COMP20003 Workshop Week 4

Complexity + Stacks & Queues

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

Remember that
Q&A with Anh available:

- right after workshop [45 minutes]
- Friday 1PM-2PM at the FYC

LAB:

- Finish Assignment 1
- do Week 4, Week 4 Extras

Our Experiment:

- Download the slide set from
  github.com/anhvir/c203
- Use it in the workshop, add notes and etc.
- Tell Anh what you think is better?
  - today's "delivered in advance, but might lack some details" version
  - or, a normal, more-detailed version at the end of the week

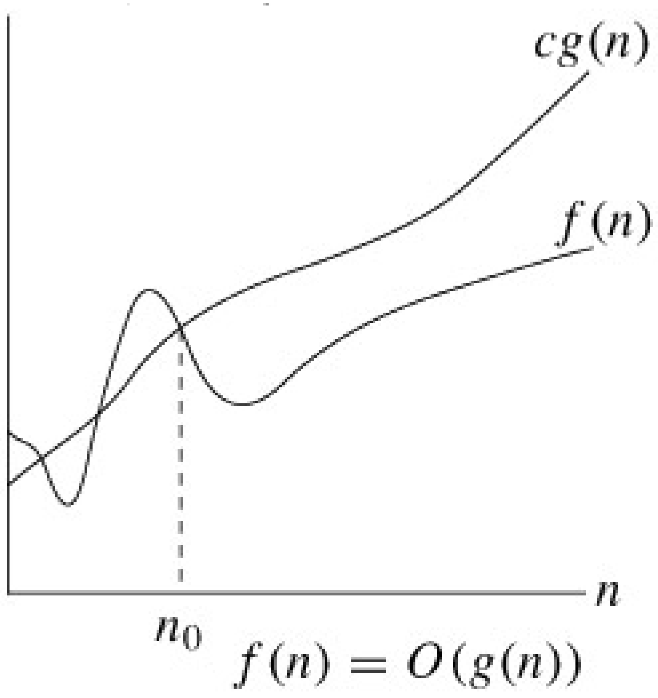
note: from next week, probably only one slide set is supplied on ED.

How fast is an algorithm?

- How to measure its the speed/efficiency of an algorithm?
- We have 2 algorithm, A and B, for solving the same problem. Which one is more efficient?
- The running time is measured as *time complexity*

Strict Big-O definition

The complexity of an algorithm is the number of operations/steps and is expressed as a function f(n) of the *input size* n.



https://web.engr.oregonstate.edu/~huanlian/teaching/cs570/

Def: We say f(n) belongs to the class O(g(n)), that is, $f(n) \in O(g(n))$, iif

- There are constants c and n_0 such 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)
- c.g(n) is an upper bound of f(n) for all large enough n

Example: $f(n)= 3n^2 + 6n + 20$

- *prove* that $f(n) \in O(n^2)$
- any other g(n) such that $f(n) \in O(f(n))$?

$$f(n) \in \begin{cases} O(\quad) \\ O(\quad) \\ O(\quad) \end{cases}$$

- We normally don't use the definition to find complexity. But if we want to prove, we need to rely on the definition.
- How?

$3n^2 + 6n + 20 \leq$

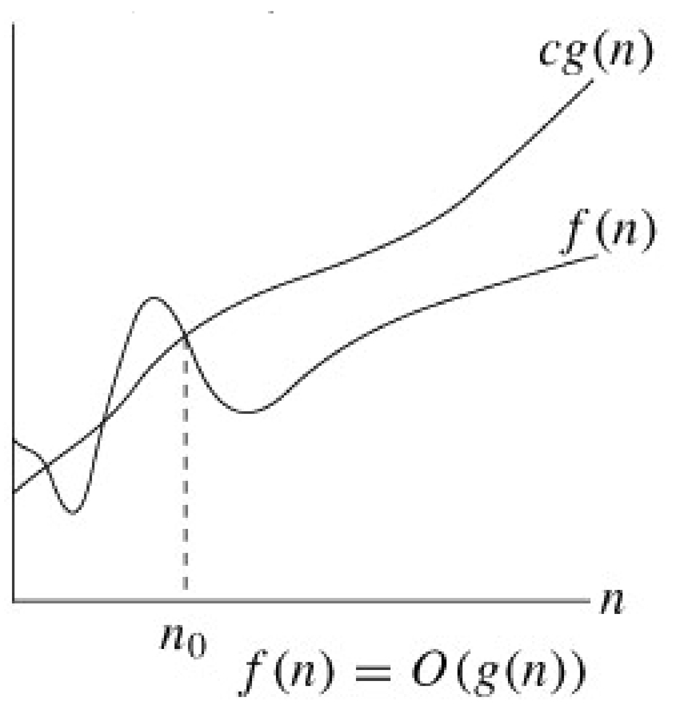
$\leq$

$\leq c.\, n^2$ for c=? , n> ?

