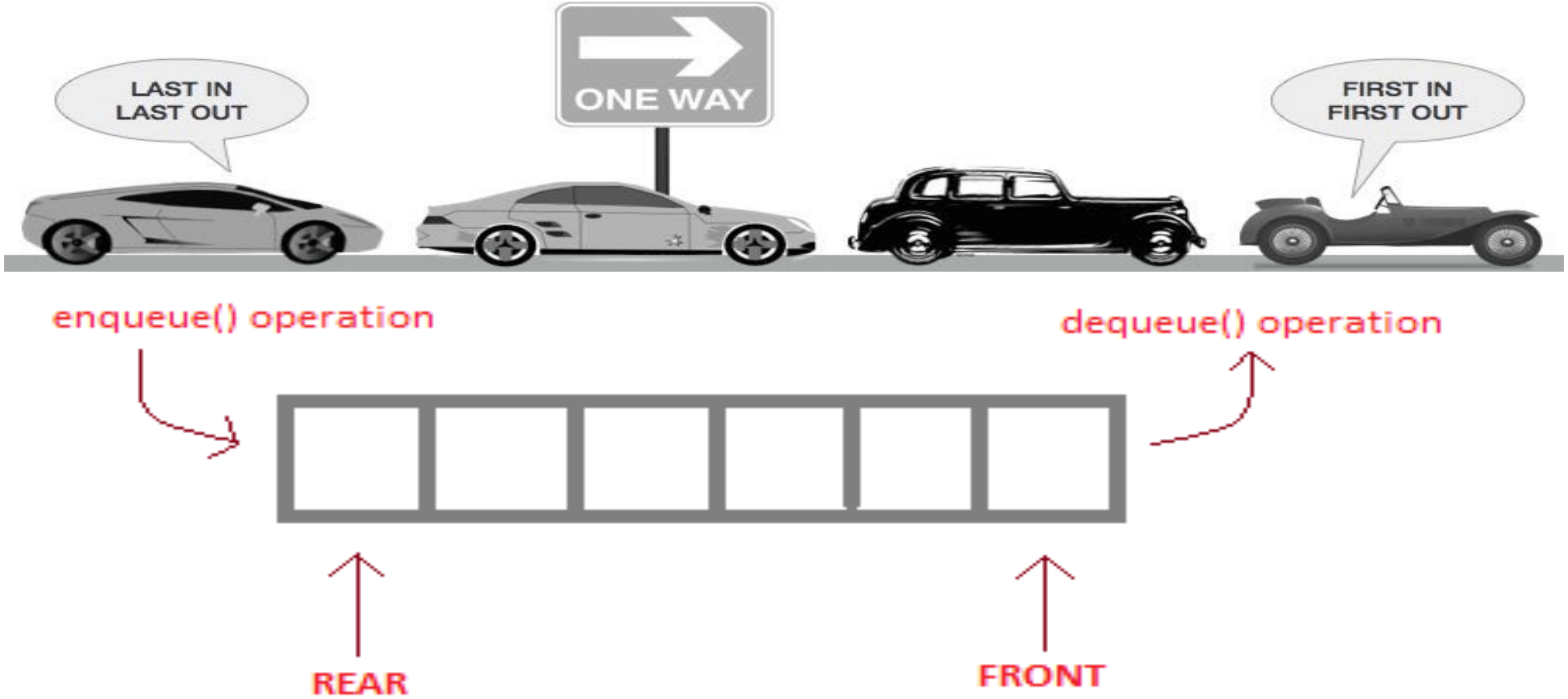
`push(x)` = insert `x` to the end of the array = `arrayAppend: O(?)`
`pop` = remove (and return) the last element of the array: `O(?)`

What is the complexity of `push(x)` in case of:

- using a static array
- using a dynamic array

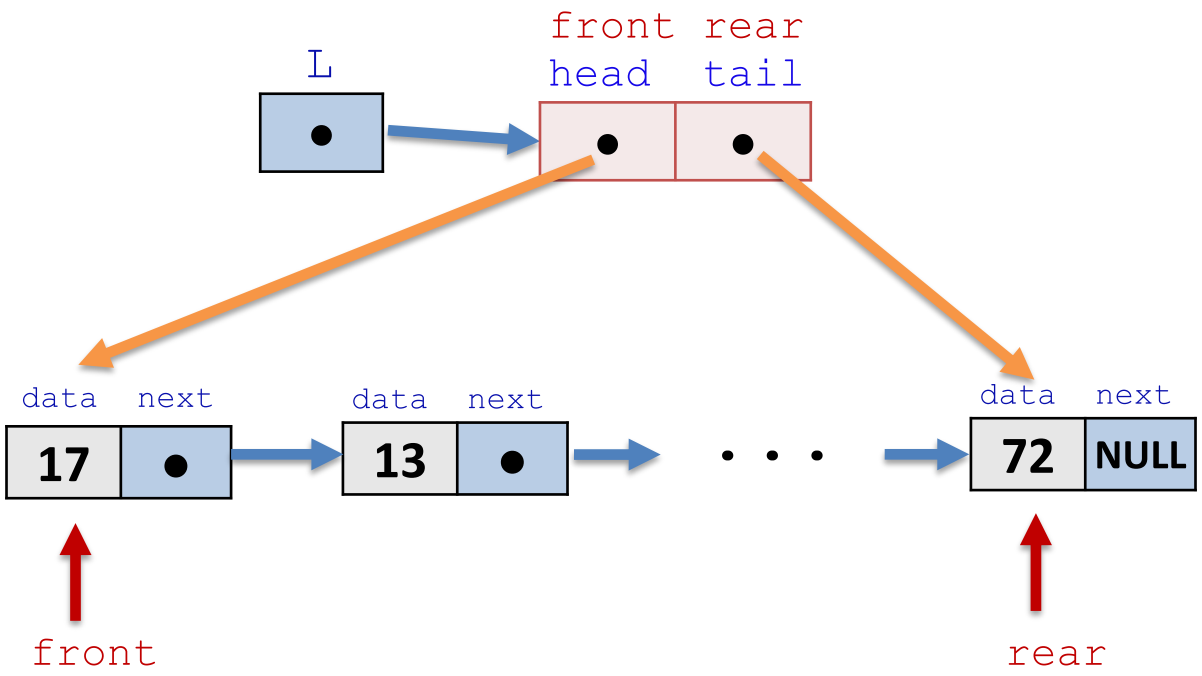
What about `pop`?

Yet another ADT: Queue (FIFO) = W4.8



Queue Operations	**`enqueue(x)`**: add **`x`** to the rear of the queue **`dequeue()`**: remove (and return) the element at front **`create()`**: create a new, empty queue **`delete()`**: delete a queue (free all associated memory) **`isEmpty()`**: check if queue is empty, or **`size()`**: return number of elements in queue

Queue: implementation using linked list



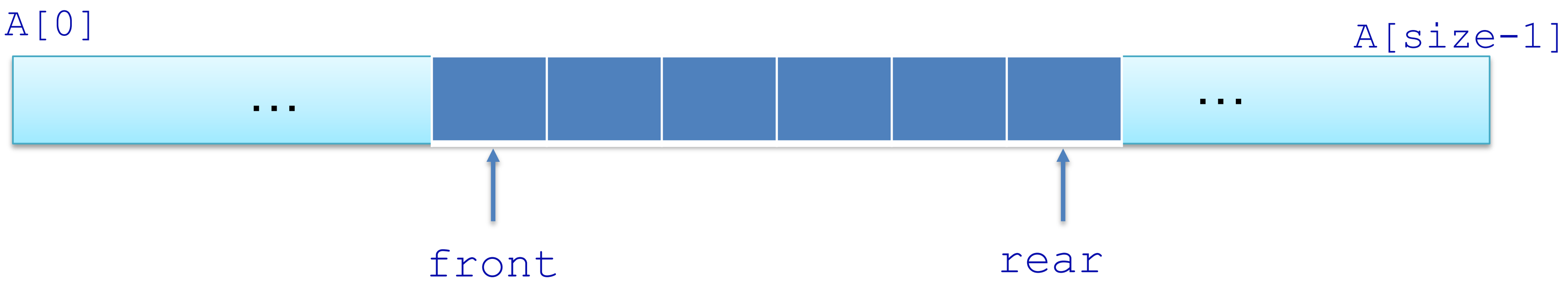
Convenient O(1) for both enqueue and dequeue:

`enqueue` == `listAppend` (can it be `listPrepend`?)

`dequeue` == `listDeleteHead` (can it be `listDeleteTail`?)

Queue: implementation using array

Describe how to implement **`enqueue`** and **`dequeue`** using an unsorted array, ensuring Θ(1) for enqueue & dequeue.

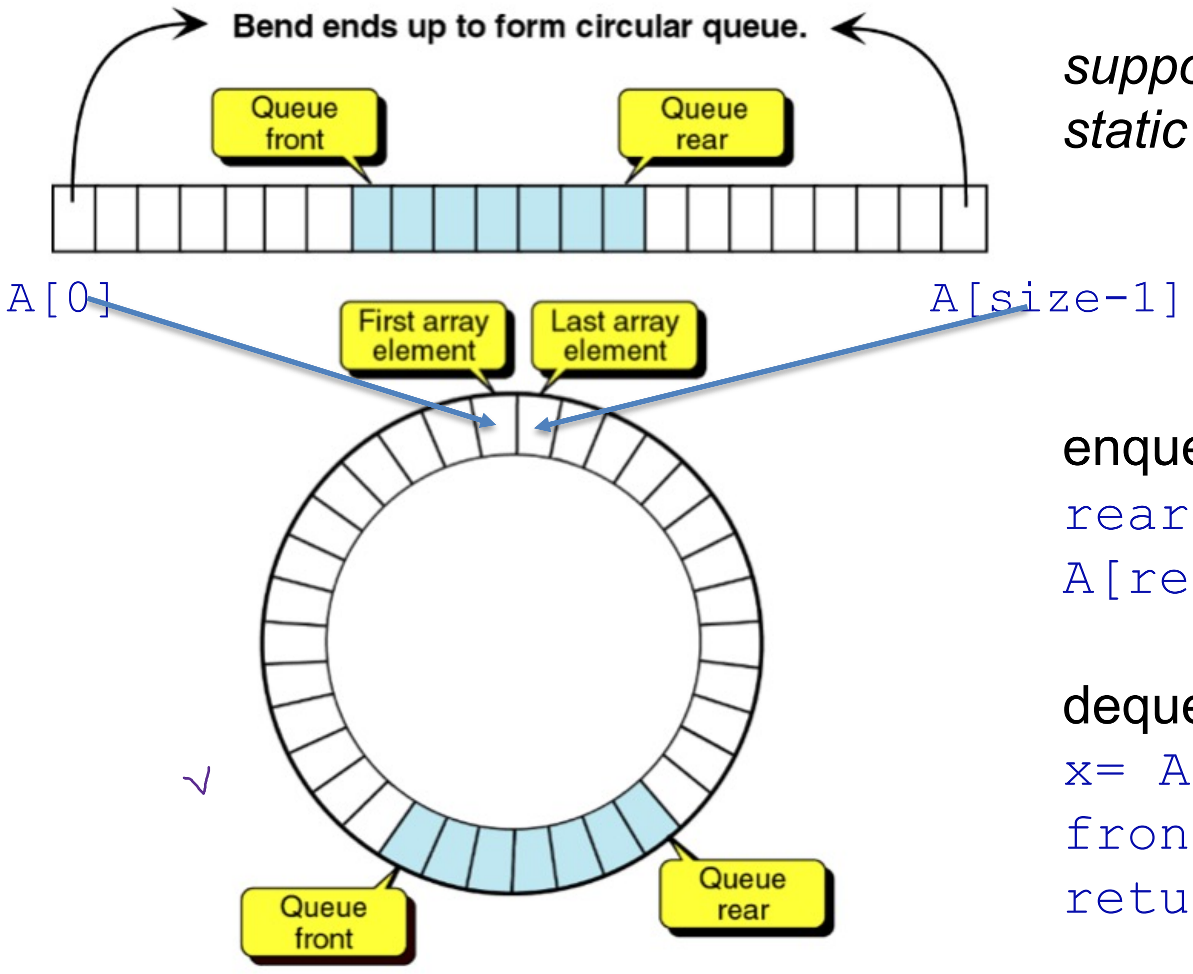


enqueue `x`: `rear= rear+1; A[rear]= x;`

dequeue: `x= A[front]; front= front+1; return x;`

any problem?

