

COMP20003 Workshop Week 4

Complexity + Stacks & Queues

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

LAB:

- Finish Assignment 1 OR do Week 4 Extras

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 .

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

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

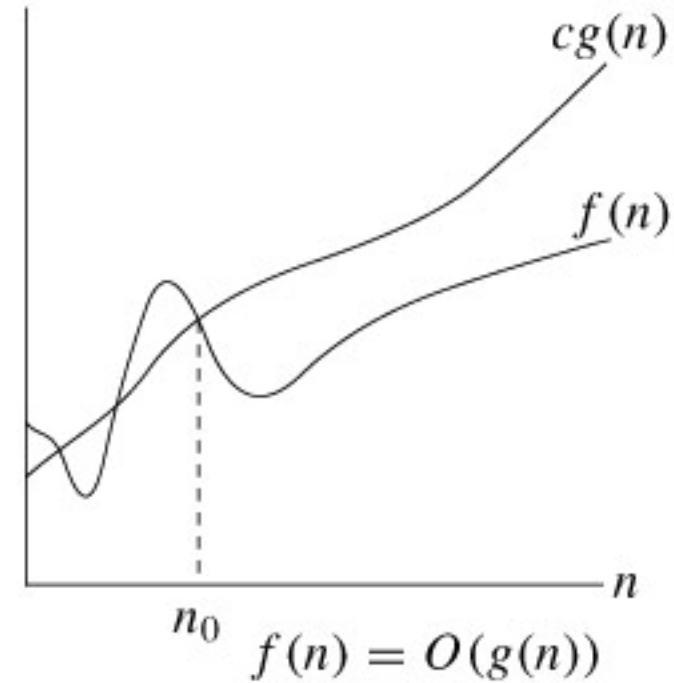
Underlying meaning:

- $f(n)$ grows slower than, or as the same rate as, $g(n)$
- $c \cdot g(n)$ 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 $f(n) \in O(g(n))$?*

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



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

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

$$\begin{aligned} 3n^2 + 6n + 20 &\leq \\ &\leq \\ &\leq c \cdot n^2 \quad \text{for } c=? , n>? \end{aligned}$$

Big-O Notation: strict definition vs. CS meaning

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

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

Underlying meaning:

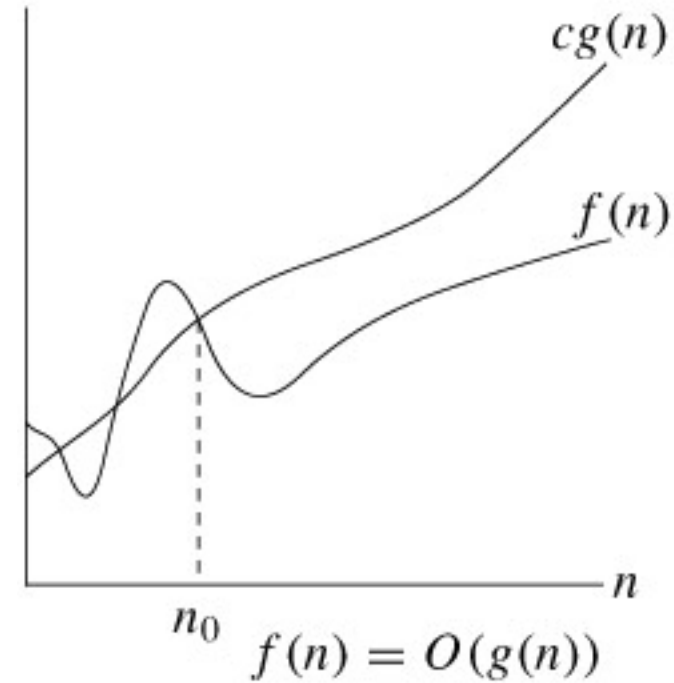
- $f(n)$ grows slower than, or as the same rate as, $g(n)$
- $c.g(n)$ is an upper bound of $f(n)$ for all large enough n