Big-O Notation: strict definition vs. CS meaning

Def: We say f(n) belongs to the class O(g(n)), that is, $f(n) \in O(g(n))$, iif

- There are constants c and n_0 such that $f(n) \le c.g(n)$ for all $n > n_0$



https://web.engr.oregonstate.edu/~huanlian/teaching/cs570/

Underlying meaning:

- f(n) grows slower than, or as the same rate as, g(n)
- c.g(n) is an upper bound of f(n) for all large enough n

CS meaning:

- c.g(n) is *the least* upper bound of f(n) for all large enough n

Example: $f(n) = 3n^2 + 6n + 20$

- *strict Big-O:* $f(n) \in O(n^2)$, $f(n) \in O(n^3)$, $f(n) \in O(n^2.\log n)$, ...
- *CS meaning:* $f(n) \in$?

Big-Ω

The pitfall of Big-O is that it *only* describes an **upper bound** on an algorithm's running time, which often corresponds to the worst-case scenario.

<table>
<tr><th>algorithm A: max value</th><th>algorithm B: linear search</th></tr>
<tr><td><pre>int max(int A[], int n) {
  int max= A[0];
  for (int i = 1; i<n; i++)
    if (A[i] > max)
       max= A[i];
  return max;
}</pre></td><td><pre>int search(int A[], int n, int key) {

  for (int i = 0; i<n; i++)
    if (A[i] == key)
       return i;
  return NOTFOUND
}</pre></td></tr>
<tr><td>O( ) ?</td><td>does O( ) fully describe the performance?</td></tr>
</table>

We use **Ω notation** to specifically describe the **lower bound** of an algorithm's running time, which often corresponds to the best-case scenario.

When an algorithm's running time has the same upper bound and lower bound, we use **Theta (Θ) notation** to describe its tight bound.

Big-Ω

The pitfall of Big-O is that it *only* describes an *upper bound* on an algorithm's running time, which often corresponds to the worst-case scenario.

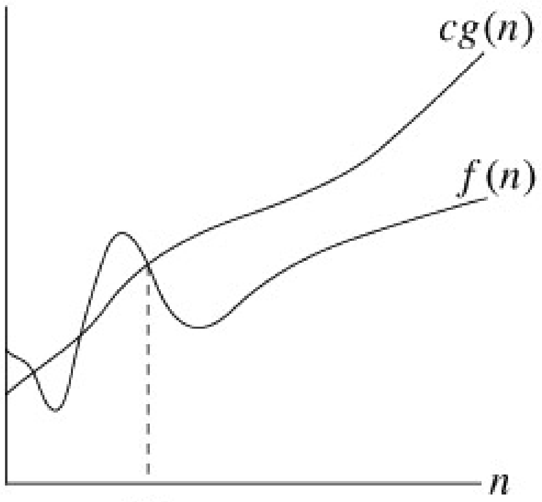
<table>
<tr><th>algorithm A: max value</th><th>algorithm B: linear search</th></tr>
<tr><td><pre>int max(int A[], int n) {
  int max= A[0];
  for (int i = 1; i<n; i++)
    if (A[i] > max)
       max= A[i];
  return max;
}</pre></td><td><pre>int search(int A[], int n, int key) {

  for (int i = 0; i<n; i++)
    if (A[i] == key)
       return i;
  return NOTFOUND
}</pre></td></tr>
<tr><td>O(n) is good, but some point missed,
Θ(n) is a stronger, more accurate</td><td>O(n) specifies the worst case
Ω(1) : in the best case, the running time is constant</td></tr>
</table>

