COMP20007 Workshop Week 2

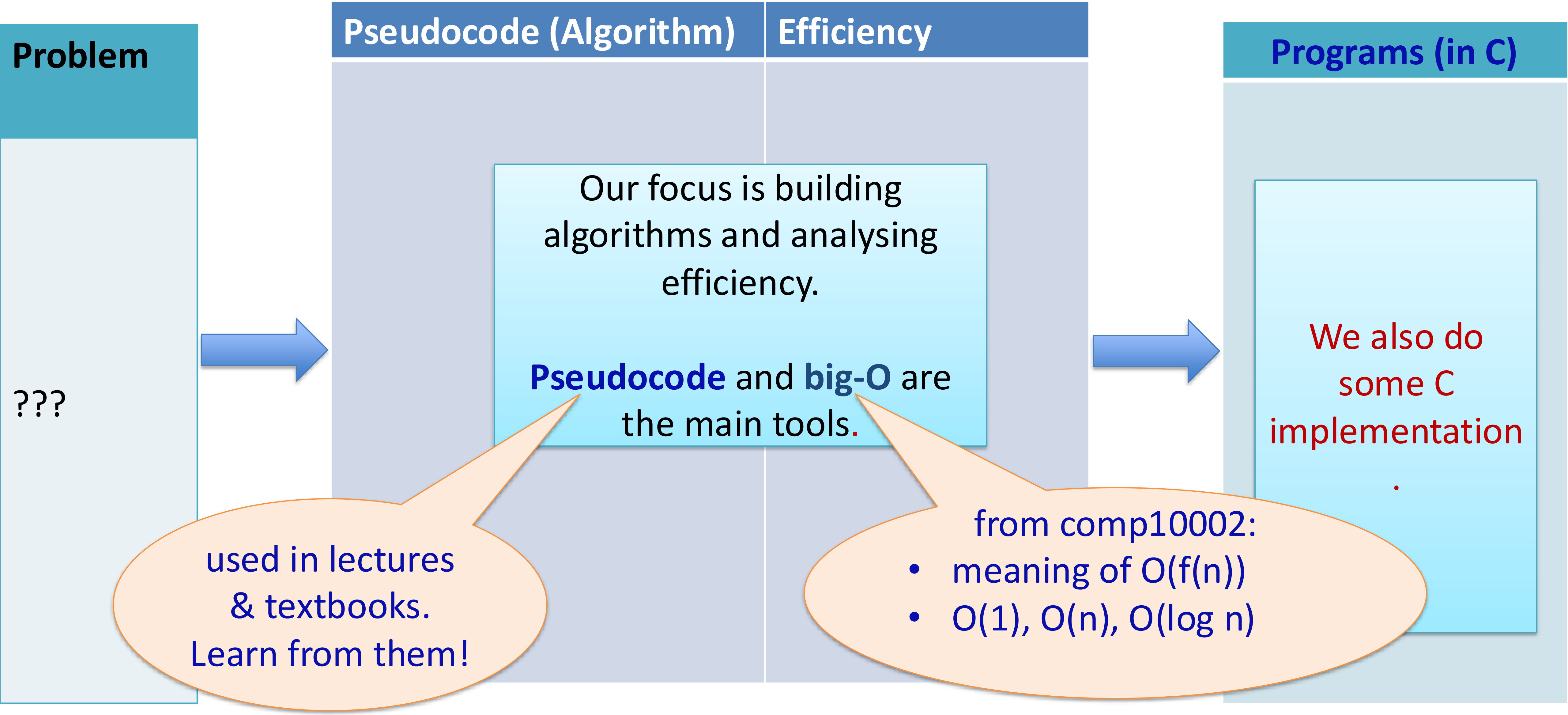
Basic Data Structures & ADT + more on C

1. Arrays & Linked Lists
2. Stacks & Queues
3. Lab:
   - C: Memory Pool, Dynamic Arrays (again?)
   - Using Linked Lists (a concrete DS) to build the stack ADT
   - other exercises

For Effective Workshops

- Learning by Doing
- Collaborating with Classmates
- Always Having Pens and Papers (or Equivalent Tools) Ready

COMP20007 focuses on algorithm design and efficiency



Problem

???

Pseudocode (Algorithm)

Efficiency

Our focus is building algorithms and analysing efficiency.

Pseudocode and **big-O** are the main tools.

used in lectures & textbooks. Learn from them!

from comp10002:

- meaning of O(f(n))
- O(1), O(n), O(log n)

Programs (in C)

We also do some C implementation
.

examples

Problem

Having a collection of data

Need to search for some specified elements

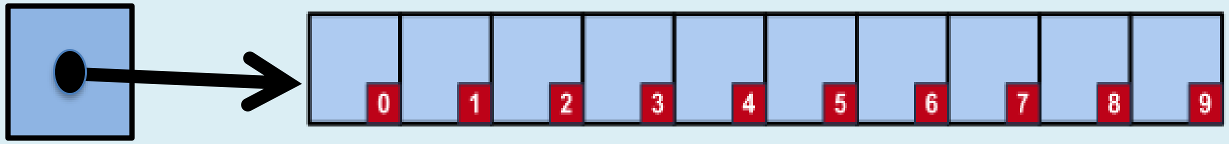
Algorithm	Efficiency	When Good?
• get data into an array • do sequential searches		
• get data into an array • sort the array • do binary searches		
…		

Programs

Topic 1: Arrays vs Linked Lists: random access and sequential access to k-th element

ARRAY

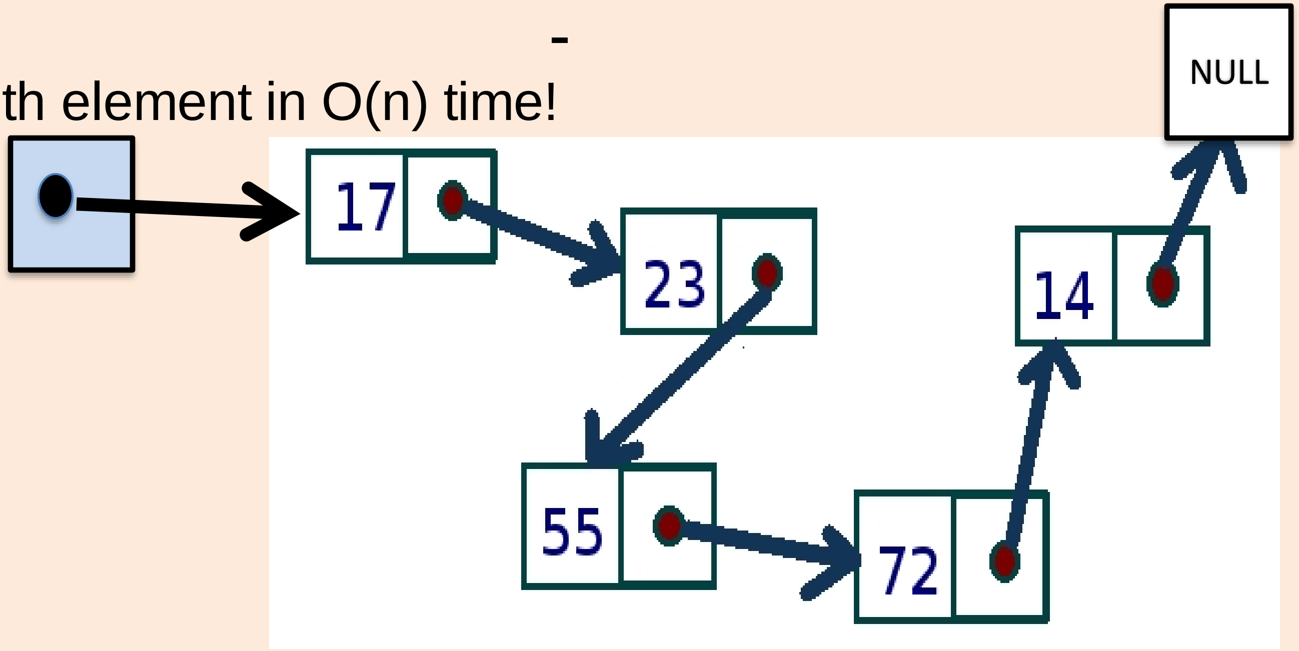
Access A[k] in O(1) time!



A (A= malloc(10 * sizeof(*A);)

LINKED LIST

Access k-th element in O(n) time!