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



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

Big-Ω

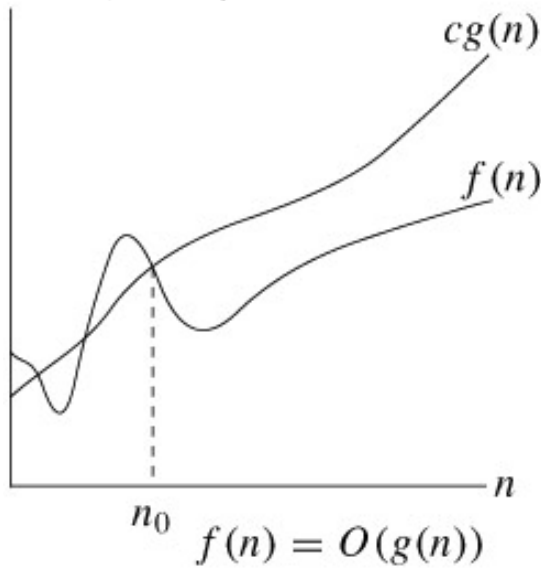
The pitfall of Big-O is that it *only* describes an **upper bound** on an algorithm's running time, which often corresponds to the worst-case scenario.

algorithm A: max value	algorithm B: linear search
<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>	<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>
$O()$?	does $O()$ fully describe the performance?

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.

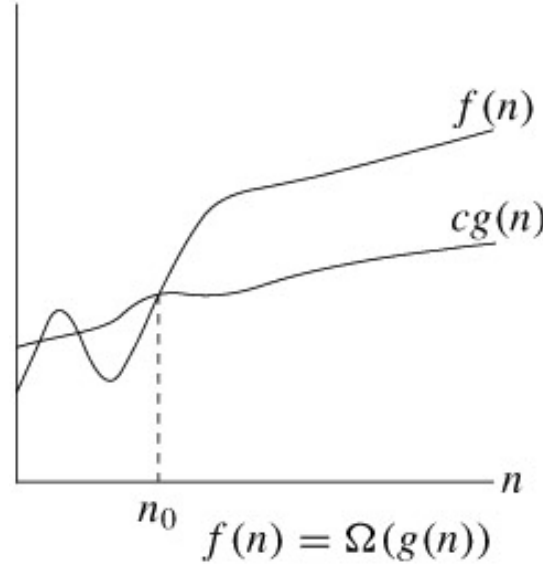
When an algorithm's running time has the *same upper bound and lower bound*, we use **Theta (Θ) notation** to describe its tight bound.



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

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

$$f(n) \leq c \cdot g(n) \text{ for all } n > n_0$$
- $f(n)$ grows slower than, or as the same rate as, $g(n)$
- $f(n) \leq c \cdot g(n)$

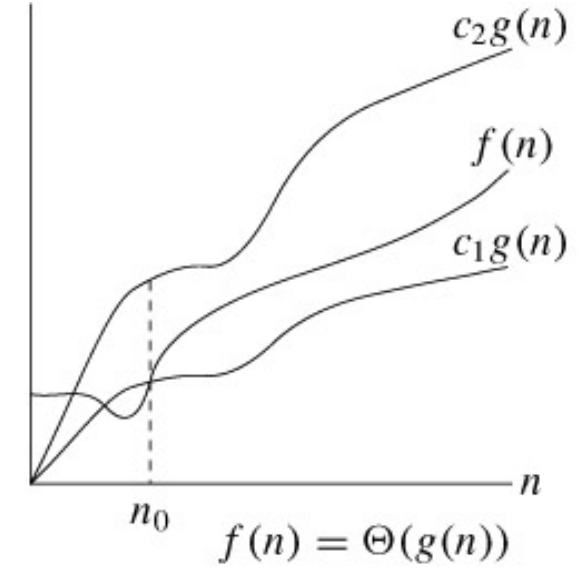


$$f(n) \in \Omega(g(n))$$

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

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

$$f(n) \geq c \cdot g(n) \text{ for all } n > n_0$$
- $f(n)$ grows faster than, or as the same rate as, $g(n)$
- $f(n) \geq c \cdot g(n)$