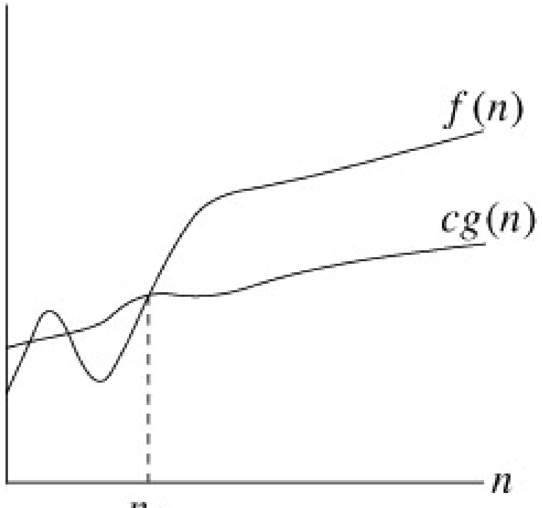
We use Ω notation to specifically describe the *lower bound* of an algorithm's running time, which often corresponds to the best-case scenario.

When an algorithm's running time has the *same upper bound and lower bound*, we use Theta (Θ) notation to describe its tight bound.

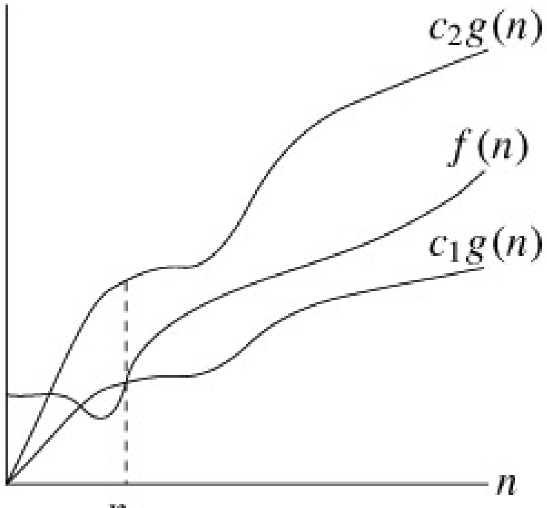
from Big-O to Big-Ω and Big-θ



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



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



$f(n) = \Theta(g(n))$

$f(n) \in O(\, g(n)\,)$

- **Def:** There are constants c and n_0 such that

  $f(n) \le c.g(n)$ for all $n > n_0$
- f(n) grows slower than, or as the same rate as, g(n)
- $f(n) \le c.g(n)$

$f(n) \in \boldsymbol{\Omega}(\, g(n)\,)$

$\Leftrightarrow \quad g(n) \in O(\, f(n)\,)$

- **Def:** There are constants c and n_0 such that

  $f(n) \ge c.g(n)$ for all $n > n_0$
- f(n) grows faster than, or as the same rate as, g(n)
- $f(n) \ge c.g(n)$

$f(n) \in \theta(\, g(n)\,)$

$\Leftrightarrow \ f(n) \in O(g(n)) \ \& \ f(n) \in \boldsymbol{\Omega}(g(n))$

- **Def:** There are constants c_1, c_2 and n_0 such that

  $c_1.g(n) \le f(n) \le c_2.g(n)$ for all $n > n_0$
- f(n) grows as the same rate as g(n)
- $c_1.g(n) \le f(n) \le c_2.g(n)$

Example: find some $\boldsymbol{\Omega}$, O and $\boldsymbol{\theta}$ for: $f(n) = 3n^2 + 6n + 20$

image source: https://web.engr.oregonstate.edu/~huanlian/teaching/cs570/

Complexity Classes

Faster growing

Order	Examples
1	Constant-time operation that does not depend on n
log n	Binary search
n	Looping through n items
n log n	Efficient sorting algorithms
n^2	Pairwise comparisons, loop within loop
⋮	
2^n	All possible subsets
n!	All possible permutations

more efficient

When working with the complexity of an algorithm, we:

- **Derive a time complexity class**, which describes how the running time scales with the input size.
- **Consider worst case and best case and use Θ or (O and Ω) notation appropriately.**

Application 1: Comparing complexity functions f(n) and g(n)

- First reduce f(n) and g(n) to their simplest form (ie. complexity classes) using Big-O arithmetic
- Then compare the classes using the increasing order: 1, $\log n$, n, $n \log n$, n^2, n^3, ..., 2^n, $n!$ (*)

Big-O Arithmetic

- $\theta(f(n)+g(n)) = \theta(\max(f(n), g(n)))$
- $\theta(c\, f(n)) = \theta(f(n))$
- $\theta(f) * \theta(g) = \theta(f*g)$

For a complexity function, we can:

- keep the most dominant term
- drop constants

Example: given

- $f(n)= 2n^2$
- $g(n)= 9n\log n + 5n + 8$

compare them in terms of complexity

Related Exercise: W4.3, W4.4

Application 2: 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.