Comparing arrays and linked lists. Which one is more efficient for:

- accessing the k-th element?
- inserting a new element at the start?
- deleting the first element?
- searching for a key?

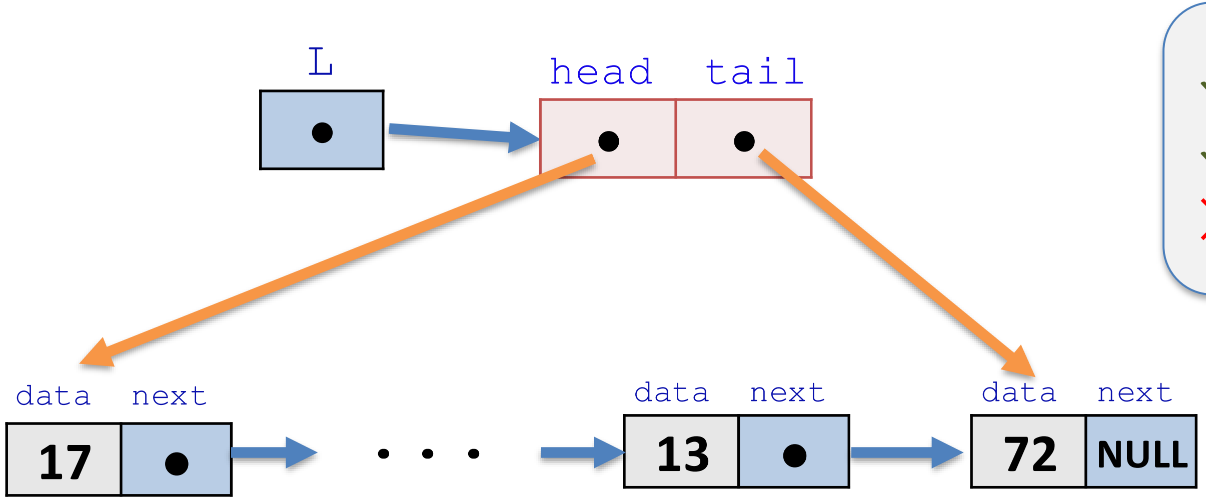
Define Linked Lists – a popular method

Linked list is a pair of pointers.

```

typedef struct node node_t;
struct node {
    data_t data;
    node_t *next;
} ;
```

```
typedef struct {
    node_t *head;
    node_t *tail;
} list_t;
list_t *L= createList();
...
```



✓ Efficient insert/delete at the start

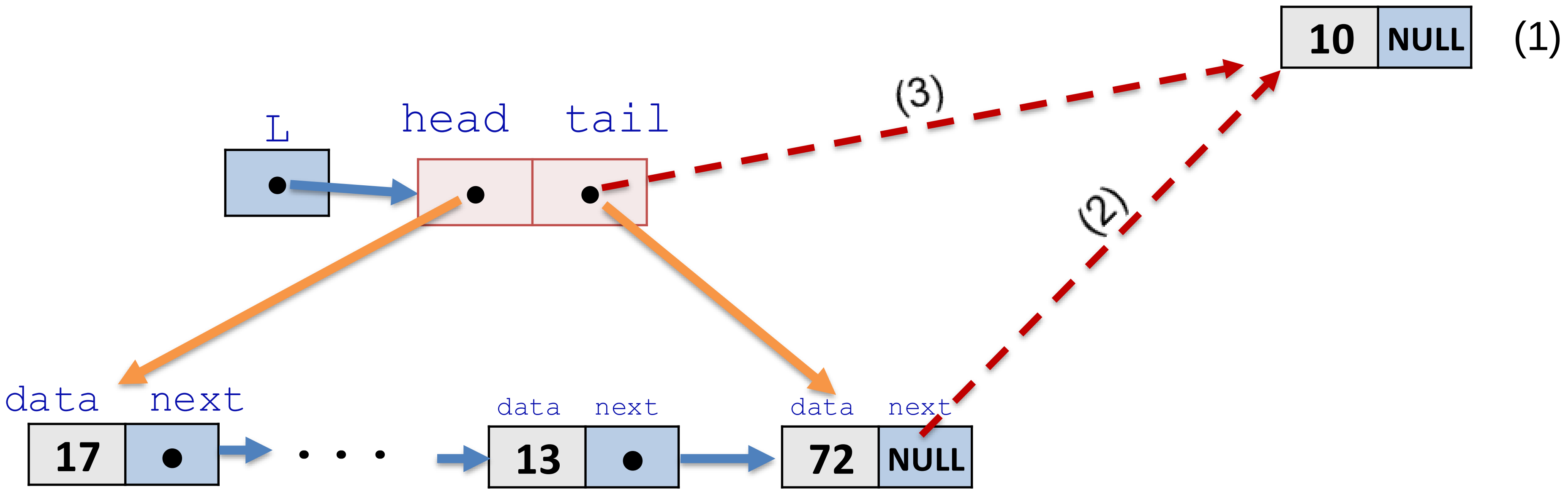
✓ Efficient insert at the end

✕ Inefficient delete at the end

Linked List Example: `insert_last` : append node with data 10 – *is the code correct?*

1. create new node and set data	`node_t *new= malloc(sizeof(*new));`
	`new->data= 10;    // here, data_t is int`
	`new->next= NULL;`

2. Link the node to the chain	`L->tail->next= new;`
3. Repair `tail`	`L->tail= new;`



Q 1: Arrays

Describe how you could perform the following operations on *sorted* and *unsorted arrays*, and decide if they are *O(1)*, *O(log n)*, or *O(n)*, where *n* is the number of elements initially in the array. Assume that there is no need to change the size of the array to complete each operation:

- Inserting a new element
- Searching for a specified element
- Deleting the final element
- Deleting a specified element

How to answer? What's expected?

Q2.1: Arrays

Describe how you could perform the following operations on *sorted* and *unsorted arrays*, and decide if they are *O(1)*, *O(log n)*, or *O(n)*, where *n* is the number of elements initially in the array. Assume that there is no need to change the size of the array to complete each operation.

Operation	Unsorted Arrays	Sorted Arrays
Searching for a specified element	**O(???)** - how-to ?	**O( )** -
Inserting a new element	**O( )** -	**O( )** -
Deleting the final element	**O( )** -	**O( )** -
Deleting a specified element	**O( )** -	**O( )** -

Q 2: Linked Lists

Describe how you could perform the following operations on singly-linked and doubly-linked lists, and decide if they are *O(1)*, *O(log n)*, or *O(n)*, where *n* is the number of elements initially in the linked list. Assume that the lists need to keep track of their final element.

Operation	Singly	Doubly
Inserting a node at the start	**O( )** - how-to:	
Inserting a node at the end		
Deleting the first node (at the start)		
Deleting last node (at the end)		

In general:

- *What complexity?*
- *What should be considered when deleting/inserting?*

Topic 2: Abstract Data Types, Stacks & Queues

- Compare

arrays
and
linked lists

?

stacks
and
queues

Topic 2: Stacks & Queues

- Compare

arrays
and
linked lists:

concrete data structures

?

stacks
and
queues:

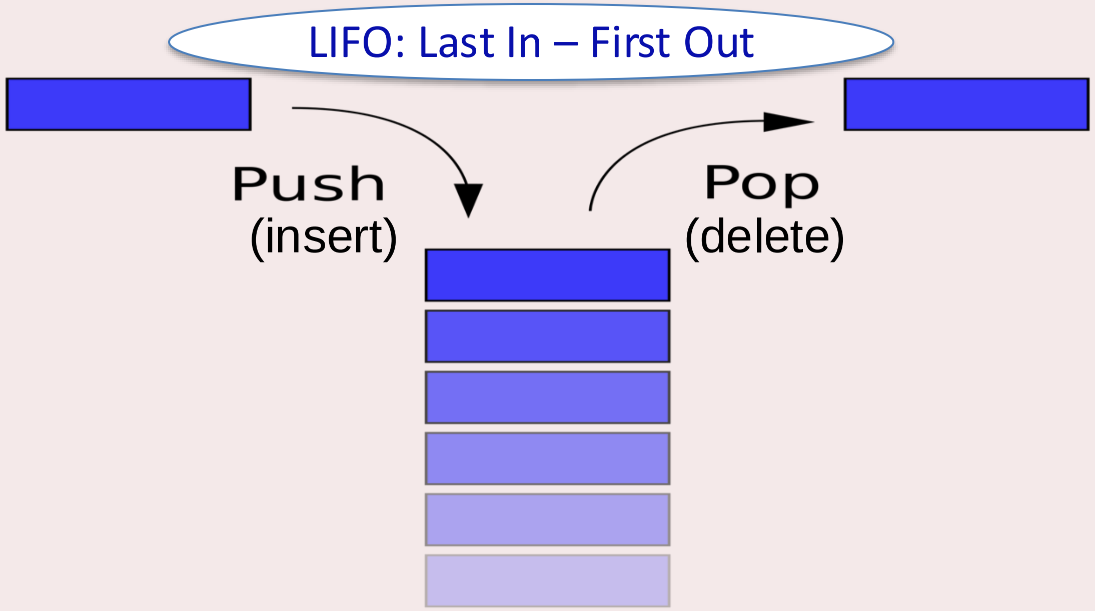
Abstract Data Types

HOW?
- representation in memory
- implementation of related operations

WHAT?
- only interface of related operations

An Abstract Data Type (ADT): Stack (LIFO)