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

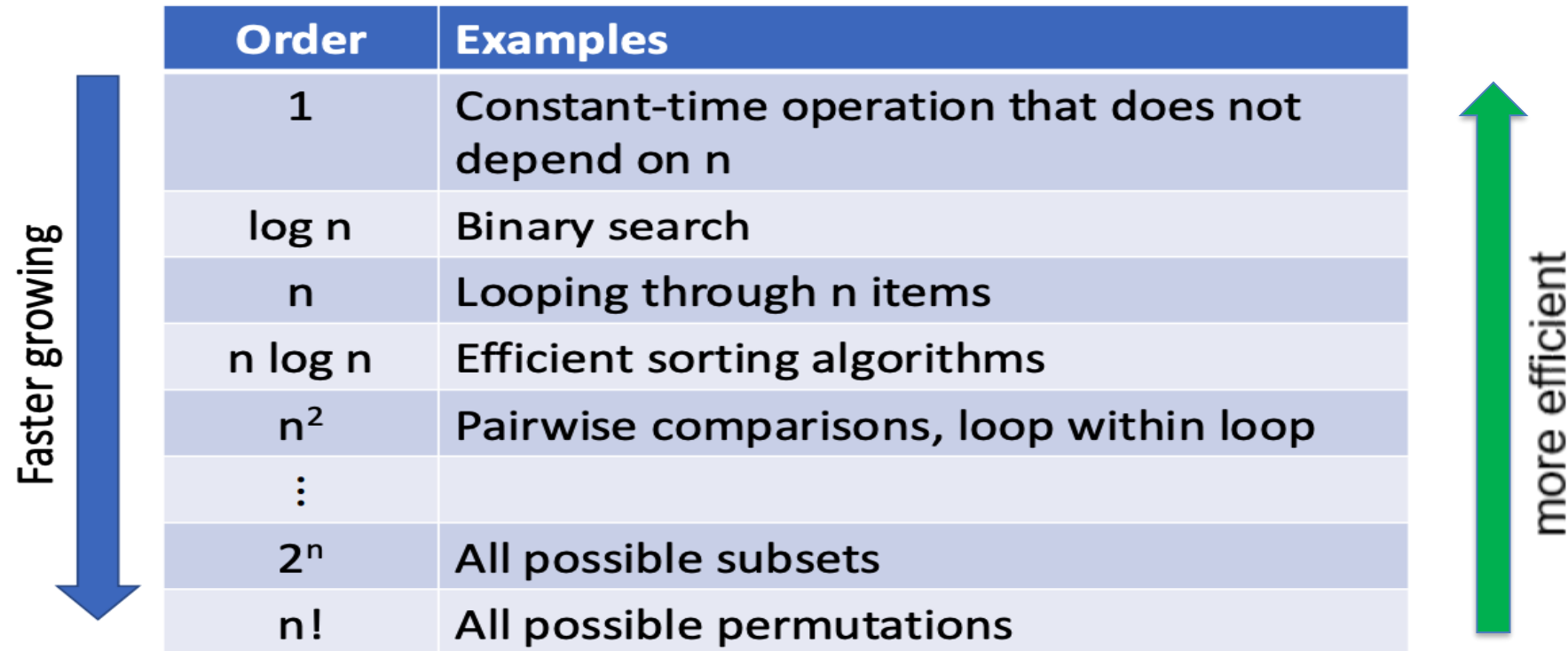
$$\Leftrightarrow f(n) \in O(g(n)) \text{ \& } f(n) \in \Omega(g(n))$$

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

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n > n_0$$
- $f(n)$ grows as the same rate as $g(n)$
- $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

Example: find some Ω , O and θ for: $f(n) = 3n^2 + 6n + 20$

Complexity Classes



The diagram features a central table with two columns: 'Order' and 'Examples'. To the left of the table is a blue arrow pointing downwards, labeled 'Faster growing'. To the right is a green arrow pointing upwards, labeled 'more efficient'. The table rows, from top to bottom, are: 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), n² (Pairwise comparisons, loop within loop), ⋮, 2ⁿ (All possible subsets), and n! (All possible permutations).

