COMP20007 Workshop Week 2

Welcome to a F2F Workshop!

1	About our workshops and comp20007
2	Arrays and Linked Lists (Tutorial Q1-Q2) ADT: Stacks, Queues (Q3-Q4)
3	5-min break for networking
4	LAB: • Programming Environment • C revision with some exercises

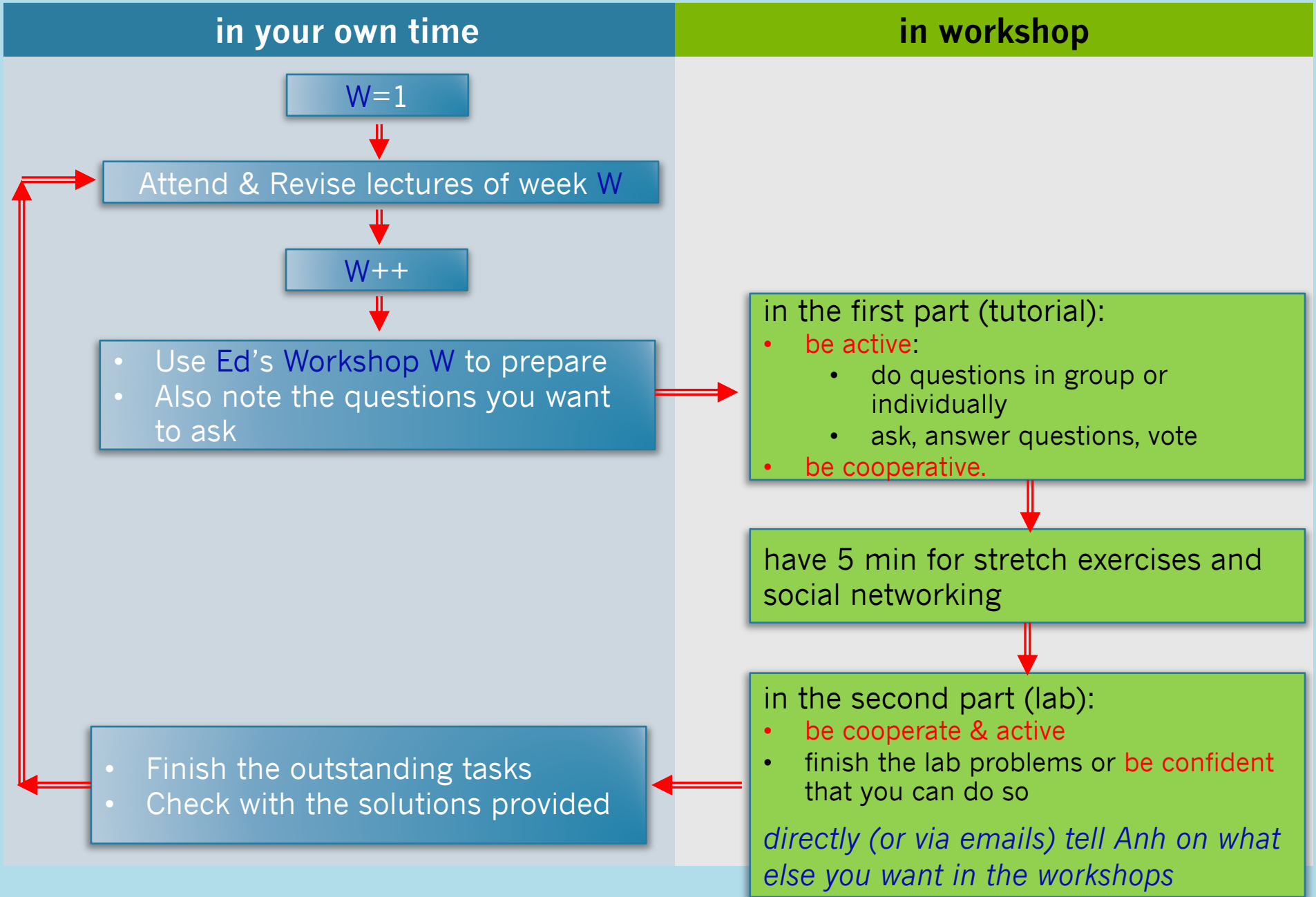
a friend of yours in comp20007

Anh Vo == Vo Ngoc Anh avo@unimelb.edu.au

if email:

- you should send from your uni's email account,
- subject better to start with "COMP20007" or just "C207"

suggested ToDos for Workshops (Learning-By-Doing)



suggested ToDos for Workshops (Learning-By-Doing)

the way we do tutorial questions in class

- Desirably using pens & papers
- Desirably working in group of 2-3 students:
  - groups can work in whiteboards
  - or at their desks
- Discussions, arguments are encouraged

The question is reviewed at the end.

for the lab:

- had a overall look at all problems when preparing at home
- make sure that you know how to do all problems if having enough time
- implement as many as possible, but focus on 1-2 most valuable problems

in workshop

in the first part (tutorial):

- be active:
  - do questions in group or individually
  - ask, answer questions, vote
- be cooperative.

have 5 min for stretch exercises and social networking

in the second part (lab):

- be cooperate & active
- finish the lab problems or be confident that you can do so

directly (or via emails) tell Anh on what else you want in the workshops

Problem → **Algorithm/Pseudocode** **Complexity**

comp20007

Focus: designing algorithms.

Use efficiency performance to evaluate designs:

- mainly, time complexity, and
- sometimes, space complexity.

```
    return p
  p := p.next
return null
```

Problem → **Algorithm/Pseudocode** **Complexity**

Searching for a specified element (amongst a series of elements)

```
j := 0
while j < last
  if A[j] == x
    return j
  j := j+1
return null
```

O(?)

```
p := head
while p != null
  if p.val == x
    return p
  p := p.next
return null
```

O(?)

```
function find(A,x,lo,hi)
  if lo > hi
    return null
  else if A[lo] == x
    return lo
  else
    return find(A,x,lo+1,hi)
```

O(?)

Problem → **Pseudocode** → C code ?

Searching for a specified element (amongst a number of elements)

```
p := head
while p != null
  if p.val == x
    return p
  p := p.next
return null
```

```
j := 0
while j < last
  if A[j] == x
    return j
  j := j+1
return null
```

```
function find(A,x,lo,hi)
  if lo > hi
    return null
  else if A[lo] == x
    return lo
  else
    return find(A,x,lo+1,hi)
```

???

???

???

Problem → **Pseudocode** → C code

Searching for a specified element (amongst a number of elements)

```
p := head
while p != null
  if p.val == x
    return p
  p := p.next
return null
```

```
node_t *search(int x, list_t *l) {
  node_t *p= l->head;
  while (p) {
    if (p->val==x) return p;
    p= p->next;
  }
  return NULL;
}
```

```
j := 0
while j < last
  if A[j] == x
    return j
  j := j+1
return null
```

???

```
function find(A,x,lo,hi)
  if lo > hi
    return null
  else if A[lo] == x
    return lo
  else
    return find(A,x,lo+1,hi)
```

???

Our focus is building algorithms and analysing efficiency.

Pseudocode and big-O are the main tools. But we also do some C implementation.

Q2.1: Arrays & Linked Lists [15m]

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(n)** • do a linear search	**O( )** •
Inserting a new element	**O( )** •	**O( )** •
Deleting the final element	**O( )** •	**O( )** •
Deleting a specified element	**O( )** •	**O( )** •

Q2.1: Check your answers

- 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(n)** • do a linear search	**O(log n)** • do a binary search
Inserting a new element	**O(1)** • put the new element at the end of the array	**O(n)** • find the right place for the new element with linear or binary search: O(logn) • shift right this and the RHS by 1 position: O(n) • put the new element in this free space: O(1).
Deleting the final element	**O(1)** • just remove it from the end of the array	**O(1)** • just remove it from the end of the array
Deleting a specified element	**O(n)** • find the element with linear search: O(n) • replace it with the last element of the array	**O(n)** • find the element with binary or linear search: O(logn) • shift left the RHS by 1 position: O(n)