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



LIFO: Last In – First Out

Push (insert)

Pop (delete)

adapted from https://simple.wikipedia.org/wiki/Stack_(data_structure)

Stack Operations

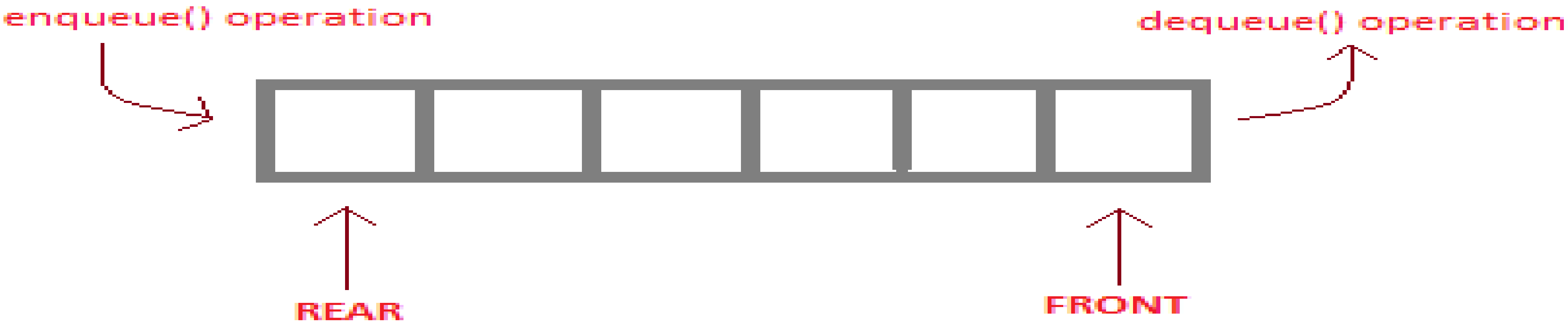
push(x) : insert element x to (the top of) stack

pop() : remove and return an element from (the top of) stack

isEmpty() : check if stack is empty

create() : create a new, empty stack

Another ADT: Queue (FIFO)



enqueue() operation

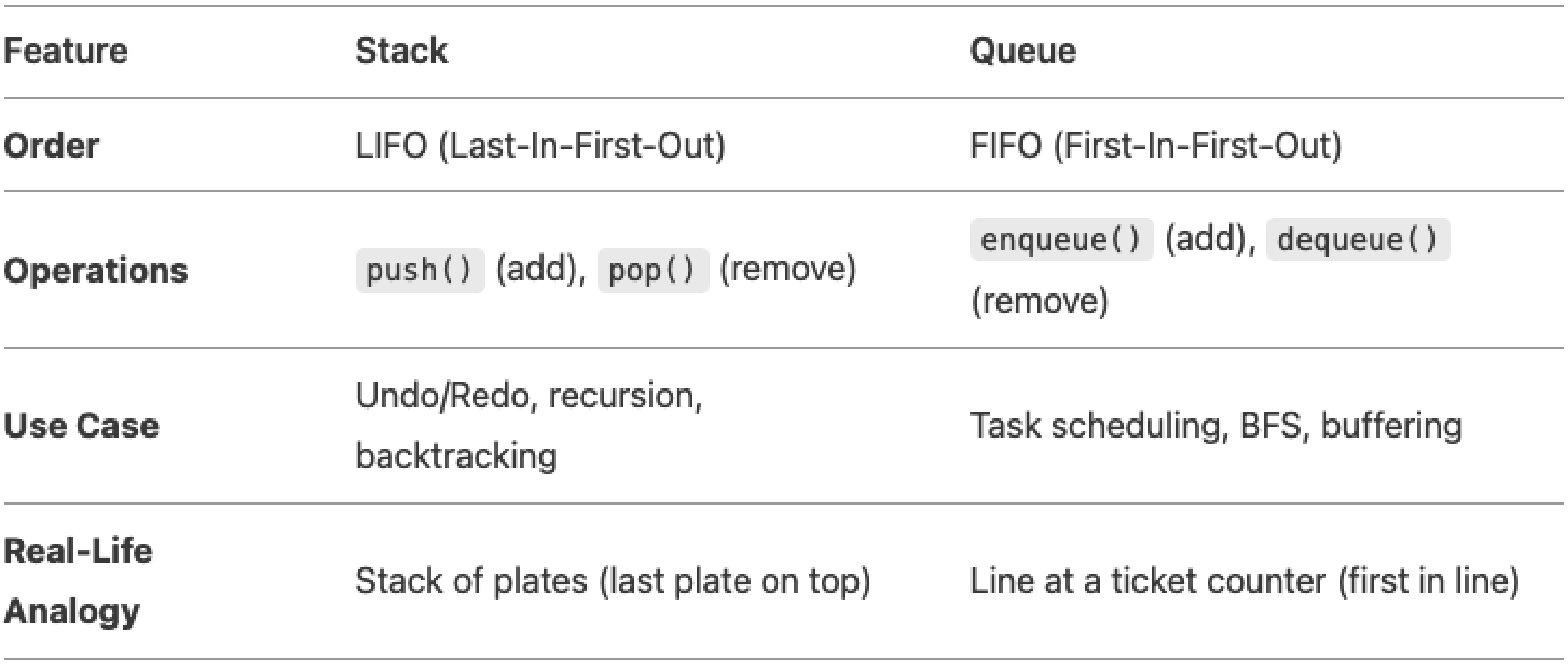
dequeue() operation

REAR

FRONT

Queue Operations	**enqueue(x) :** add **x** to (the rear of) the queue **dequeue() :** remove and return the element from (the front of) the queue **create() :** create a new, empty queue **isEmpty() :** check if queue is empty

Key Differences Between Stacks and Queues



Feature	Stack	Queue
Order	LIFO (Last-In-First-Out)	FIFO (First-In-First-Out)
Operations	`push()` (add), `pop()` (remove)	`enqueue()` (add), `dequeue()` (remove)
Use Case	Undo/Redo, recursion, backtracking	Task scheduling, BFS, buffering
Real-Life Analogy	Stack of plates (last plate on top)	Line at a ticket counter (first in line)

When to Use Stacks vs. Queues

- Use a stack when we need to track the most recent actions or reverse the order of operations (e.g., undo, recursion).
- Use a queue when we need to process tasks in the order they arrive (e.g., scheduling, BFS).

Peer Activity: Evaluating Expressions

Which abstract data structure is most suitable for evaluating this arithmetic expression? Why?

a. dictionary

b. queue

c. stack

d. none of the above

$$a \times (b + (c - d))$$

Q 3: Stacks

Describe how to implement `push` and `pop` using

- an unsorted array?
- a singly-linked list?

Using an (unsorted) array	**Using a (singly-)linked list**

What if the array is full before `push`?