Practically:

Any combination of memory accesses, assignments, expressions is just $\theta(1)$ if the total number of operations does not depend on the input size n

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;
```

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

Application 2: Find complexity of algorithms, including C codes. Example 1

- Single operation or memory access is 1 step
- Loops are not considered as simple operations. Instead, they are the composition of many single-step operations.

```
for (i=0; i<n; i++)
    sum = sum + A[i];
```

What's is the complexity of the above algorithm?

Application 2: Find complexity of algorithms, including C codes. Example 1

Model of computation:

- Single operation or memory access is 1 step
- Loops are not considered as simple operations. Instead, they are the composition of many single-step operations.

```
for (i=0; i<n; i++)
    sum = sum + A[i];
```

The complexity of the above algorithm $\boldsymbol{\Theta}(n)$

Here $\boldsymbol{\Theta}()$ is used instead of O() and/or $\boldsymbol{\Omega}()$

$\boldsymbol{\Theta}(f(n))$ is a strong statement because it means:

- $\Theta(f(n))$ means the algorithm's time complexity is both $O(f(n))$ and $\Omega(f(n))$
- The running time is always proportional to f(n), regardless of the specific data arrangement

Application 2: Find complexity of algorithms, including C codes. Example 2

```
// find complexity of the following linear search
for (i=0; i<n; i++) {
    if (x==A[i]) return i;
}
return -1;
```

Solutions:

Related Exercises: W4.5

Application 2: Find complexity of algorithms, including C codes. Example 2

```
// find complexity of the following linear search
for (i=0; i<n; i++) {
    if (x==A[i]) return i;
}
return -1;
```

Solutions: O(n) and **Ω**(1)

- is Ω(1) because in its best-case scenario, the running time is a constant
- is O(n) because in its worst-case scenario, the running time is bounded by a constant multiple of n

We cannot use Θ() in this case because the best-case and worst-case running times are in different complexity classes

Related Exercises: W4.5

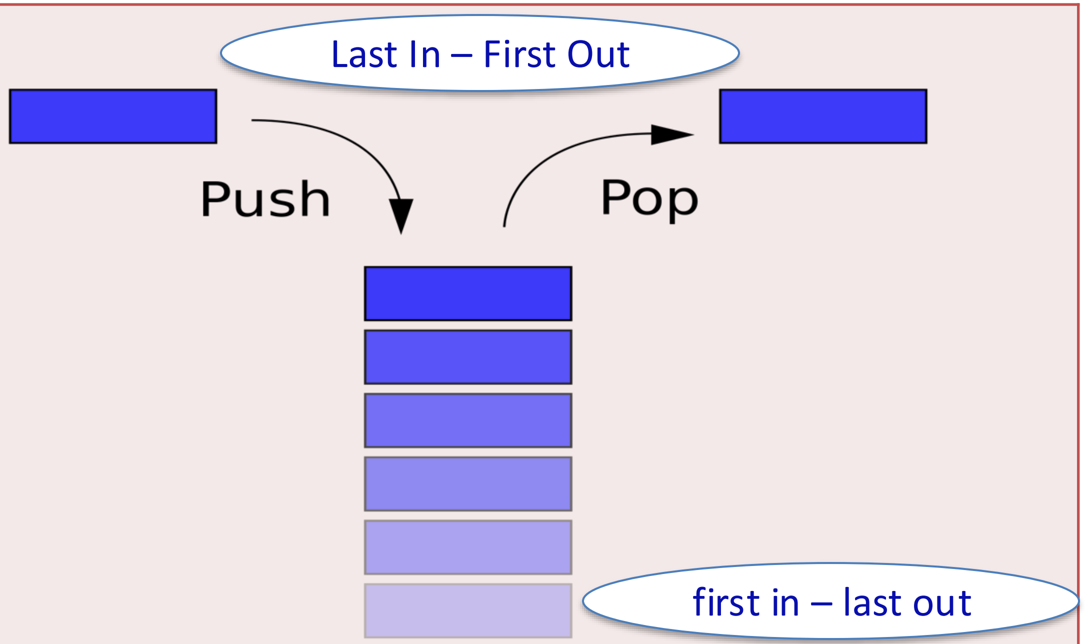
Peer Activity W4.10

What is the strongest statement on the time complexity of this code snippet?

A. O(n)
B. O($n \log n$)
C. O(n^2)
D. O($n^2 \log n$)