Q2.2: Linked Lists [10 m]

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( )	
Inserting a node at the end		
Deleting the first node (at the start)		
Deleting last node (at the end)		

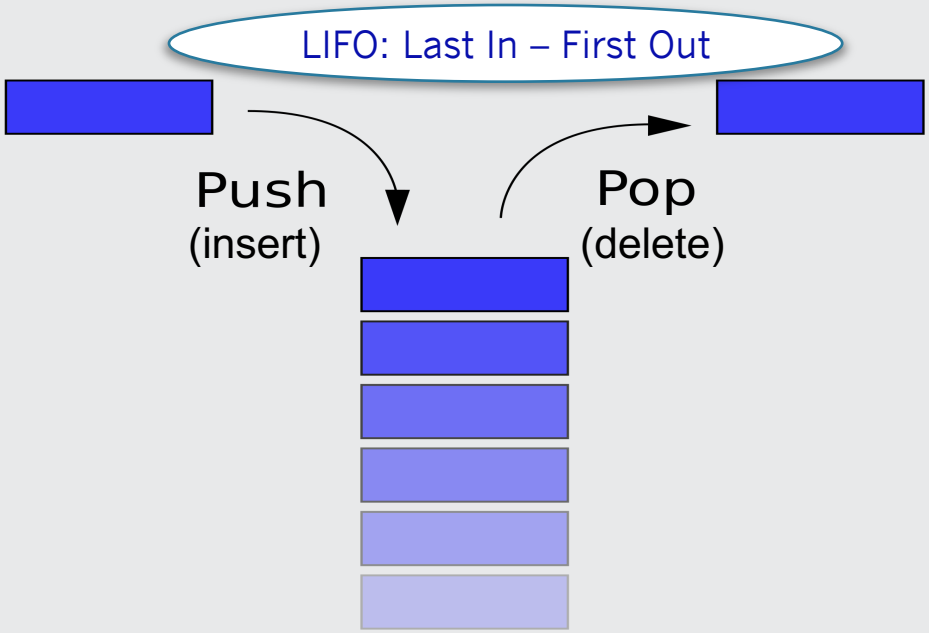
Q2.2: Check your answers

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. **Note: also assume that the links to the first and last element of a list is called `head` and `foot` respectively.**

<table>
<tr><th>Operation</th><th>Singly</th><th>Doubly</th></tr>
<tr><td>Inserting a node at the start</td><td rowspan="3">O(1)
ensure that the link in the inserted/removed node, the link in the next or previous node are updated
ensure that the links <code>head</code> and <code>foot</code> of the list are updated accordingly; note that when inserting to an empty list, and when deleting from a list that has a single element, both <code>head</code> and <code>foot</code> need to be updated</td><td rowspan="4">O(1)
as in the first row of LHS, but note that each node has 2 links, and both links of the inserted/removed node need to be updated.
note that all operations, including deleting the last node, is O(1).</td></tr>
<tr><td>Inserting a node at the end</td></tr>
<tr><td>Deleting the first node (at the start)</td></tr>
<tr><td>Deleting the last node (at the end)</td><td>O(n)
just like the above, but we need to identify the second last element in order to disconnect the last element and to update the link <code>foot</code>. To do that, we need to the follow the list all the way from the start to the second last element, and that causes O(n).</td></tr>
</table>