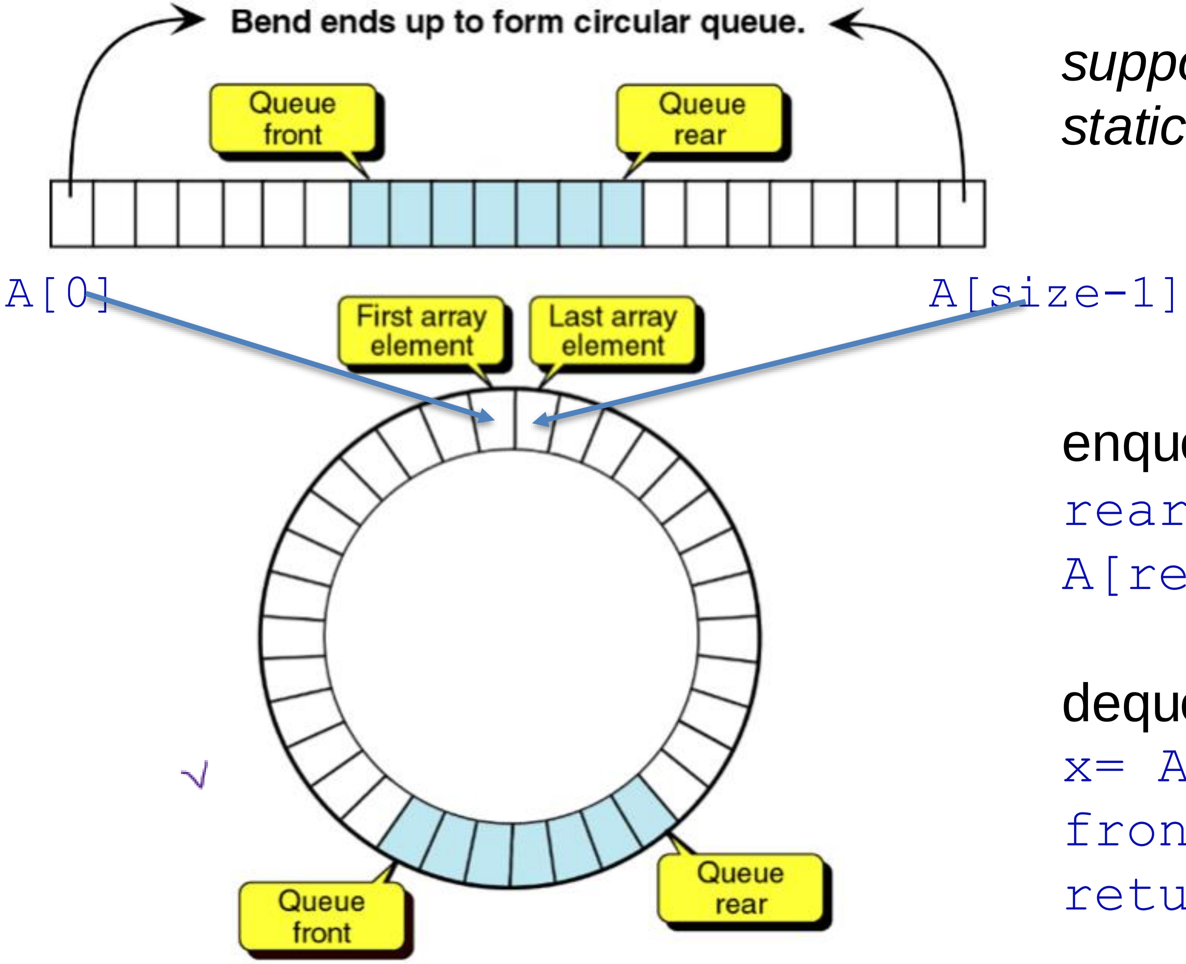
Q 4: Queues

Describe how to implement **`enqueue`** and **`dequeue`** using an unsorted array, and using a singly-linked list. Is it possible to perform each operation in constant time?

Using an array	**Using a linked list**

What if the array border is crossed?
What if the array is full?

Queue: using circular arrays



suppose using a big-enough static array

enqueue `x`:

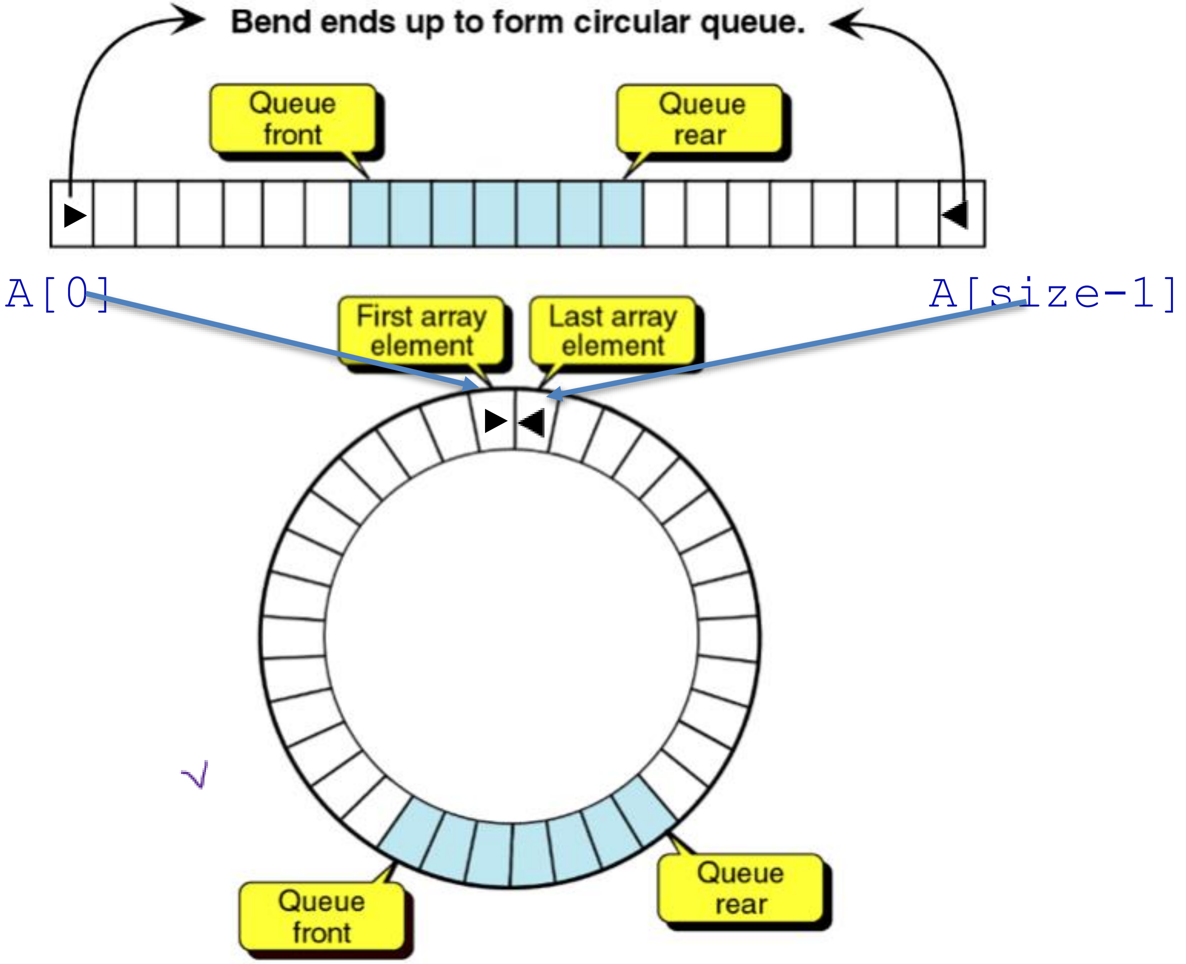
```
rear= rear+1;       ??
A[rear]= x;
```