```
...
  int ops = 0;
  for (int i = 0; i < n - 1; i++) {
  for (j = 1 << i; j < n; j++)
  // 1<<i has value 2^i
  ops++;
  }
  ...
```

Another ADT: Stack (LIFO)



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 (and recursive 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);
}
```

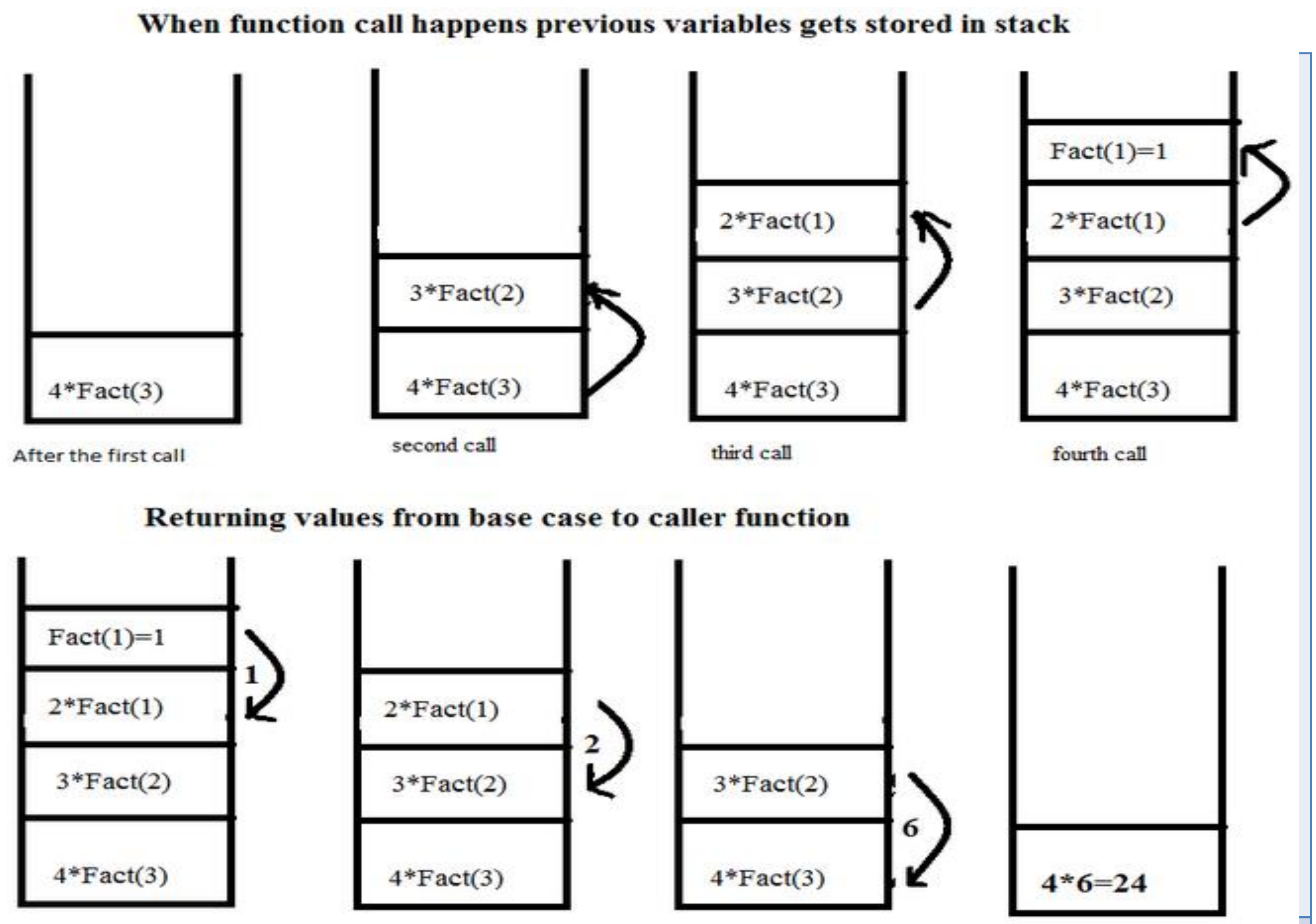
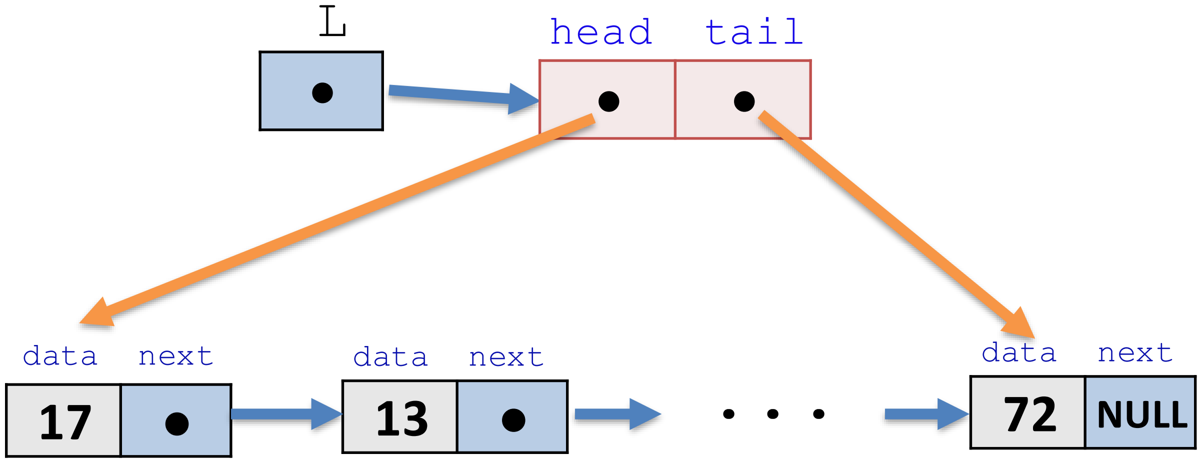


Image source: http://stackoverflow.com/questions/19865503/can-recursion-be-named-as-a-simple-function-call

Stacks: Implementation using linked lists (exercise W4.6 + more)

```
// interface list.h given in WS3-page-8
// declare struct node
// declare struct list
struct list *create();
void prepend(struct list *, int);
void  append(struct list *, int);
int deleteHead(struct list *);
int deleteTail(struct list *);
void freeList(struct list *);
```



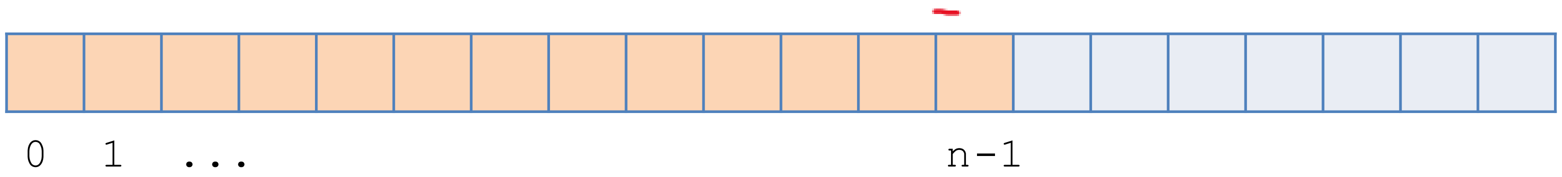

```
#include "list.h"
typedef stru

stack_ADT createStack() {

}

void push(stack_ADT s, int data){

}
```

Questions:

- How to efficiently implement Stacks using Linked Lists?
- What's the complexity of push? of pop?
- Using the above list interface, fill in the missing spaces in the RHS implementation.

Stacks: Implementation using arrays

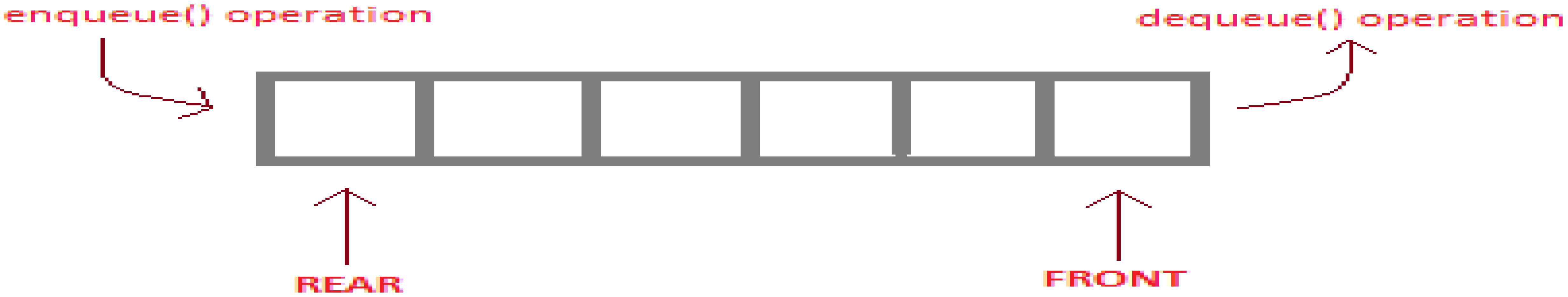
How to implement stacks using arrays?



`push(x)` ⇔ ..., complexity= ?
`pop` ⇔ ..., complexity= ?

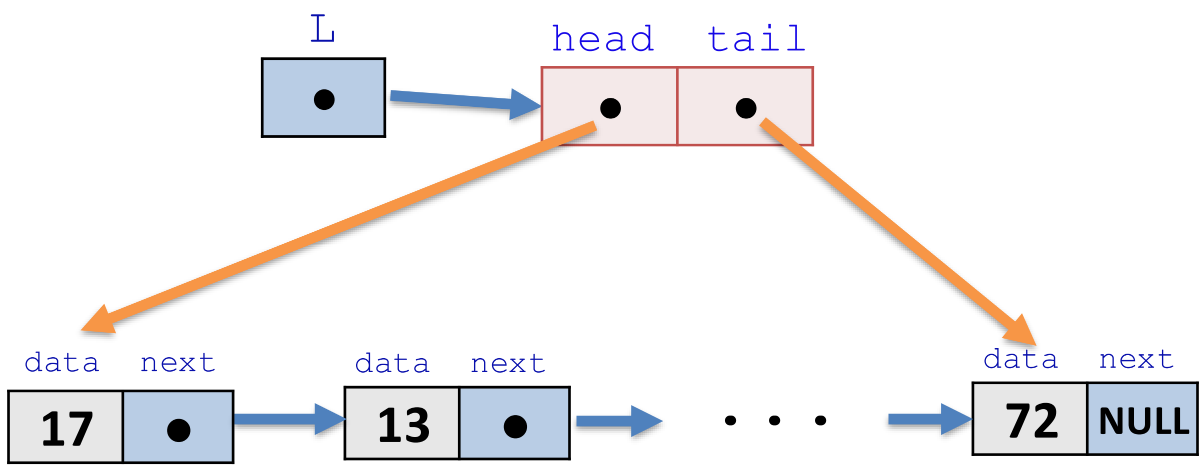
Any potential problem?

Yet another ADT: Queue (FIFO) (exercise W4.7)



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



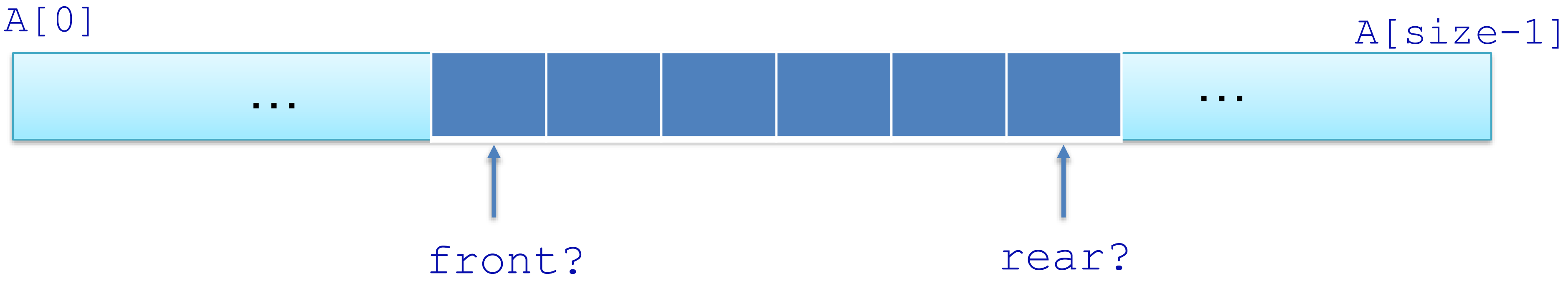
How to implement queues using Linked Lists:

enqueue ⇔

dequeue ⇔

Queue: implementation using array

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

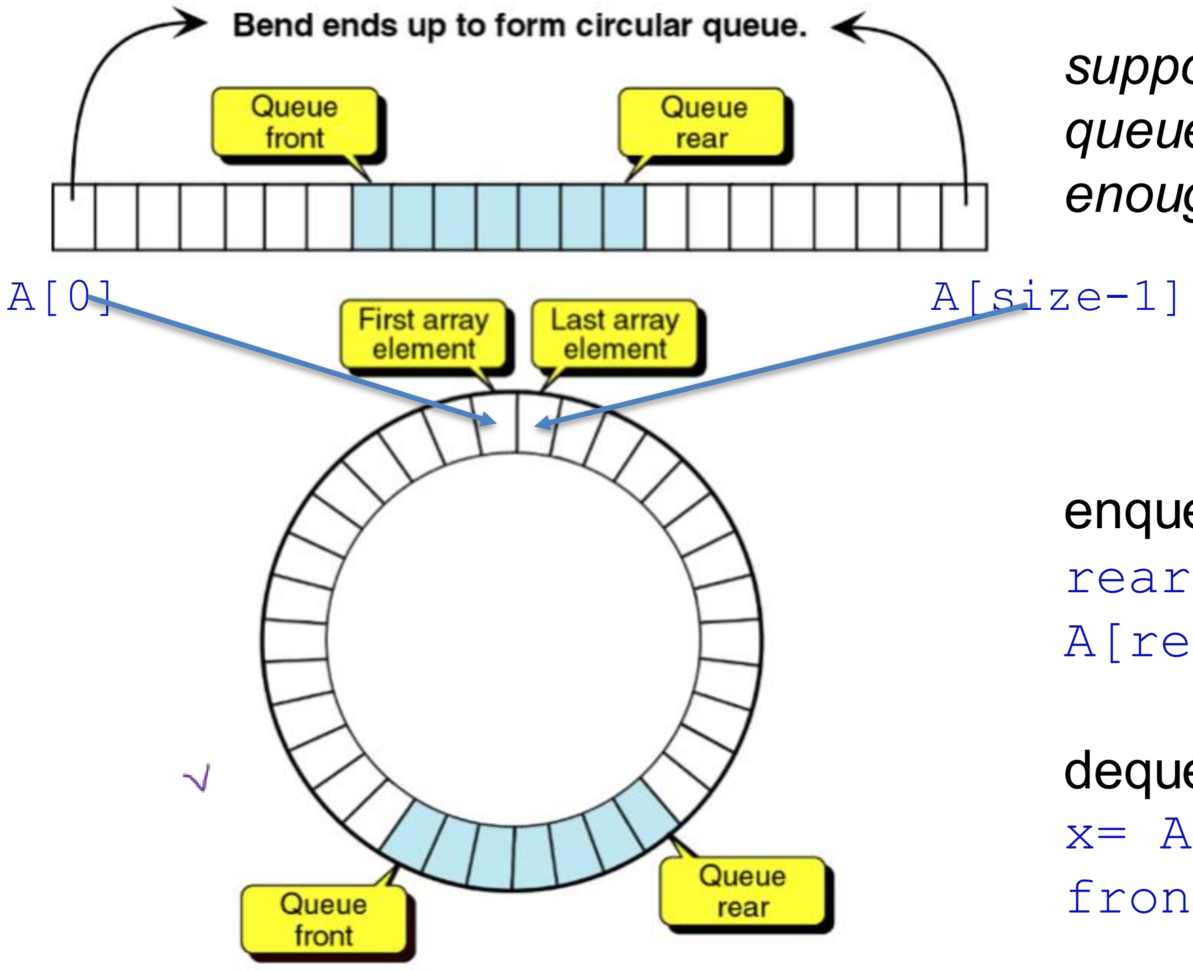


enqueue `x`: `rear= rear+1; A[rear]= x;`

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

any problem?

Queue: using circular arrays (if known the maximal size)



supposition: knowing upper bound of queue length at any time, 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

Lab Time

Finish and/or refine Assignment 1

If Assignment 1 done:

- get all green ticks for Week 4 Workshop
- Do exercises in Week 4 Extras