Queue: using circular arrays



suppose using a big-enough static array

enqueue `x`:

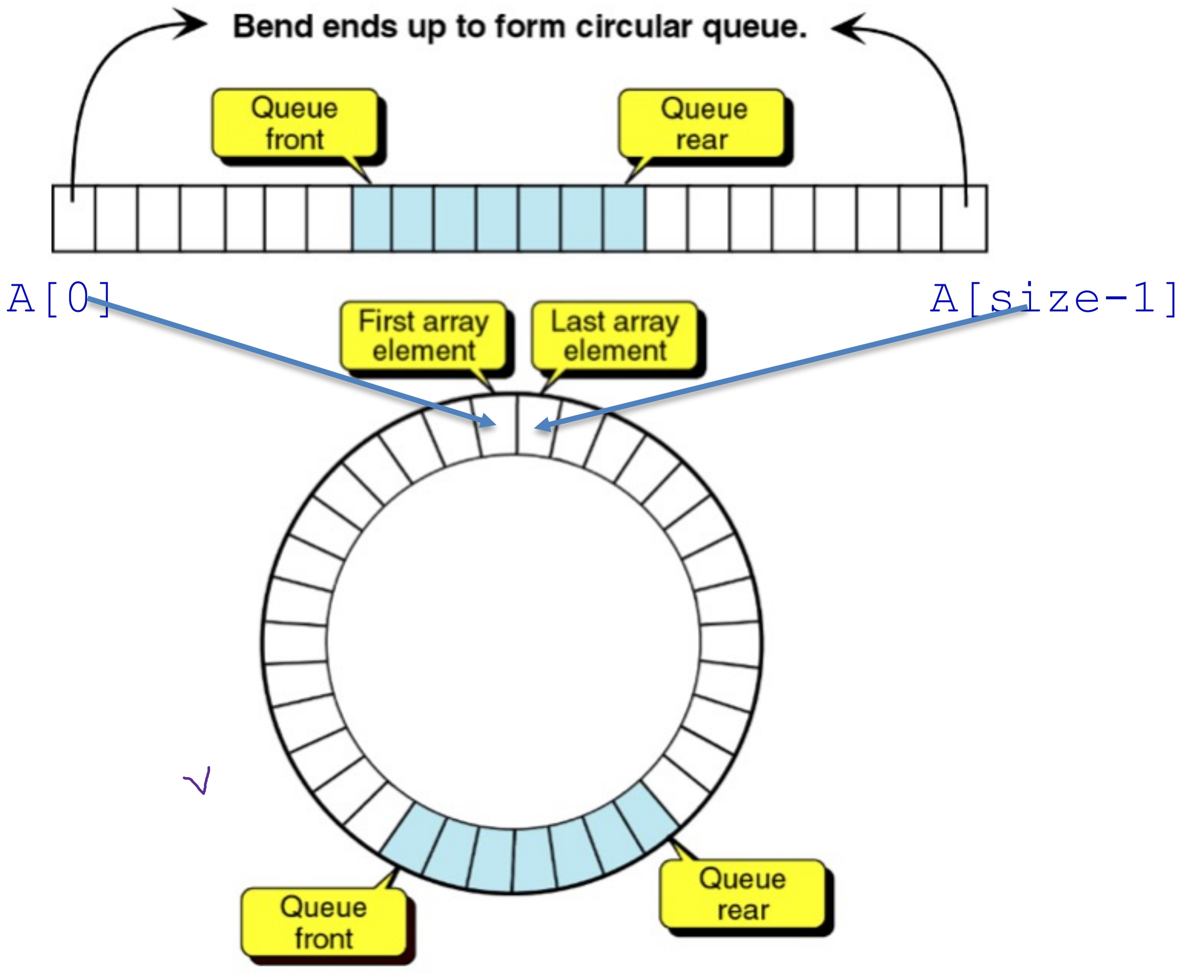
```
rear= rear+1;      ??
A[rear]= x;
```

dequeue:

```
x= A[front];
front= front+1;     ??
return x;
```

Picture source: https://www.chegg.com/homework-help/questions-and-answers/using-system-using-systemcollectionsgeneric-using-systemling-using-systemtext-using-system-q42462337

Queue: using circular arrays



suppose using a big-enough static array

enqueue `x`:

```
rear= (rear+1)%size;
A[rear]= x;
```

dequeue:

```
x= A[front];
front= (front+1)%size;
return x;
```

Picture source: https://www.chegg.com/homework-help/questions-and-answers/using-system-using-systemcollectionsgeneric-using-systemling-using-systemtext-using-system-q42462337

Peer Activity W4.10: 3 questions, then Lab Time

Finish and/or refine
Assignment 1

If Assignment 1 done:

- [Easy] get all green ticks for
  `Week 4 Workshop`
- Do exercises in
  `Week 4 Extras`