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



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

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;
```

Q2.5 [homework]: Stacks & Queues

If you have access only to stacks and stack operations, can you faithfully implement a queue? How about the other way around?

using stacks to implement a queue

enqueue	dequeue

using queues to implement a stack

push	pop

5-minute break

stretch exercises

networking

Memory Pools: a C programs uses three memory pools during run-time

stack:

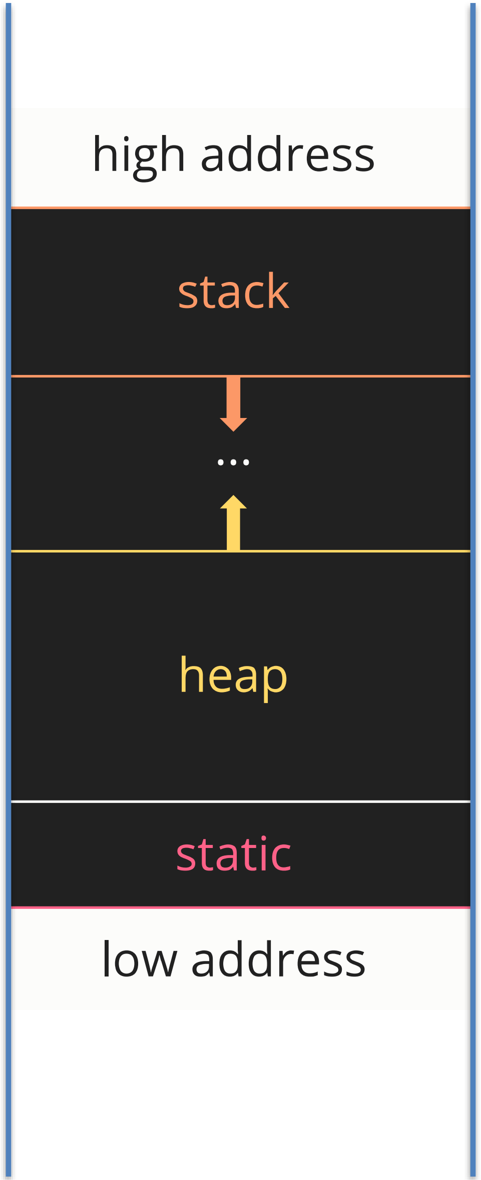
- where local variables live
- automatically allocated when a function starts
- automatically free-ed when the function ends
- has a limited size

heap:

- where dynamically-allocated memory lives
- allocated by programmers via *alloc() calls
- free-ed by programmers via free() calls
- virtually has unlimited size

static data segment:

- for global and static variables

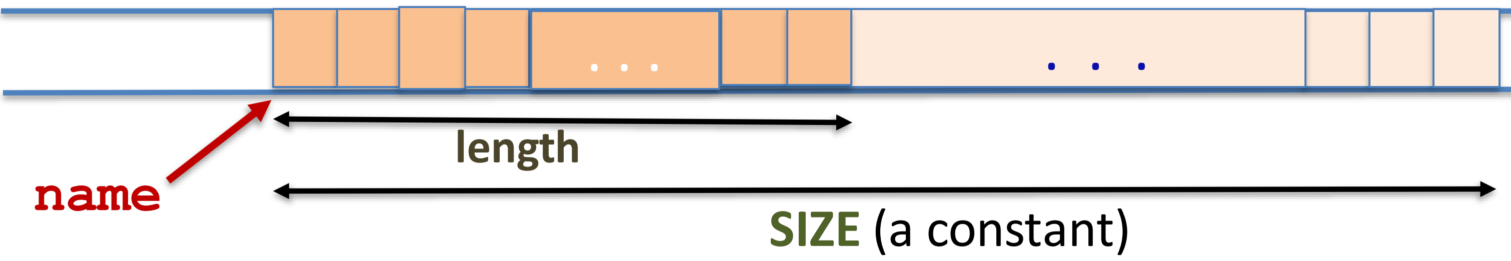


Static Arrays (for storing sequences of data)

A static array:

is a consecutive chunk of memory

has **name** (== pointer constant), **SIZE** (aka. capacity), **length** (or **n**, aka. **Used** - number of currently used elements)



examples:

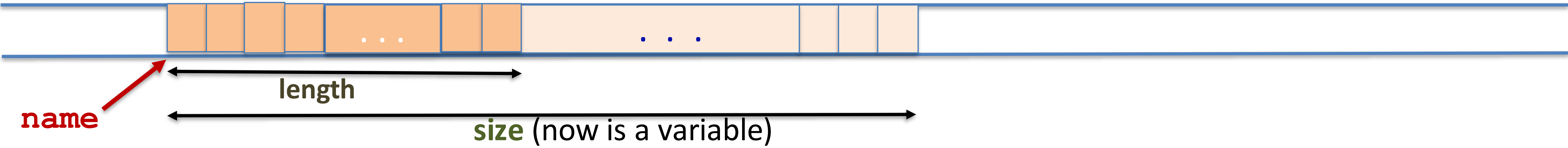
```
#define SIZE 1000
#define MAX 40
typedef struct {
   int id;
   char name[MAX + 1];
} student_t;
int A[SIZE];       // array of int
student_t B[SIZE]; // array of student_t
// code for setting value of n, and A[i], B[i] for i in range
0..n-1
```

```
void *P[SIZE]; // polymorphic array of SIZE elements

// use P as an array of pointers to student_t
for (i=0; i<n; i++)
  P[i]= B+i;  // same as P[i]= &B[i];

for (i=0; i<n; i++) {
  student_t *s= P[i];
  printf(%d %s\n”,  s->id, s->name);
  // note P[i]->id is invalid, why?
}
```

Review: Dynamic Arrays



<table>
<tr><th></th><th>Static Arrays</th><th>Dynamic Arrays</th></tr>
<tr><td>Example</td><td><pre>
#define SIZE 100
int i, n= 0, x;

int A[SIZE];

while (scanf(“%d”, &x)==1) {
  if (n==SIZE) {
    break;
  }

  A[n]= x;
  n++;
}
</pre></td><td><pre>
#define INIT_SIZE 4
int size=INIT_SIZE, i, n=0, x;
int *A;
A= malloc(size * sizeof(*A));
assert(A);

while (scanf(“%d”, &x)==1) {
  if (n==size) {
    size = size * 2;
    A= realloc(A, size*sizeof(*A));
    assert(A != NULL);
  }
  A[n]= x;
  n++;
}
// NOTE: better to pack {A, size, n} into a struct, and also use polymorphism
</pre></td></tr>
</table>

Peer Activity 1: Memory Pools

Consider the following code snippet:

```
...
int x = 2;
int y = 0;
int z = 5;
*(&y + 1) = 1;
...
```

Assuming that the C compilers allocate memory to local variables in order of declaration. What is most likely to happen when the code snippet is run?

A. *Error: Invalid syntax*
B. x == 1
C. y == 256
D. z == 1
E. Error: Segmentation fault

Peer Activity 2: Dynamic Arrays

What is the right ordering for these code snippets to implement a function

```
int ensure_array_size(struct array *arr)
```

that expands a struct array’s data space when it is full? Assume that there is a “return 0;” at the end of the function body.

A. 3–2–5–1–4

B. 3–2–5–4–1

C. 2–5–1–4-3

D. 2–5–4–1-3

```
/* Snippet 1 */
arr->data = res;
```

```
/* Snippet 2 */
arr->size *= 2;
```

```
/* Snippet 3 */
if (arr->used < arr->size) return 0;
```

```
/* Snippet 4 */
if (res == NULL) {
   arr->size /= 2;
   return 1;
}
```

```
/* Snippet 5 */
void *res =
  realloc(arr->data, arr->size*sizeof(void*));
```

Memory Management: Caveats

C grants programmers the **great power** of governing over memory.
That comes with a **great responsibility**.

Overstepping memory boundaries is a very real possibility with C.
Its consequences range from:

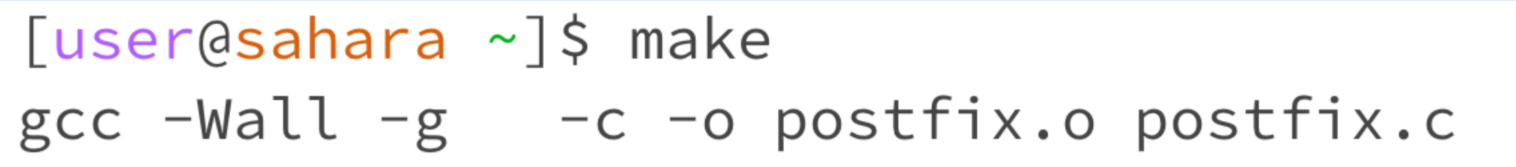
- **best:** getting immediate error (e.g. Segmentation fault) and crashing
- **worse:** overwriting memory ‘housekeeping’ data and crashing some time later
- **worst:** silently overwriting other variables and continuing execution

Lab Time: Use Ed for exercises

1. (Together with Anh) Complete W2.3 (reverse Polish notation) – understanding stacks/linked list and be familiar with multi-file programming
2. W2.1: Implement functions in `functions.c`, which reviews *function and function parameters*
3. W2.2*: dynamically resizing arrays* with `malloc`/`realloc` and `free`.
4. W2.4: sieve of Eratosthenes

Exercise W2.3

- Skim the requirement, inspect postfix.c
- Open Terminal and compile the project with