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
\vdots	
2^n	All possible subsets
$n!$	All possible permutations

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/Skienna:

- 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.

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

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

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

1	// find complexity of the following linear search
2	for (i=0; i<n; i++) {
3	if (x==A[i]) return i;
4	}
5	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
1	for (i=0; i<n; i++) {
2	if (x==A[i]) return i;
3	}
4	return -1;

Related Exercises: W4.5

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. $O(n^2 \log n)$

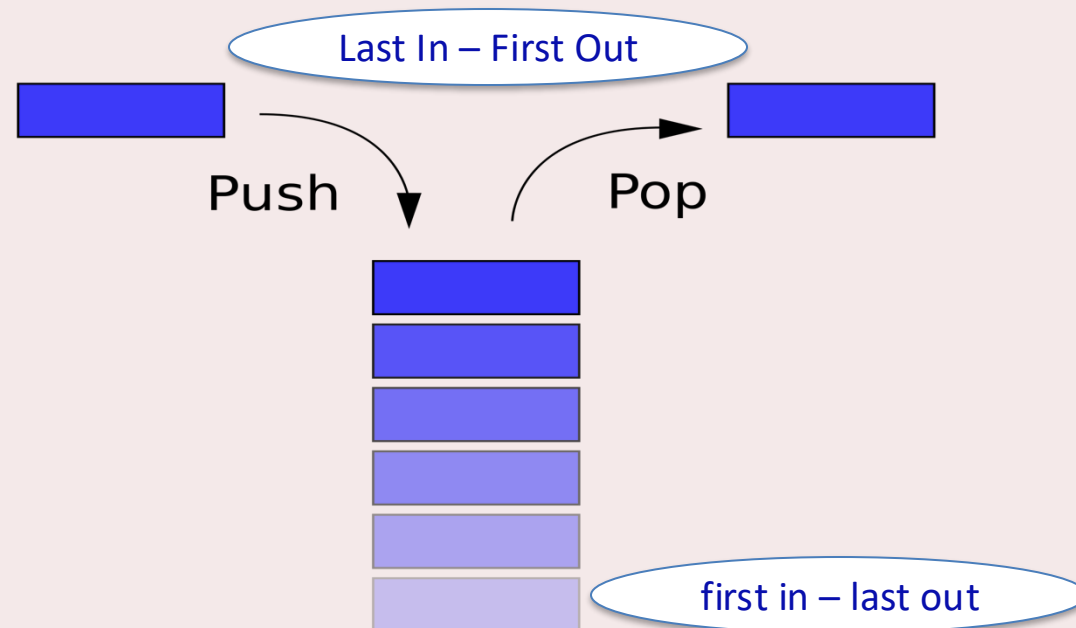
```
...  
    int ops = 0;  
    for (int i = 0; i < n - 1; i++) {  
        for (j = 1 << i; j < n; j++)  
            // 1<<i has value 2i  
            ops++;  
    }  
...
```

Another ADT: Stack (LIFO)



<http://www.123rf.com/stock-photo/tyre.html>

Stack
Operations



[https://simple.wikipedia.org/wiki/Stack_\(data_structure\)](https://simple.wikipedia.org/wiki/Stack_(data_structure))

