TEMASEK JUNIOR COLLEGE

**Temasek Junior College**

**2022 JC1 H2 Computing**

**Data Structures 2 – Stacks**

## §2.1 Introduction to Stacks

A **stack** is an abstract data type that holds an ordered, linear sequence of data elements. It is a **last-in-first-out (LIFO)** data structure, where the last data element added to the data structure is the first data element to be removed.

***Analogy***
A stack in a computer works in the same way as a stack of books waiting to be marked or
a stack of dishes waiting to be washed up – whichever item was added to the top of the stack last will be the first one to be dealt with.

## §2.2 Stack Pointers

Unlike physically removing a weight from a stack of weights, a data element in a stack data structure is not actually removed. What happens is that a **top pointer** (or stack pointer) keeps track of where the top of the stack is.

To implement a stack, you need to maintain a **top pointer** that points to the **top** of the stack (the last element being added). In addition, there needs to be a **base pointer** that always points to the **base** of the stack.

When the stack contains at least one data element, the base pointer will always point to the first element being added. If the top pointer and the base pointer both point to the same data element, there is only one data element in the stack.

## §2.3 Stack Operations

The main stack operations are given in the table below.

| Keyword | Operation |
|---|---|
| `PUSH(<DATA>)` | Adds a data element to the top of the stack. |
| `POP()` | Removes the data element currently on the top of the stack. |
| `PEEK()` | Returns a copy of the element on the top of the stack without removing it. |
| `IS_EMPTY()` | Checks whether a stack is empty. |
| `IS_FULL()` | Checks whether a stack is at maximum capacity when implemented as a static (fixed-size) structure. |

- **Pushing**
  - ✓ The process of adding a new data element to the top of the stack.
  - ✓ When a data element is pushed onto the top of the stack, the top pointer moves up to point to that data element.

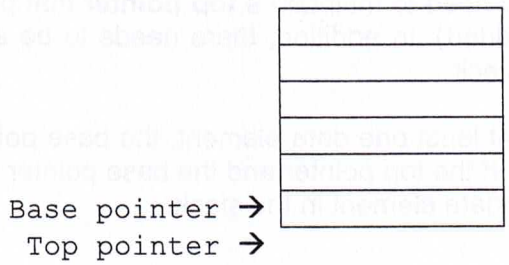
- **Popping**
  - ✓ The process of removing a data element from the top of the stack.
  - ✓ When a data element is popped off the top of the stack, the top pointer moves down to the element that will now be on the top of the stack.
  - ✓ A copy of the popped data element remains on the stack. Depending on how the stack is implemented, this copy of the data can be deleted, replaced with a `NULL` value, or left to be overwritten in the future.

- **Peeking**
  - ✓ The process of identifying the data element at the top of the stack without popping it.

**Example 1**

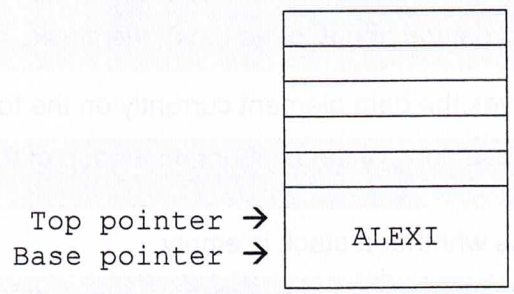
- This is a simplified example of a stack in use.
- The stack in this example can only store six data elements.
- The top pointer is used to show where the top of the stack is.
- The base pointer always points to the base of the stack.

- The stack is currently empty.



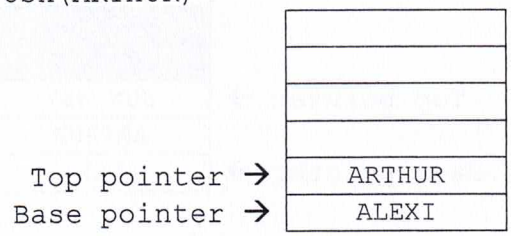

- The first data element `ALEXI` is pushed onto the top of the stack.
- The top pointer and the base pointer are now pointing at the same data element. This happens when there is only one data element in the stack.

```
PUSH(ALEXI)
```

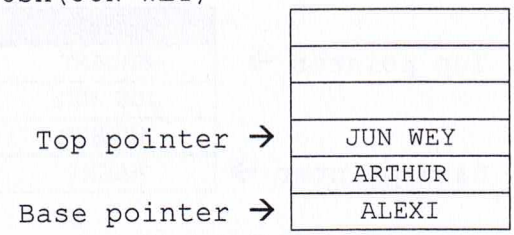


- Continue to push four more data elements, ARTHUR, JUN WEY, BRYANT, and CONROY, one after another onto the top of the stack.
- The top pointer will always move up to point to the top of the stack.
- The base pointer remains pointing at the base of the stack, now containing the first data element.

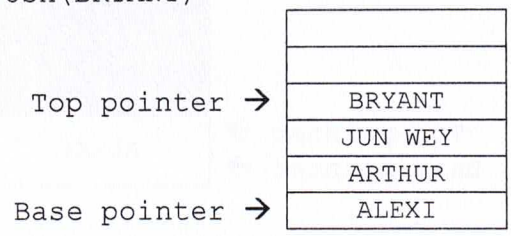
PUSH(ARTHUR)



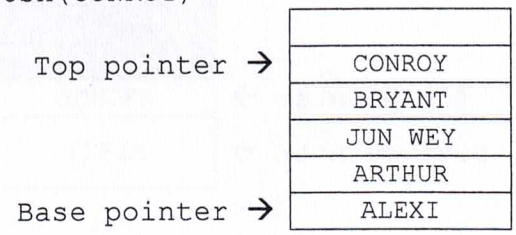

PUSH(JUN WEY)




PUSH(BRYANT)

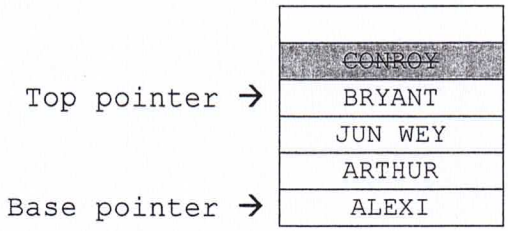



PUSH(CONROY)



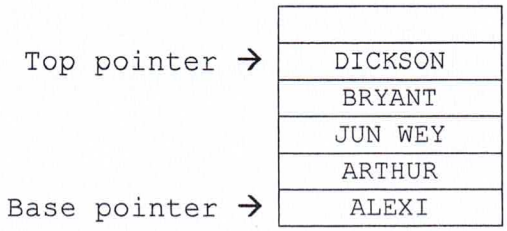

- Remove CONROY from the top of the stack.
- The top pointer moves down to point to BRYANT. This will show that CONROY is no longer in the stack.
- However, a copy of CONROY remains on the stack. Depending on the code used to implement the stack, CONROY can be deleted, replaced with a NULL value, or left to be overwritten in the future.

POP()



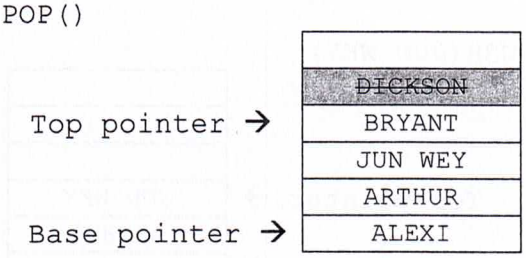

- DICKSON is now pushed onto the top of the stack.
- The copy of CONROY is now overwritten by DICKSON.
- The top pointer moves up to point to DICKSON.

PUSH(DICKSON)

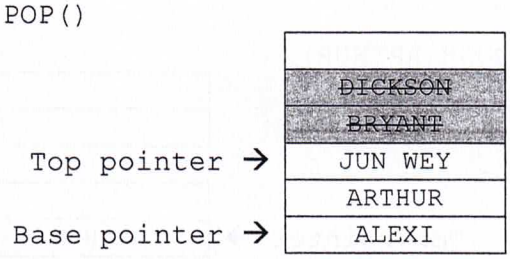


- Remove the data elements one by one until the stack is empty.
- Each time an element is removed, the top pointer moves down to point to the element that is now at the top of the stack

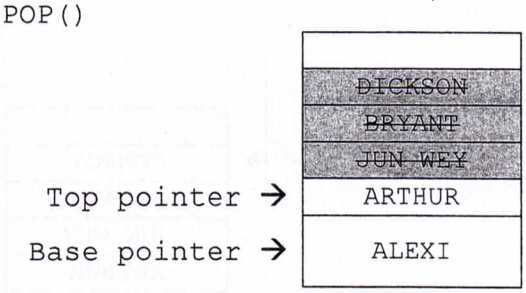
POP()



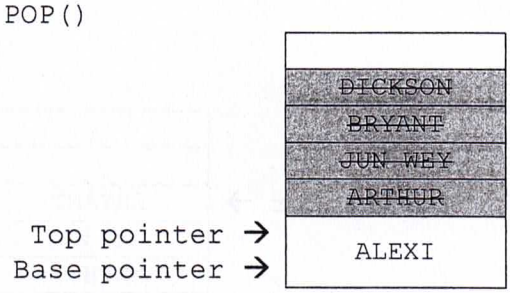

POP()



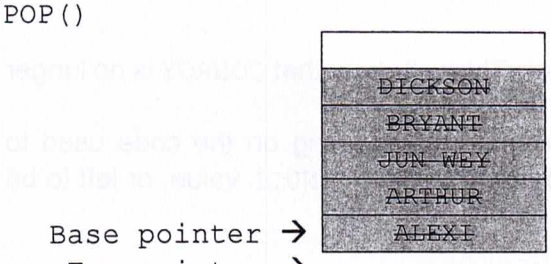

POP()




POP()




POP()



**Exercise 1**

Provide a stepwise illustration when the following stack operations are carried out on an empty stack that can store up to five data elements. Start with the illustration of the empty stack.

```
PUSH(JERESON)
PUSH(IAN)
PUSH(ANGEL)
PEEK()
POP()
PUSH(LAVANYA)
POP()
```

---

**[Solution]**

Initial empty stack

```
PUSH(JERESON)
```

```
PUSH(IAN)
```

```
PUSH(ANGEL)
```

```
PEEK()
RETURN ANGEL
```

```
POP()
```

```
PUSH(LAVANYA)
```

```
POP()
```

## §2.4 Stack Overflow and Stack Underflow

It is possible for a stack to need more memory than has been allocated to it.

In **Example 1**, the stack was implemented as a static data structure that can store only six data elements. If we attempt to push a seventh data element onto the top of the stack, this data element would have "nowhere to go". In this case a **stack overflow** error occurs.

A stack overflow error is an error that occurs when there is insufficient memory allocated to the stack to contain additional elements

Similarly, if the stack was empty and we try to pop an item, this would result in a **stack underflow** error as there is no more data element to be popped.

## §2.5 Implementation of Stacks using Static Arrays

A stack can involve a static or dynamic implementation. This commonly involves the use of a static array (in static implementation) or a linked list (dynamic implementation).

In this section, we shall focus on static implementation. We will delve into dynamic implementation after learning linked lists.

With a **static array**, the stack will have a fixed capacity, which means that if you continuously add items to the stack, it will result in a **stack overflow**. The `IS_FULL()` operation must be included to check whether a stack is at its maximum capacity. Similarly, **stack underflow** can occur if you try to remove elements from an empty stack. The `IS_EMPTY()` operation can be used to check if a stack is empty.

A constant `basePointer` represents the base pointer that will always point to the base of the stack. It commonly assumes the value `1`, which is the index of the first element in the static array.

A variable `topPointer` represents the top pointer. It will assume the index value of the last element in the static array. The top of the stack will change every time you add or remove a data element. This results in a corresponding change in the index value of the last element in the static array. Since the stack is initially empty, the initial value of `topPointer` is taken to be `0`. (Note: this is if the index of the array starts from 1)

| Process | Pseudocode | |
|---|---|---|
| Setting up a stack | `DECLARE stack ARRAY[1 : n] OF INTEGER`<br>`DECLARE global topPointer : INTEGER`<br>`DECLARE global basePointer : INTEGER`<br>`DECLARE global stackMaximum : INTEGER`<br><br>`basePointer ← 1`<br>`topPointer ← 0`<br>`stackMaximum ← LENGTH(ARRAY)` | |
| Defining the `IS_FULL()` function | `FUNCTION IS_FULL()`<br>`  IF topPointer = stackMaximum`<br>`  THEN`<br>`  RETURN TRUE`<br>`  ELSE`<br>`  RETURN FALSE`<br>`  ENDIF`<br>`ENDFUNCTION` | When the `topPointer` is of the same value as `stackMaximum`, the stack is full. |

| Process | Pseudocode | |
|---|---|---|
| Defining the `IS_EMPTY()` function | `FUNCTION IS_EMPTY()`<br>`  IF topPointer = basePointer - 1`<br>`  THEN`<br>`  RETURN TRUE`<br>`  ELSE`<br>`  RETURN FALSE`<br>`  ENDIF`<br>`ENDFUNCTION` | When the `topPointer` is 1 less than the `basePointer`, the stack is empty. |
| `PUSH` data element onto stack | `PROCEDURE PUSH(stack, data)`<br>`  IF IS_FULL() = TRUE`<br>`  THEN`<br>`  OUTPUT("Stack is full")`<br>`  ELSE`<br>`  topPointer ← topPointer + 1`<br>`  stack[topPointer] ← data`<br>`  ENDIF`<br>`ENDPROCEDURE` | When using a static array, you need to check that there is space in the stack, before pushing `data` onto the stack.<br><br>`IS_FULL()` can be used to check if the stack is full.<br><br>If there is room on the stack, `topPointer` is incremented by 1 and `data` is added. |
| `POP` data element from top of stack | `FUNCTION POP(stack)`<br>`  IF IS_EMPTY() = TRUE`<br>`  THEN`<br>`  OUTPUT("Stack is empty")`<br>`  RETURN NULL`<br>`  ELSE`<br>`  popItem ← stack[topPointer]`<br>`  topPointer ← topPointer - 1`<br>`  RETURN popItem`<br>`  ENDIF`<br>`ENDFUNCTION` | Before popping a data element from the stack, you need to check that the stack is not empty.<br><br>`IS_EMPTY()` can be used to check if the stack is empty.<br><br>If the stack is not empty, the data element at the top of the stack will be returned and `topPointer` will be decremented to point to the data element below.<br><br>Note that the element does not need to be removed from the array, as it can be overwritten next time a new item is pushed onto the stack. |
| `PEEK` element at the top of stack | `FUNCTION PEEK(stack)`<br>`  IF IS_EMPTY() = TRUE`<br>`  THEN`<br>`  OUTPUT("Stack is empty")`<br>`  RETURN NULL`<br>`  ELSE`<br>`  RETURN stack[topPointer]`<br>`  ENDIF`<br>`ENDFUNCTION` | When you peek an element, you must return the element that is located at the top of the stack, but you must **not** adjust the top pointer.<br><br>`IS_EMPTY()` is called to check if the stack is empty.<br><br>If the stack is not empty, the element referenced by `topPointer` will be returned. |

**Exercise 2**

**(i)** Write pseudocode for a static implementation of the stack data structure to store the following array of elements:

```
namelist = [BRAYDEN, AIDAN, PHONG, PARAM, PETER]
```

**(ii)** Write pseudocode to remove the top 2 elements from your stack in **(i)**.

**(iii)** Write pseudocode to check the topmost element of the stack after the 2 elements have been removed in **(ii)**.

**(iv)** Write Python code for **(i)**, **(ii)** and **(iii)**.

---

**[Solution]**

```
DECLARE n : INTEGER
n ← 5
FOR name IN namelist
    PUSH(stack, name)
ENDFOR
POP(stack)
POP(stack)
PRINT(PEEK(stack))
```

## §2.6 Applications of stacks

There are many uses for stacks. Due to their LIFO nature they can be used anywhere where you want the last data item in to be the first one out

Examples include:

- Backtracking the path taken in a game involving a maze.
- Maintaining a list of operations for the 'UNDO' function in MS Office programmes, where the most recent operation is the first to be undone.
- Reversing the contents of a list.
- Stack frames for storing information in a running program.

Can you think of more instances where the stack data structure is applicable?

---

**Exercise 3**

A list consists of five names: `[SAMUEL, SHERYL, SURAJ, LI XUAN, WEIZHI]`.

**(i)** Briefly outline how you can make use of an empty stack to reverse the order of the elements in the list.

**(ii)** Write Python code for the process you have described in **(i)**.

---

**[Solution]**

### §2.6.1 Stack Frames (Self-Read)

Consider the following pseudocode for calculating the volume of a cuboid with a square base:

```
FUNCTION get_area(side)
    RETURN side * side
ENDFUNCTION

FUNCTION get_volume(side, height)
    volume ← get_area(side) * height
    RETURN volume

MAIN
    OUTPUT get_volume(3, 4)
ENDMAIN
```

Let us try to understand what happens when the main program is executed:

- The main program is to output the return value of `get_volume(3, 4)`. It calls the function `get_volume`. The arguments `3` and `4` will be passed to the parameter `side` and `height` of `get_volume` respectively.
- The function `get_volume` is to assign the product of `get_area` and `height` to its local variable `volume`. It calls the function `get_area` which receives `3` as `side`.
- The `get_area` function is to return the product of `side * side`. It takes in the value `3` as `side` and calculates `side * side` as `9`, which is returned to the `get_volume` function.
- In the `get_volume` function, the return value of `get_area(3)`, which is `9`, is multiplied to the value of `height`, `4` to get the product `36`. The value `36` is assigned to the local variable `volume`, which is then returned to the main program.
- The main program outputs the return value of `get_volume(3, 4)`, which is `36`

Consider now the following:

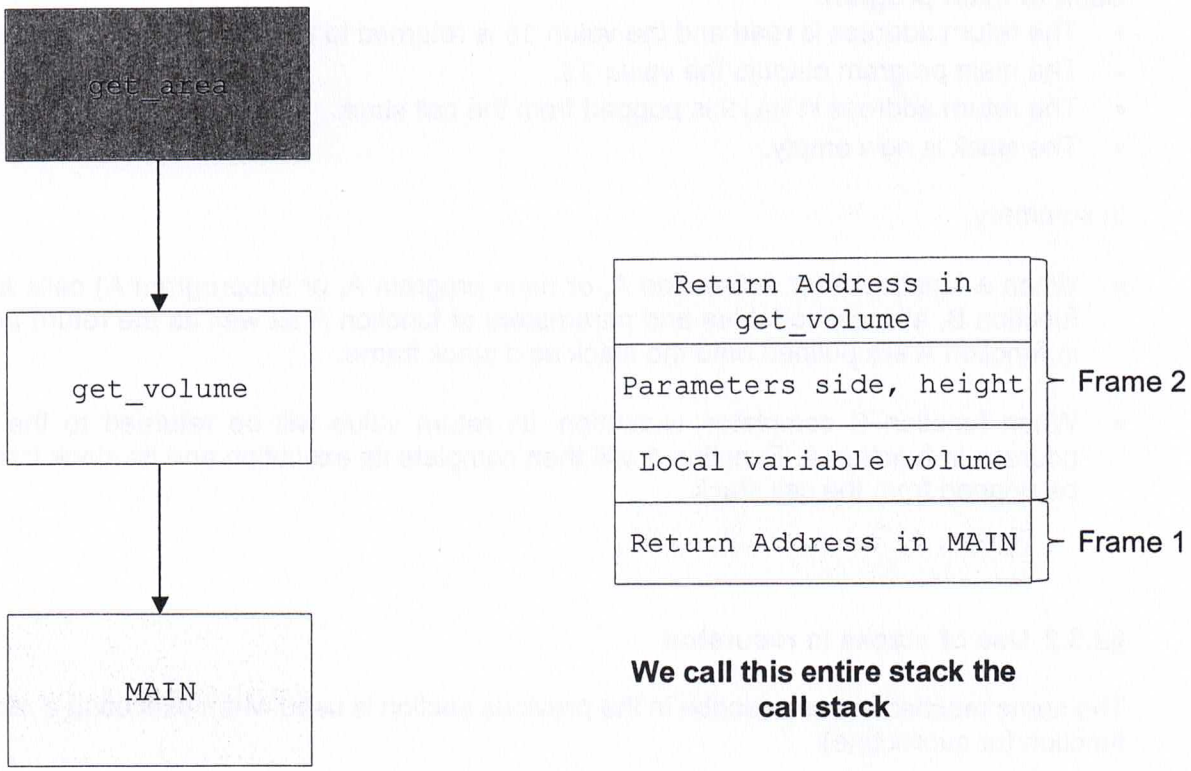
- Once the main program is executed, it will look at the first line of instruction
  `OUTPUT get_volume(3, 4)`
  and go to the function `get_volume`.
- However, it needs to remember to come back to the main program once it is done with executing `get_volume`. Otherwise, it will never proceed to `OUTPUT` the return value.
- Similarly, when the function `get_volume` is executed, it will look at the first line of instruction
  `volume ← get_area(side) * height`
  and go to the function `get_area`.
- After the `get_area` function is executed and the return value is obtained, the value needs to go back to `get_volume`, and at the right point within `get_volume`, otherwise, the volume will never be obtained.

It is thus important that the execution of the main program and the functions being called be accurately tracked to ensure that the correct output is obtained.

This is achieved through the creation of a **stack frame** every time a function (or subroutine, or main program, or subprogram) calls another function after its execution starts. Within a stack frame, the following data, where appropriate, are being stored:

- return address: this is the point to go back to when the function (or subroutine) being called has been executed completely
- local variables: these are the variables in the current function (or subroutine) that it should restore
- parameters: the data that is passed to the function (or subroutine).

Let us try to visualise this with a diagram:




**Main program**

- Main program has control.
- It calls the function `get_volume` and passes the arguments `3` and `4` to its parameters `side` and `height` respectively.
- The return address in `MAIN` is pushed onto the call stack.

**Stack frame for `get_volume`**

- Control goes to the `get_volume` function.
- The parameters `side` and `height` receive values `3` and `4` respectively.
- The local variable `volume` and the parameters `side` and `height` are pushed onto the call stack.
- The `get_volume` function calls the `get_area` function.
- The return address in `get_volume` is pushed onto the call stack.

**Execution of `get_area`**

- Control goes to the `get_area` function.
- The parameter `side` receives the value `3`.
- The `get_area` function returns the value of `side * side`, which is `9`.
- It executes its return statement.
- This causes the execution of the function to terminate.

**Back to stack frame for `get_volume`**

- The return address in `get_volume` is read and the value `9` is returned to this address.
- The `get_volume` function calculates the value of `9 * 4`, which is `36` and assigns it to the local variable `volume`.
- It executes its return statement.
- The return causes the execution of the function to terminate.
- The stack frame for `get_volume` is popped from the call stack.

**Back to main program**

- The return address is read and the value `36` is returned to this address.
- The main program outputs the value `36`.
- The return address in `MAIN` is popped from the call stack.
- The stack is now empty.

In summary,

- When a function A (or subroutine A, or main program A, or subprogram A) calls another function B, all local variables and parameters of function A as well as the return address in function A are pushed onto the stack as a stack frame.
- When function B completes execution, its return value will be returned to the return address in function A. Function A will then complete its execution and its stack frame will be popped from the call stack.

## §2.6.2 Use of stacks in recursion

The same mechanism as describe in the previous section is used when executing a recursive function (or subroutine).

Consider the following pseudocode for calculating the sum of all positive integers from 1 to n:

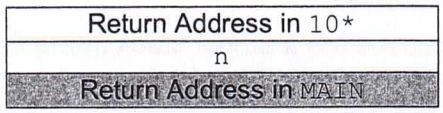
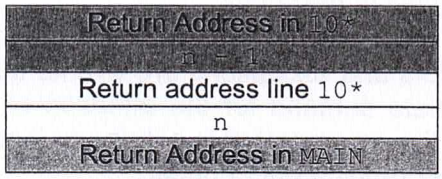
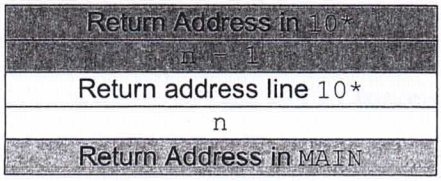
```
1  MAIN
2      OUTPUT sum_to_n(3)
3  ENDMAIN
4
5  FUNCTION sum_to_n(n)
6      IF n = 1
7          THEN
8              RETURN 1
9          ELSE
10             RETURN n + sum_to_n(n-1)
11     ENDIF
12 ENDFUNCTION
```

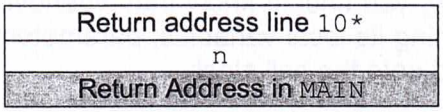
From the pseudocode, you can see that in the else clause, the function `sum_to_n` is called again with the argument `n - 1`. This **call to itself** is how you can identify that the function is recursive.

By now, you should know that whenever a function (or subroutine) is called, a stack frame containing its local variables, parameters and current address in memory (return address) is pushed onto the call stack.

To trace a recursive subroutine, you may find it useful to sketch out the stack frames.

Let us consider the case where n = 3:

| | |
|---|---|
| When the function `sum_to_n` is first called in `MAIN`, the return address in `MAIN` is pushed onto the call stack. |  |
| Control goes to the first call.<br><br>A stack frame consisting of the parameter `n` that takes in the argument `3` is pushed onto the stack.<br><br>The function runs to line `6` where the statement evaluates to `FALSE`.<br><br>It goes to line `9`, then line `10`. In line `10`, the function `sum_to_n` calls itself a second time with argument `(n - 1)` i.e. `3 - 1 = 2`<br><br>The return address in this call is pushed onto the stack. |  <br>Since the call is made at line `10`, let us use line `10*` as the return address for simplicity (`*` indicates the exact juncture in line `10`) |
| