

COMP1511 Week 10

adt, stacks and recursion

Joanna Lin

What we'll cover today

- Abstract data types (ADT)
 - Hiding away complex logic.
- Stacks
 - The *concept* of a stack.
 - Representing stacks in code.
- Recursion
 - The base case.
 - Recursive logic.



Abstract Data Types

- ADTs are defined by purely by their behaviour and the operations that can be done on them.
- They are 'abstract' because the details of how these operations are implemented are hidden from the user. The only part of the ADT that is exposed to the user are the necessary operations.
- We can compare an ADT to a car:

Operation	Behaviour
Accelerate	The car moves faster
Steer	The car turns
Brake	The car stops

- A manufacturer builds (implements) a car based on the defined operations, careful to expose only the necessary parts (the accelerator, steering wheel and brake) to the user to perform these operations, hiding (abstracting) away all of the complicated engineering behind a car.

Benefits of ADTs

- Until now, we've implemented everything with full control over every piece of data in our code. Think back to how, when we worked with linked lists, we had direct access to the data field and the next field simply by dereferencing our struct node pointers.
- However, having to keep track of every piece of data is not practical or efficient when you want to work on larger projects that require use of several data structures with complicated implementations. Remember, data types are a means to an end — not the end!
- Having full control over every bit of information makes it easy for a data type to become inconsistent leading to undefined behaviour.
- Imagine if, to operate a car, you needed to manually set the engine rotation speed and man the combustion chambers. It wouldn't even be practical to use a car!
- Furthermore, allowing so much access to the inner workings of the car means one is much more likely to break it by accident. For example, if you operate the engine incorrectly, pressing the accelerator might not work anymore!

Stacks

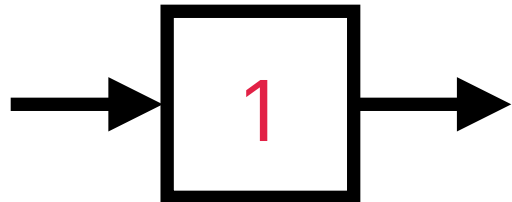
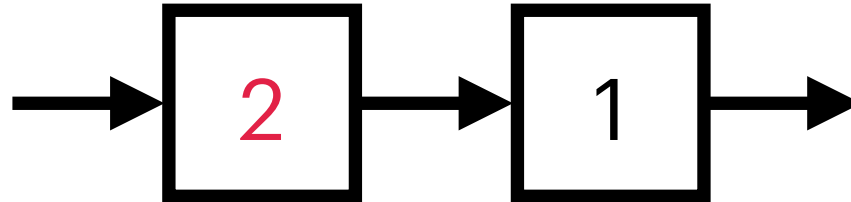
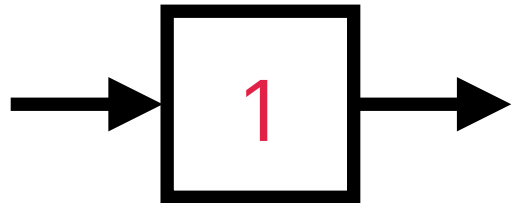

An ADT

- Consider a stack of books: the last book you placed onto the stack has to be the first book you take off of it.
- A stack in programming also has this last in, first out (LIFO) behaviour.
- Some operations (**bold** ones are the most pertinent to stacks)

Operation	Behaviour
Creating a stack	Gives the user the stack data structure
Pushing	An element is added to the stack.
Popping	The last element that was added in is taken out of the stack. The user is informed of the value that was removed.
Check if empty	Tells the user whether the list is empty

Stack (Linked List Version)

- **Implementation:** When the user pushes a number, we insert a node at the start of the list. When the user pops, we store the value inside the node the start of the list, then remove that node.
- Example usage:

What the user does	Our implementation	What the user sees
Push the number 1 into stack		—
Push the number 2 into stack		—
Pop from the stack		The number 2
Pop from the stack		The number 1

Stack (Array Version)

- **Implementation:** We keep track of the size of the stack, starting from 0. When the user pushes a number, we add a value at index given by the size of the stack and increment the size counter. When the user pops, we return the element at index size - 1, and then decrement the size of the list.
- Example usage (notice how the ‘what the user sees’ section is exactly the same as the linked list version):

What the user does	Our implementation	What the user sees
Push the number 1 into stack	<div><div><div>1</div><div>0</div><div>0</div><div>...</div></div><div>size: 1</div></div>	—
Push the number 2 into stack	<div><div><div>1</div><div>2</div><div>0</div><div>...</div></div><div>size: 2</div></div>	—
Pop from the stack	<div><div><div>1</div><div>2</div><div>0</div><div>...</div></div><div>size: 1</div></div>	The number 2
Pop from the stack	<div><div><div>1</div><div>2</div><div>0</div><div>...</div></div><div>size: 0</div></div>	The number 1

Factorials

Recursive logic in practice.

- We can define $n!$ as
 - $0! = 1$
 - $n! = n \times (n - 1)!$ when $n \geq 1$.
- This is a recursive definition: to calculate a factorial, we need to know the factorial of the number one less, which needs to know the factorial of the number one less than that etc. until we hit $0!$ — the ‘base case’ — one in which we define the answer.
- All recursive logic follows this principle: we must write a base case and a recursive case.

Iterative Solution

Using a While Loop

To calculate a factorial, we'd think of using this approach:

```
// A function that calculates the factorial of an integer
int factorial(int n) {
    int ans = 1;
    int i = 1;
    while (i <= n) {
        ans *= i;
        i++;
    }
    return ans;
}
```

Recursive Solution

Using a Recursive Function

Consider the following function

```
// A function that calculates the factorial of an integer
int factorial(int n) {
    assert(n >= 0); (error checking)
    if (n == 0) {
        return 1; (base case)
    } else {
        return n * factorial(n - 1); (general case)
    }
    notice how the function is calling itself!
}
```

Memory Model

What happens in memory when we write this function call (using recursive solution)?

```
int fac = factorial(3);
```