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

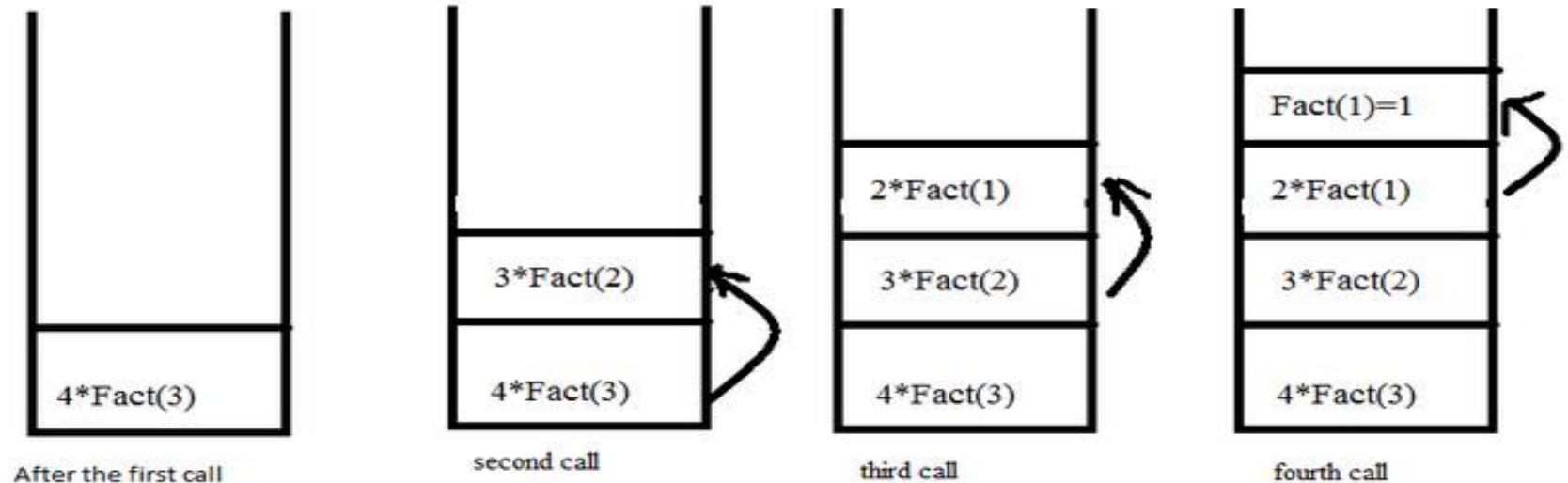
Stack is widely used in implementation of programming systems. For example, compilers employ stacks for keeping track of function calls and execution.

Stack for :

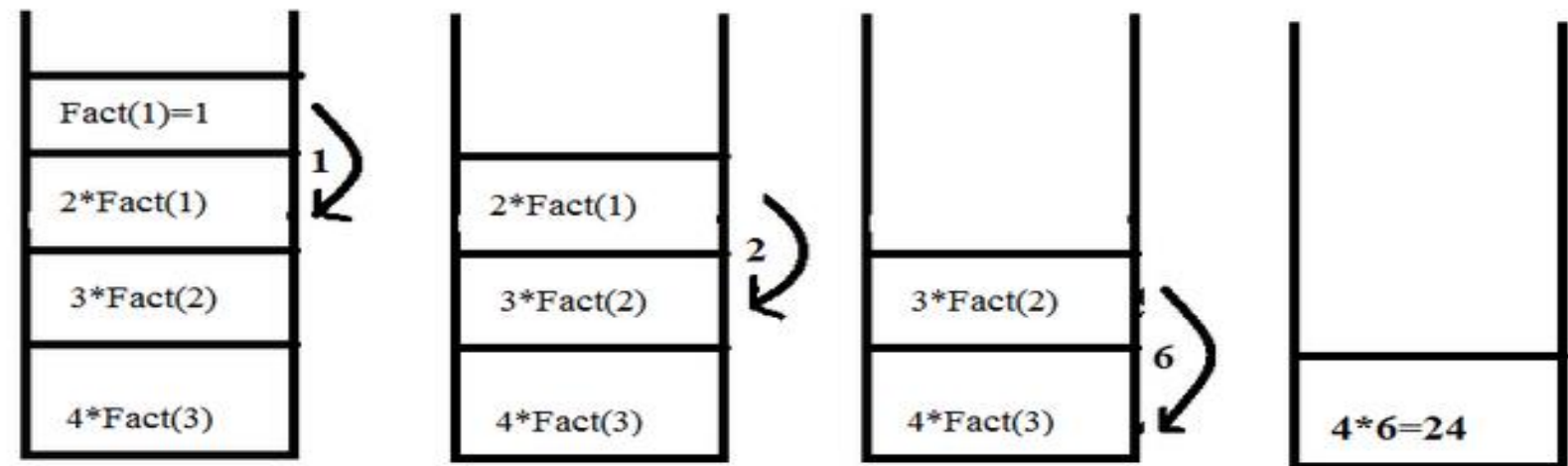
`fact(4)`

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

When function call happens previous variables gets stored in stack

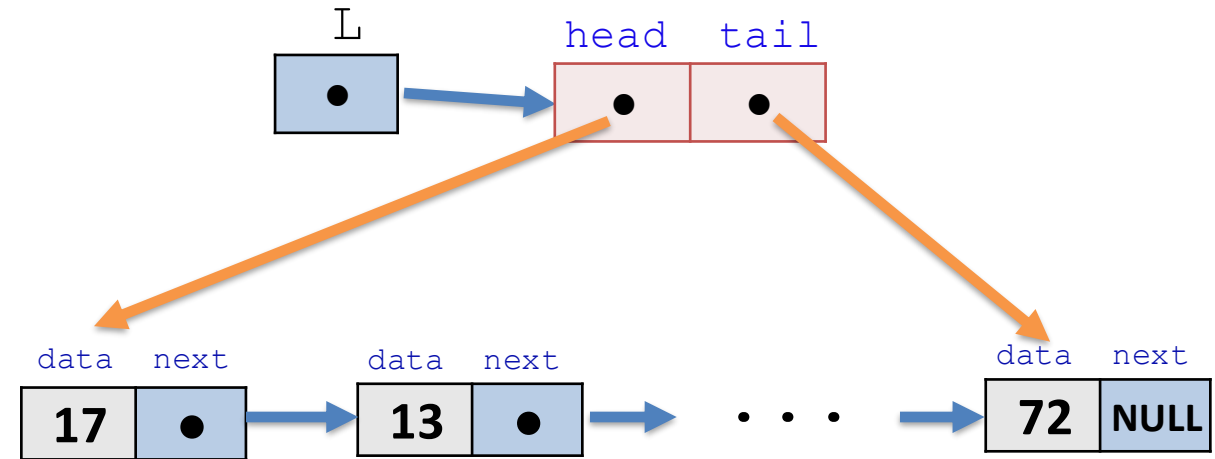


Returning values from base case to caller function



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 *);
```



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.

```
#include "list.h"
```

```
typedef struct
```

```
stack_ADT createStack() {
```

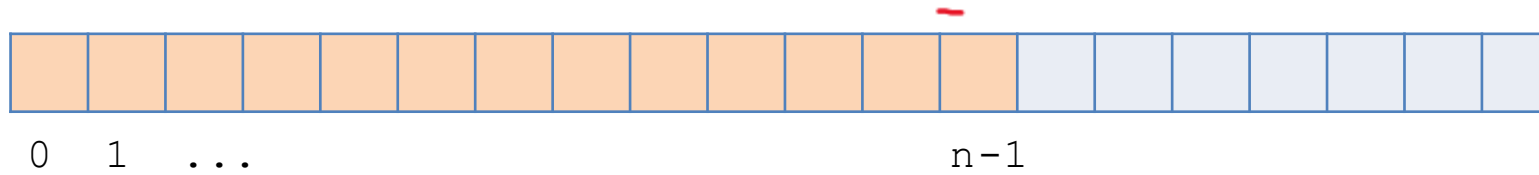
```
}
```

```
void push(stack_ADT s, int data){
```

```
}
```

Stacks: Implementation using arrays

How to implement stacks using arrays?



`push(x)` \Leftrightarrow ... , complexity= ?
`pop` \Leftrightarrow ... , complexity= ?

Any potential problem?

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



enqueue() operation

dequeue() operation



REAR

FRONT

Queue Operations

enqueue (x) : add **x** to the rear of the queue

dequeue () : remove (and return) the element at front

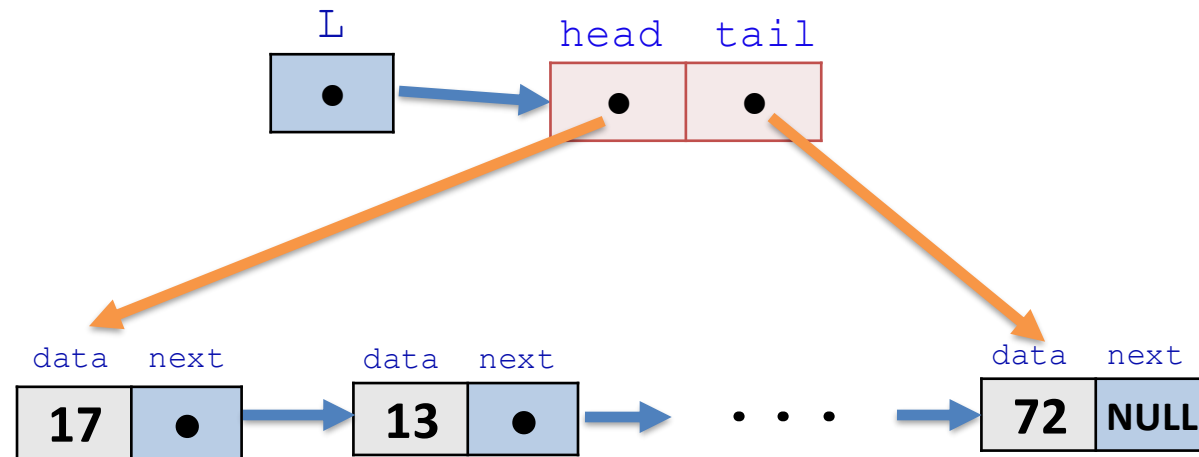
create () : create a new, empty queue

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



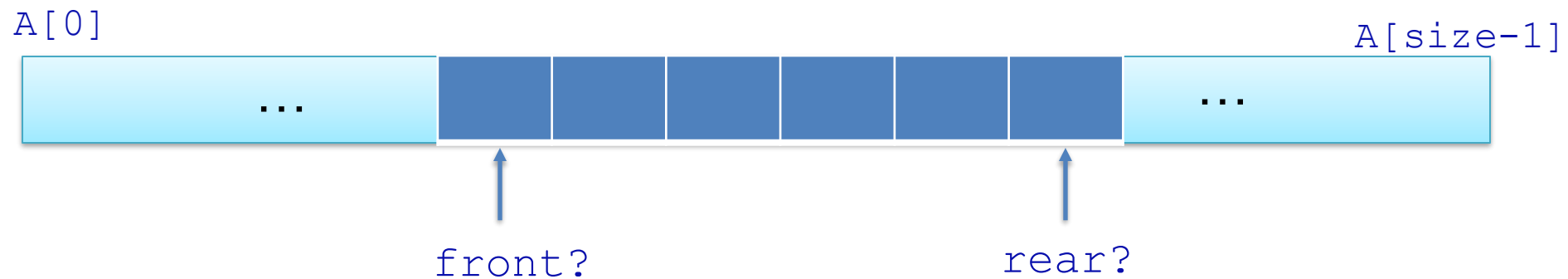
How to implement queues using Linked Lists:

`enqueue` \Leftrightarrow

`dequeue` \Leftrightarrow

Queue: implementation using array

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

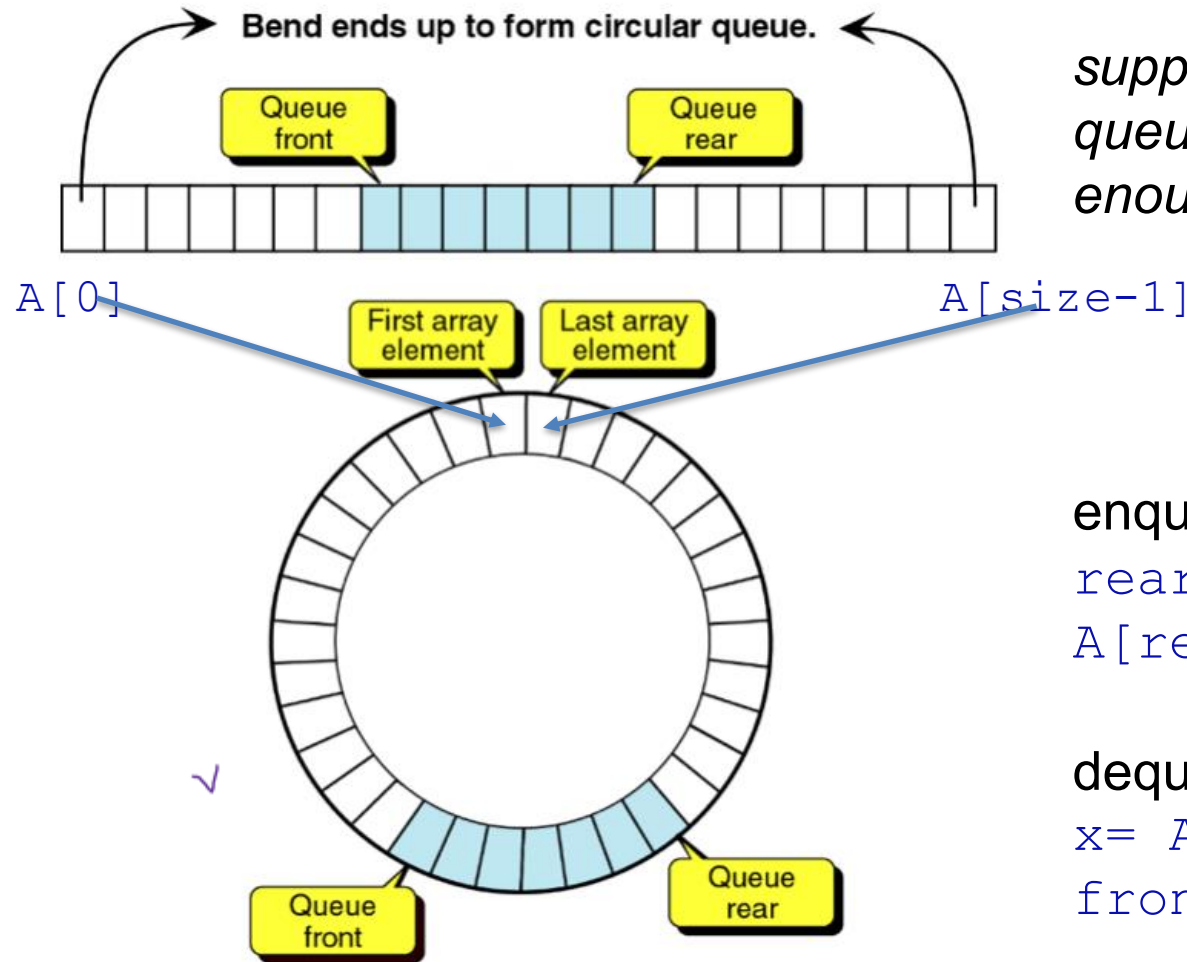


enqueue `x`: `rear= rear+1; A[rear]= x;`

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;
```

Finish and/or refine
Assignment 1

If Assignment 1 done:

- get all green ticks for [Week 4 Workshop](#)
- Do exercises in [Week 4 Extras](#)