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

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



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

Stack Operations

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

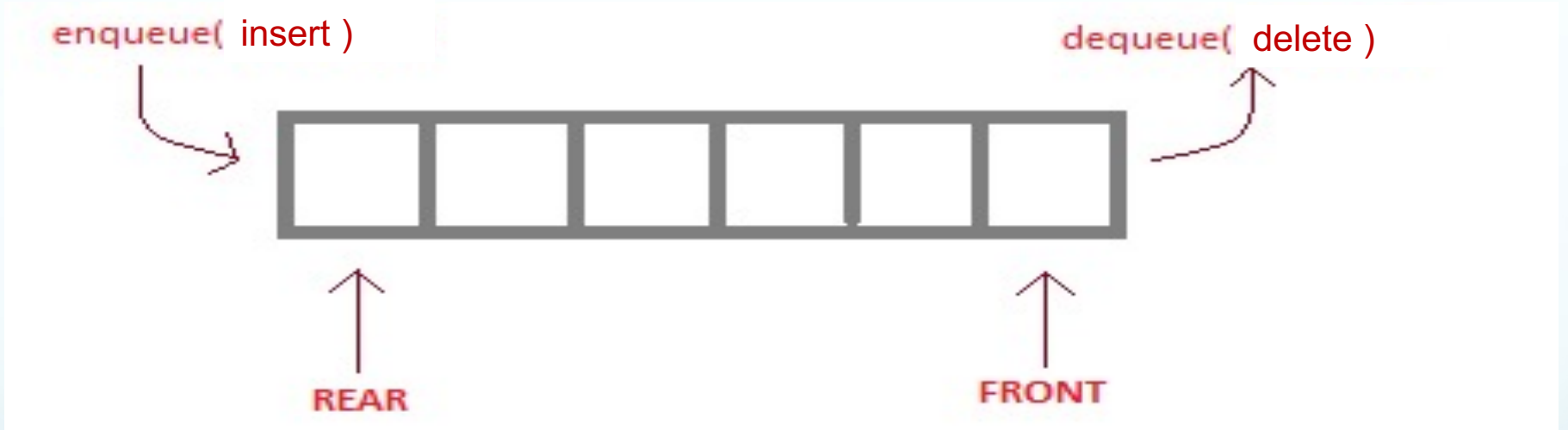
`pop()` : remove and return an element from (the top of) stack

`isEmpty()`: check if stack is empty

`create()` : create a new, empty stack

`delete()` : delete (free all associated memory)

Another ADT: Queue (FIFO)



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, or **`delete()`:** delete a queue (free all associated memory)

Q2.3: Stacks [8 m]

- Describe how to implement `push` and `pop` using an unsorted array, and using a singly-linked list.

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

Q2.3: Check your answers

- Describe how to implement `push` and `pop` using an unsorted array, and using a singly-linked list.

Using an (unsorted) array	Using a (singly-)linked list
`push`(x) is done by adding x to the end of the array → O(1) `pop`() is done by removing the element at the end of the array → O(1)	`push`(x) is done by adding x to the start of list → O(1) `pop`() is done by removing the element at the start of the list → O(1) Note: it's not good to add/remove at the end of the list, because in this cases `pop`() will have O(n) complexity

Q2.4: Queues [12 m]

- 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

Q2.4: Check your answers

- 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
we need 2 indices: `front` which points to the leftmost element, and `rear` that points to the rightmost element of the array `enqueue`(x) by adding x to the end of the array, and updating `rear` → O(1) `dequeue` by removing the element at the start of the array and updating `front` → O(1). A problem is the growing of `front` and `rear`, which requires a large array with wasted space at the start. To avoid that we can use a circular array where the first element of the array is considered as immediately following its last element. Using a circular array of size S is simple: • when `dequeue`: instead of `front:= front+1` now we use `front:= (front+1) % S` • when `enqueue`: instead of `rear:= rear+1` now we use `rear:= (rear+1) % S`	In this case, the pointer `head` and `foot` of the linked list will serve as the queue's `front` and `rear` respectively. `enqueue`(x) by adding x to the end of the list (and update `foot`) → O(1) `dequeue` by removing the element at the start of the list (and update `head`) → O(1) Note: adding at the start and removing from the end are not suitable, because removing from the end takes linear time in a singly-linked list.

Q2.5 [optional]: Stacks & Queues

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

~~You may assume that your stacks and queues also come with a size operation, which returns the number of elements currently stored.~~

Your answer: using stacks to implement a queue

enqueue	dequeue

Your answer: using queues to implement a stack

push	pop

5-minute break

- stretch exercises
- networking

Just for fun (perhaps at home)

google ”algorithm for making friends”
and watch ”The Friendship Algorithm”
(a 2.5-minute videos).

Lab Time: Use Ed for exercises and assignments

1. *Start* with `helloworld.c` [DoItTogether with Anh]
2. (Together) Implement functions in `functions.c`, which reviews *function and function parameters*
3. *dynamically resizing arrays* with `malloc`/`calloc` and `free`. Forgot `malloc`? Try command "`man malloc`" in your terminal.

Why `Ed`?

- Strong: powerful editor, shell, compilers, `valgrind`, `gdb`, …
- Safe : codes and files will never be lost
- Sound : codes/files can be accessed from any devices
- Sane : your assignments will be tested on `Ed`

4. *Optional*: download Alistair's `listops.c` (google it!), then add a *least-effort* implementation of stack with:
   - data type `mystack_t` ,
   - functions `createStack`, `freeStack`, `push`, `pop`.

Do, or ask Anh how to, **implement stacks in just 5 minutes**!

Wrap Up

- array and linked list as concrete data types.
- stack and queue:
  - as and Abstract Data Type (ADT),
  - operations,
  - implementation using array and linked list.
- C revision, especially:
  - functions, pointers, `malloc`/`realloc`, `free;`
  - dynamically resizing arrays (or just *dynamic arrays*).

- Technical stuffs:
  - Use Ed for programming exercises & assignments!
  - Self-Learn to use `Ed`'s `gcc` and debugging tools `gdb`, `valgrind` at home.

Have Fun with comp20007 and avo

Additional Pages

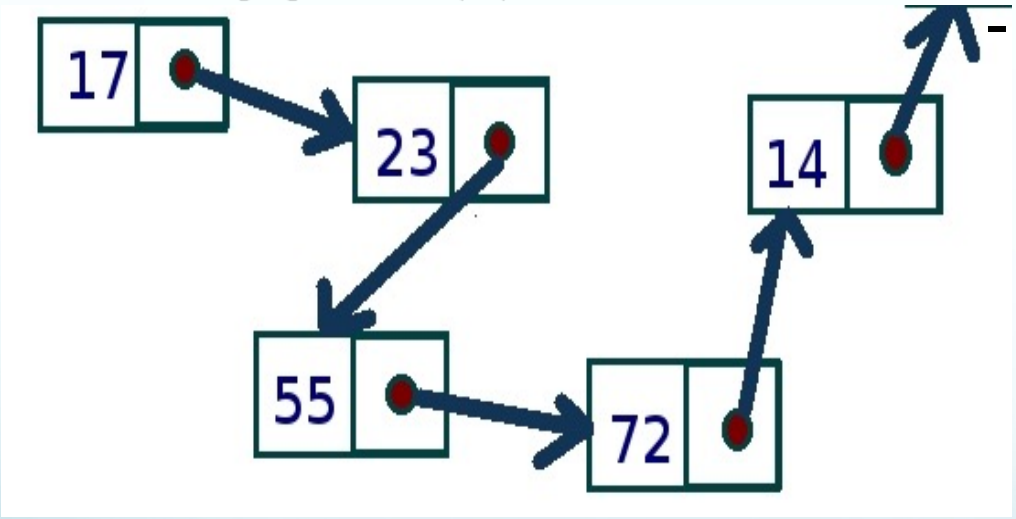
Data Types & ADT

- 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.

Two concrete data types: Arrays & Linked Lists

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

Access L[k] in O(n) time!



In C:

- How to specify an array? How to traverse it?
- How to specify a linked list? How to traverse it?

Example of using Stacks ?

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);
```

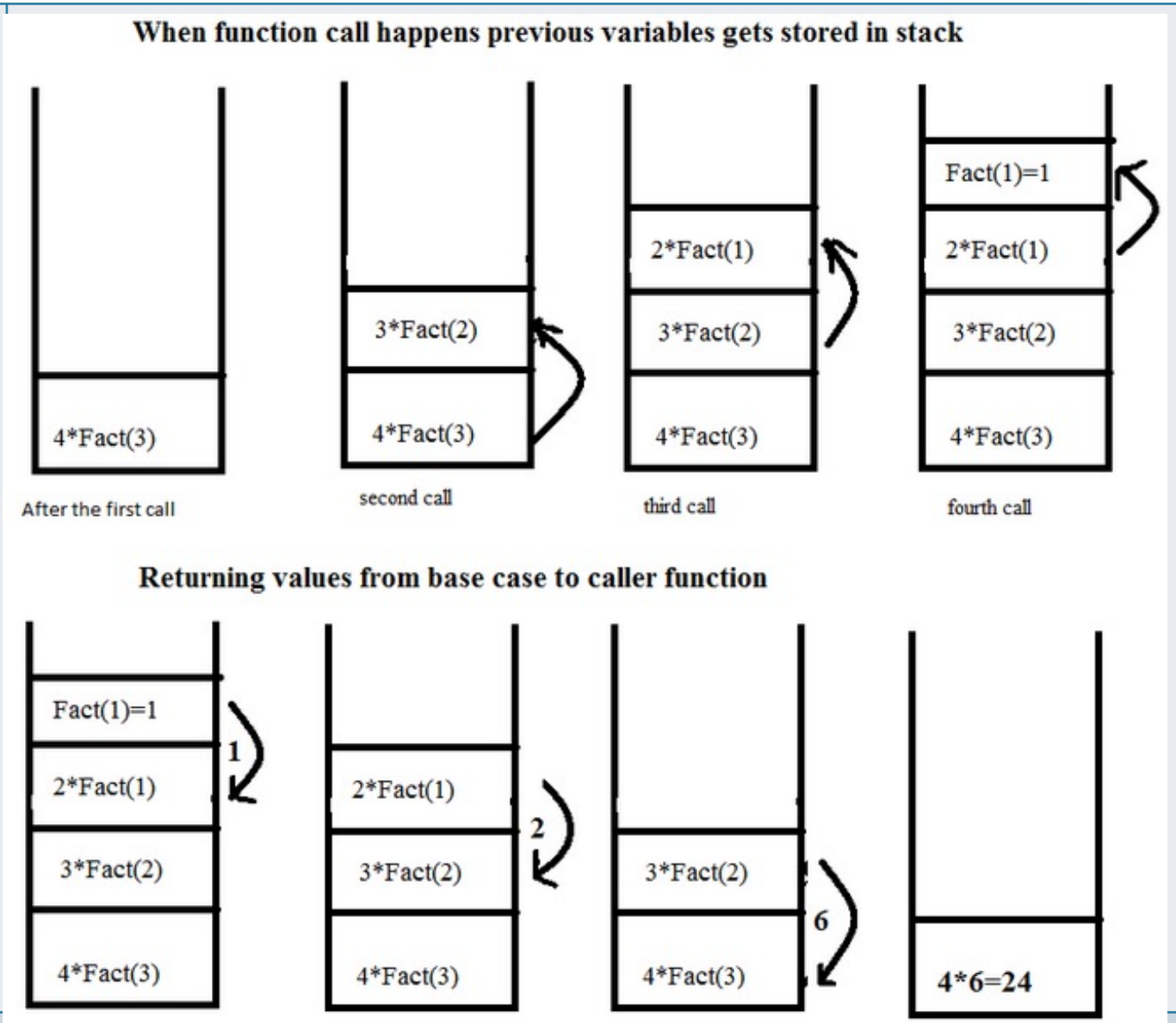


Image source: http://stackoverflow.com/questions/19865503/can-recursion-be-named-as-a-simple-function-call

ADT: Queue (FIFO)