```
[user@sahara ~]$ make
gcc -Wall -g    -c -o postfix.o postfix.c
```

- Deal with compiler’s messages

Wrap Up

pseudocode

Big-O and efficiency

array and linked list as concrete data types

- representation in memory,
- how to implement search/insert/delete.

stack and queue as Abstract Data Types (ADT):

- operations,
- when to use,
- implementation using array and linked list.

Additional Pages

Data Types & ADT

A *concrete data type*, such as array or linked list, specifies a representation of data, and programmers can rely on that to implement operations (such as `insert`, `delete`).

An *abstract data type* specifies possible operations, but not representation. Examples: stacks, queues, dictionaries.

- When implementing an ADT, programmers use a concrete data type. For example, we might attempt to employ array to implement stack.
- When using an ADT, programmers just use its facilities and ignore the actual representation and the underlined concrete data type.

Example of using Stacks & Queues ?

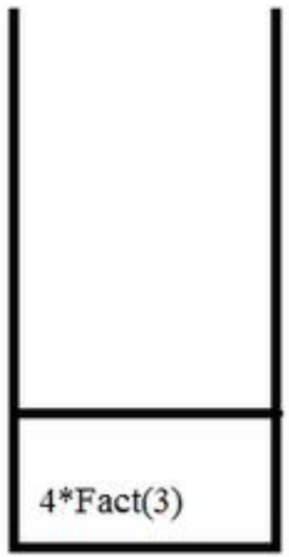
When function call happens previous variables gets stored in stack

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);
```



After the first call

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

When function call happens previous variables gets stored in stack

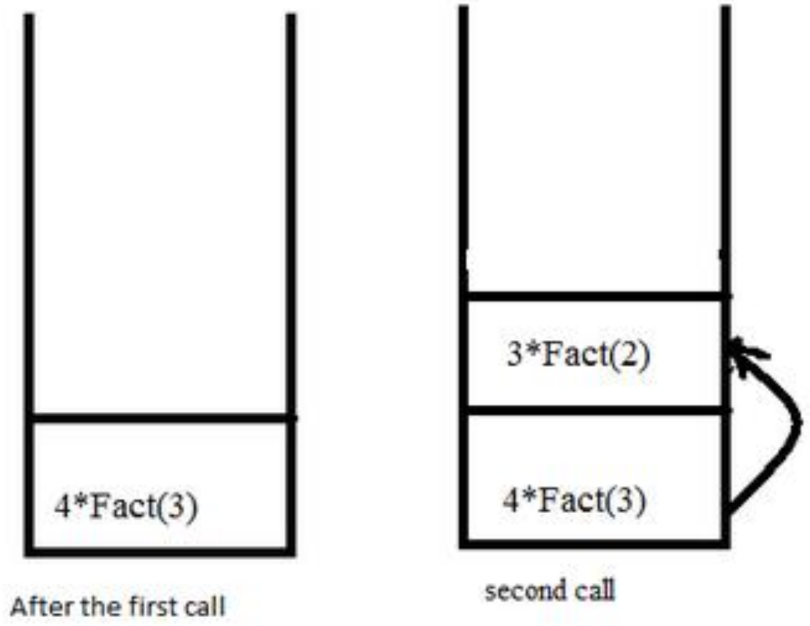
Stack is widely used in implementation of programming systems. For example, compilers employ stacks for keeping track of function calls and execution.

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



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

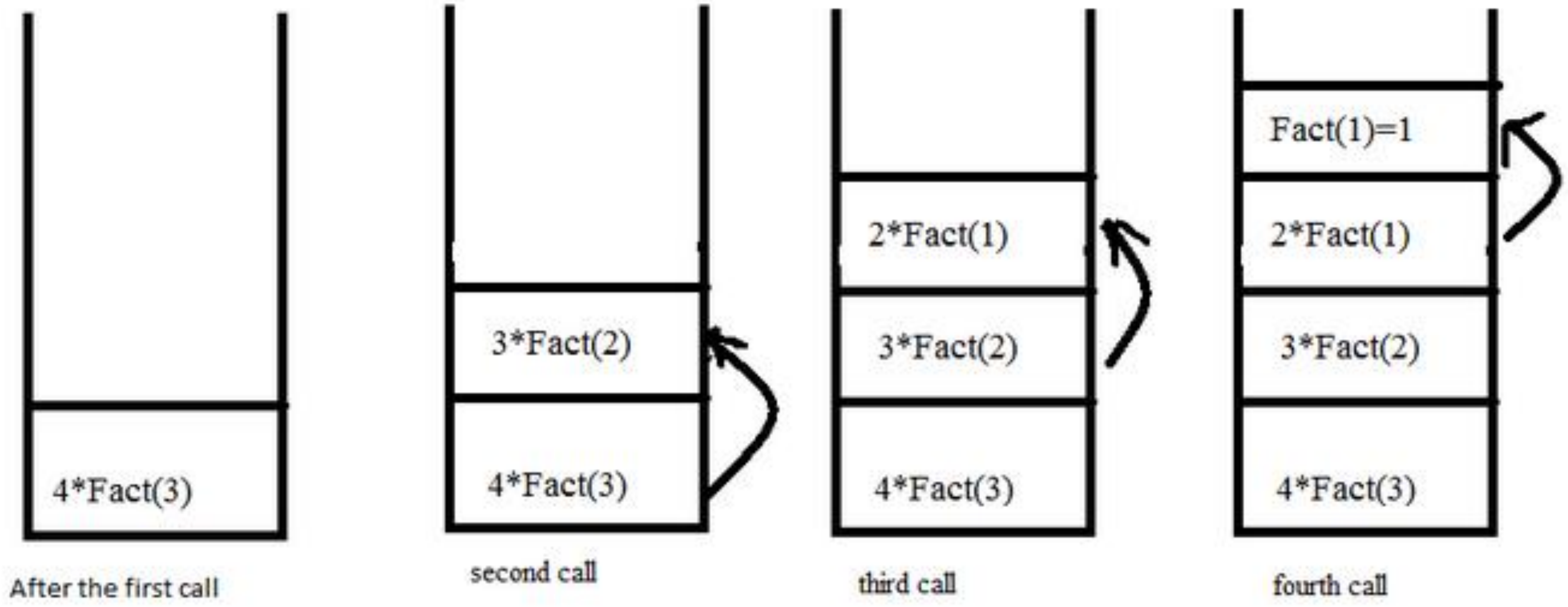
Stack for :

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

When function call happens previous variables gets stored in stack



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

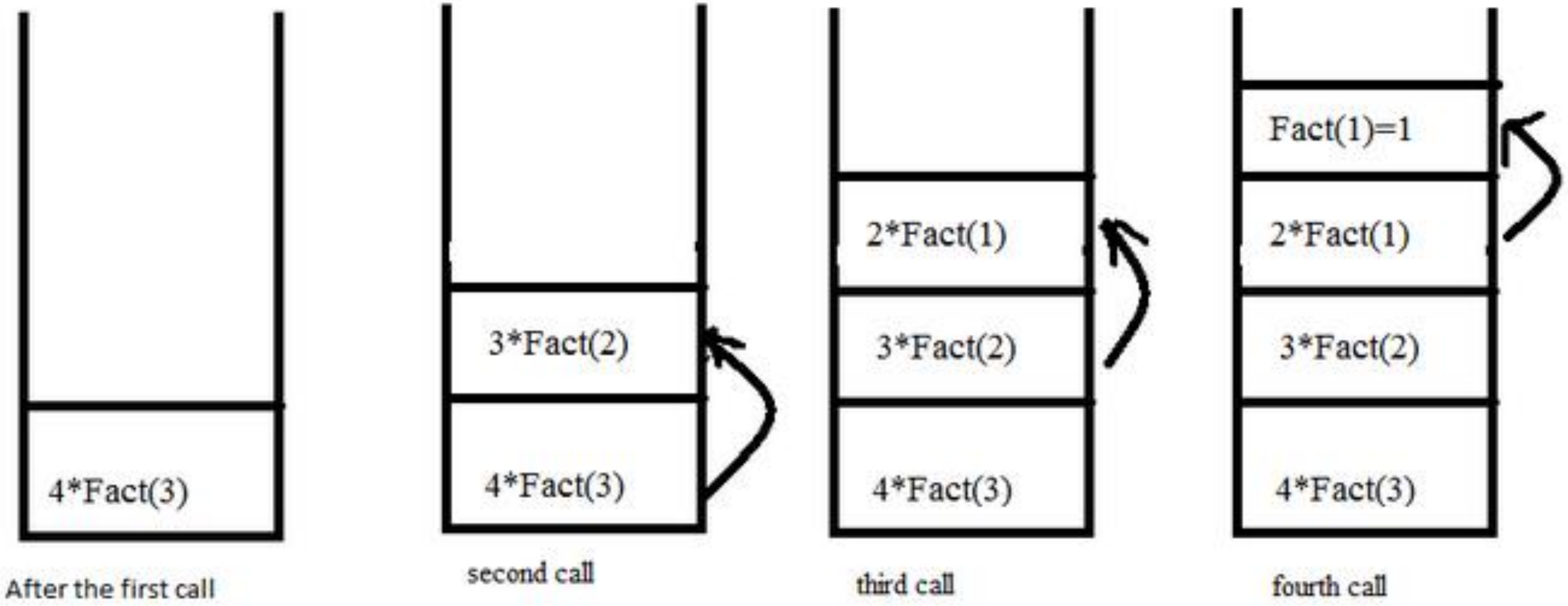
Stack for :

Fact(4)

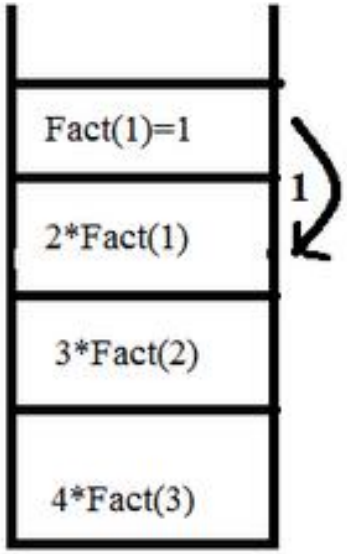
```
int Fact( int n ) {
 if ( n<=1 )
     return 1;
 return n*fact(n-1);
```

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

When function call happens previous variables gets stored in stack



Returning values from base case to caller function



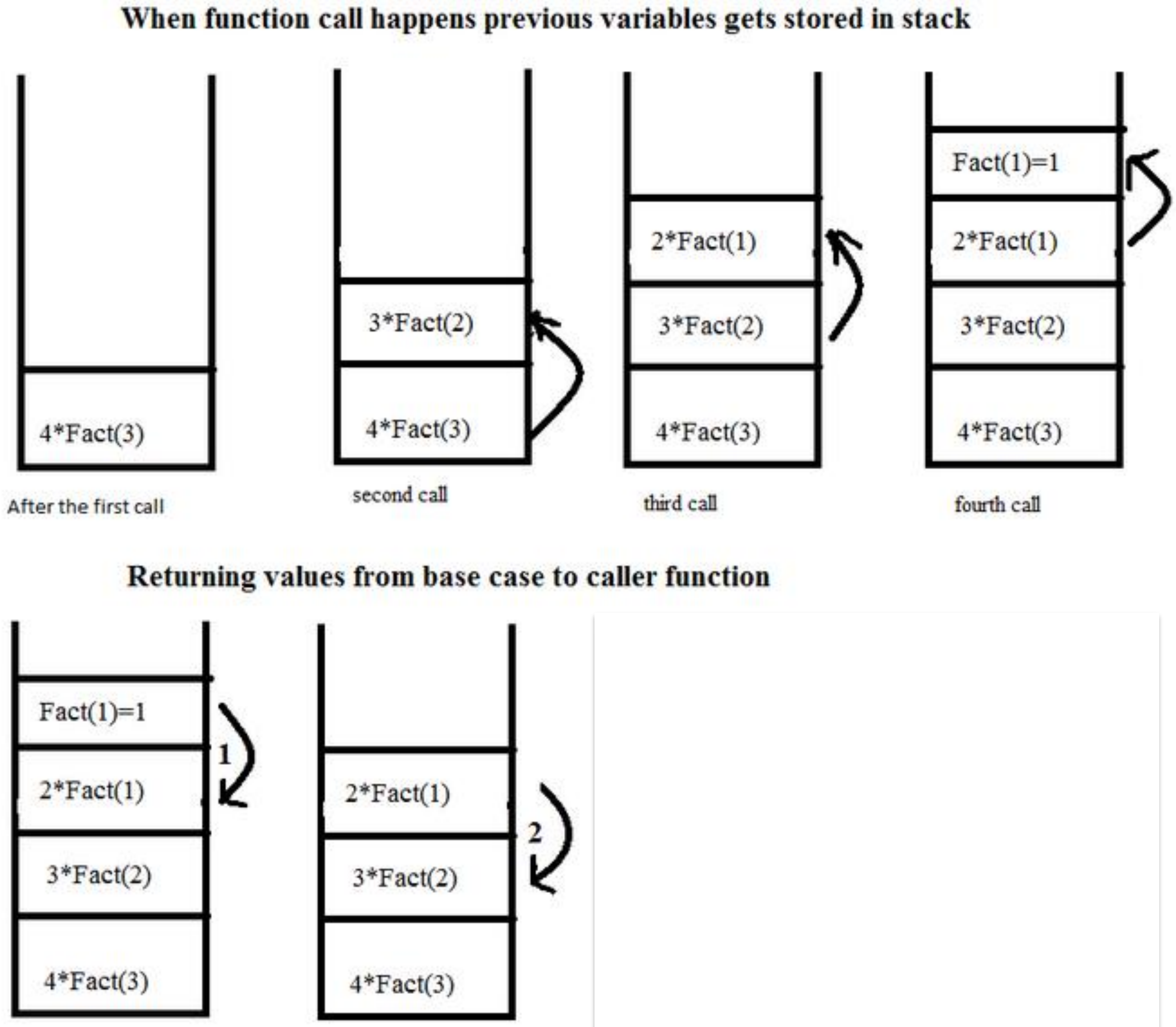
Stack is widely used in implementation of programming systems. For example, compilers employ stacks for keeping track of function calls and execution.

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



When function call happens previous variables gets stored in stack

Returning values from base case to caller function

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);
```

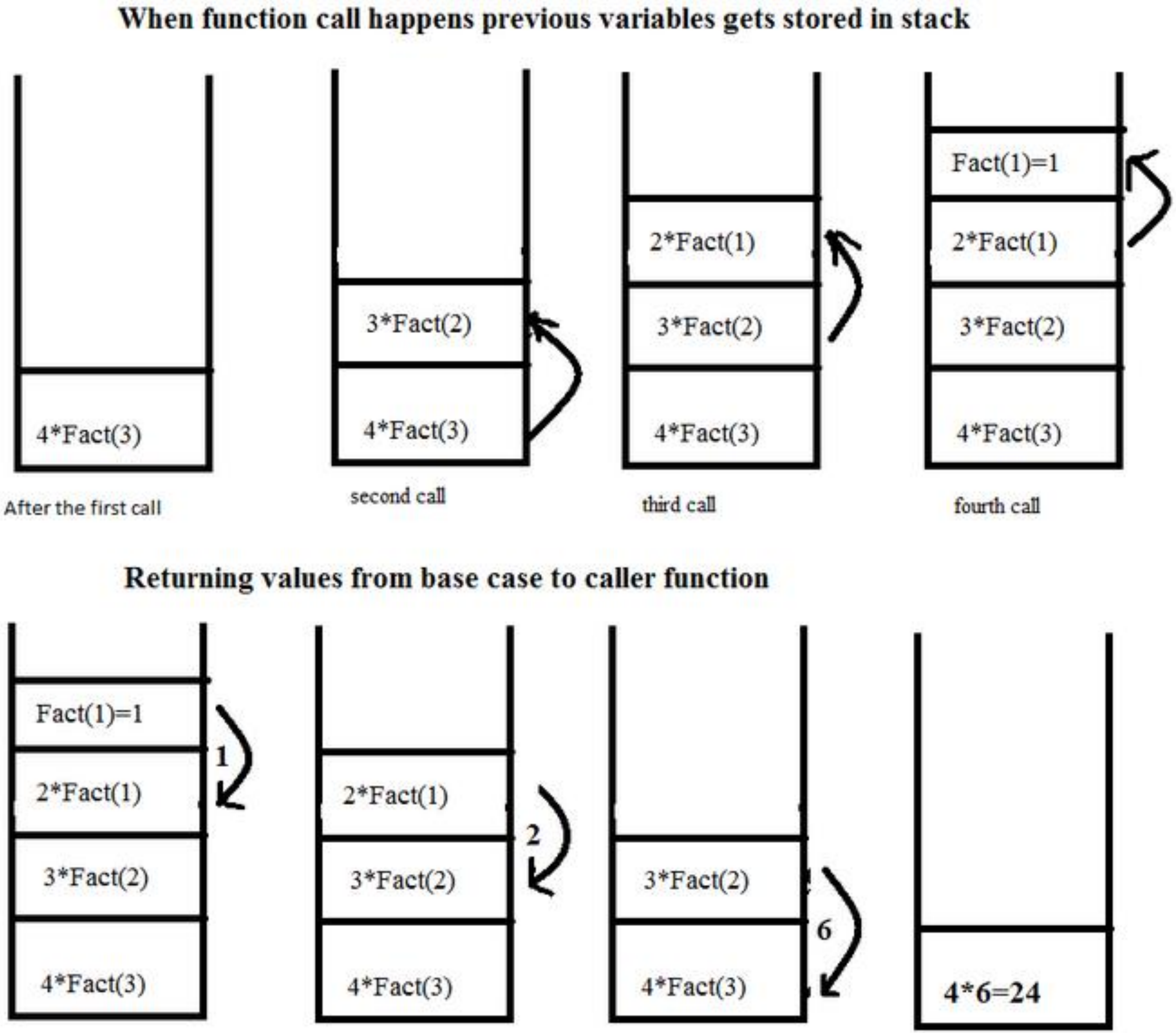


Image source: http://stackoverflow.com/questions/19865503/can-recursion-be-named-as-a-simple-function-call

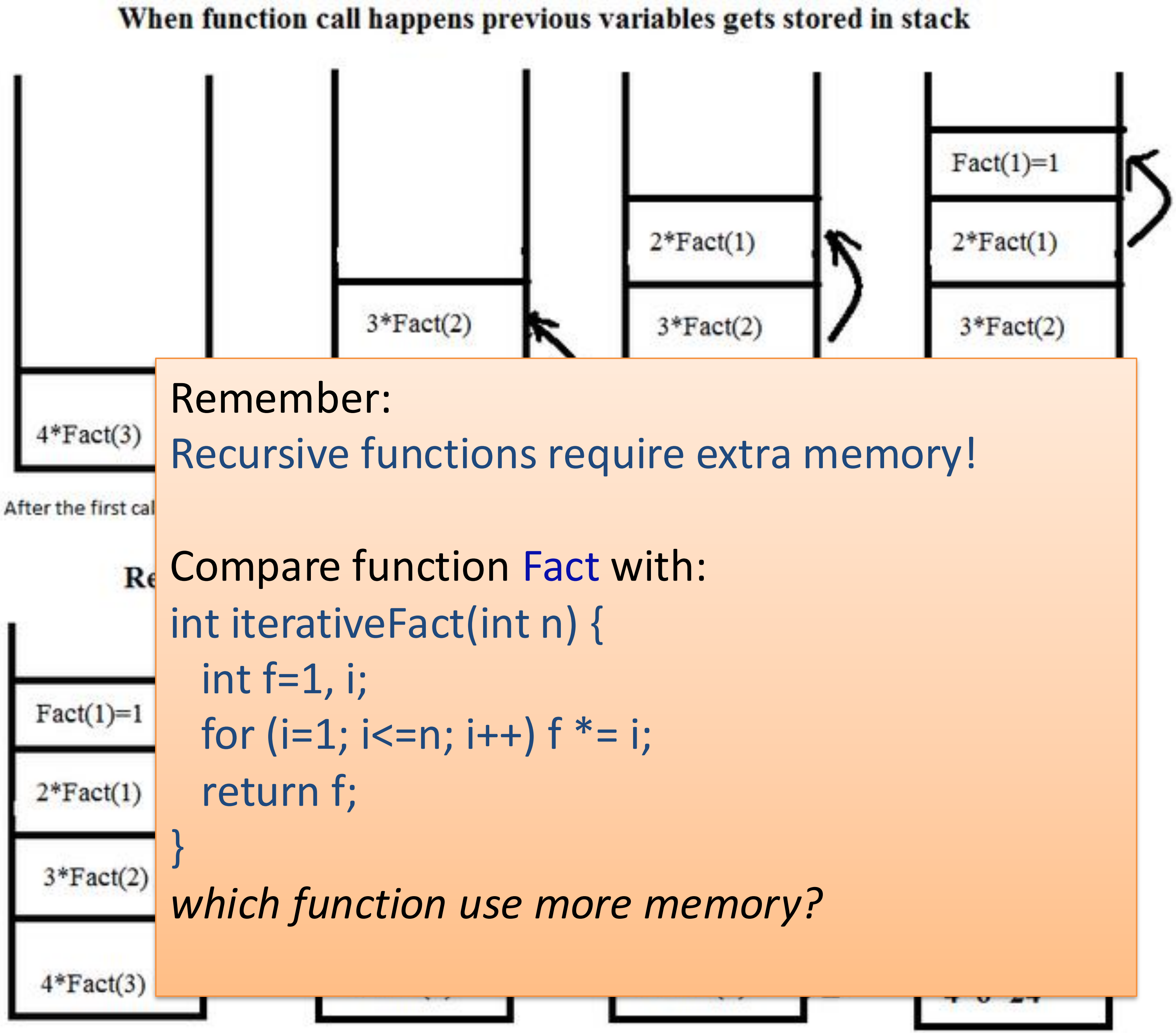
Stack is widely used in implementation of programming systems. For example, compilers employ stacks for keeping track of function calls and execution.

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



Remember:

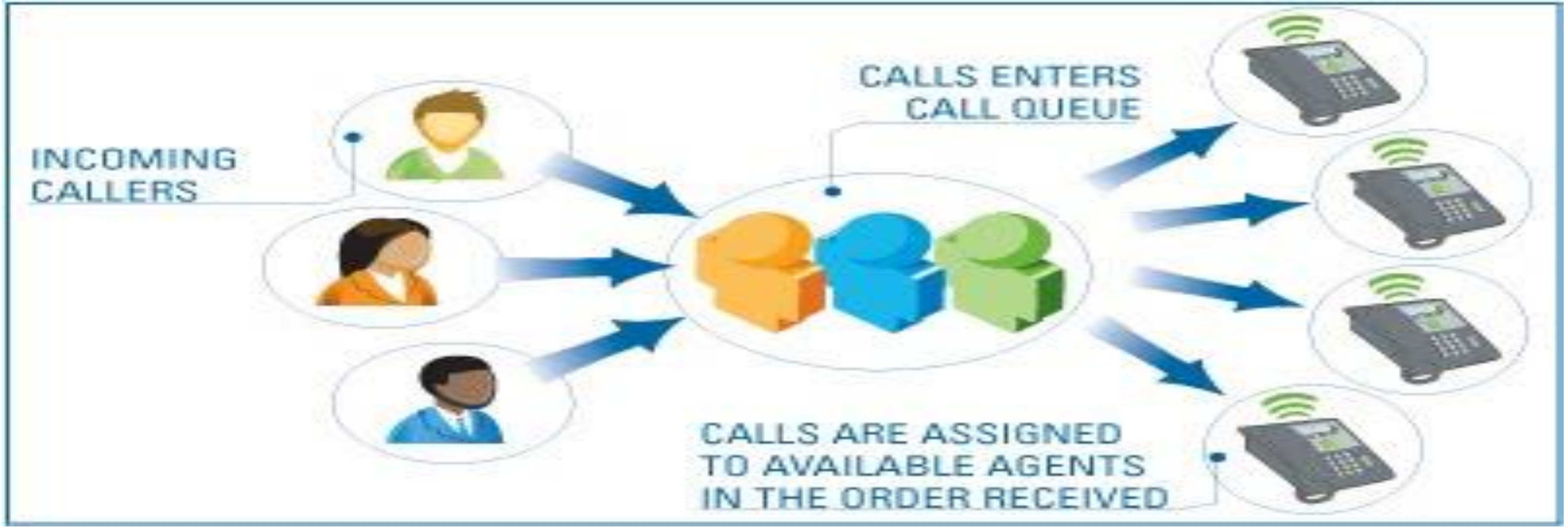
Recursive functions require extra memory!

Compare function Fact with:

```
int iterativeFact(int n) {
  int f=1, i;
  for (i=1; i<=n; i++) f *= i;
  return f;
}
```

which function use more memory?

ADT: Queue (FIFO)



INCOMING CALLERS
CALLS ENTERS CALL QUEUE
CALLS ARE ASSIGNED TO AVAILABLE AGENTS IN THE ORDER RECEIVED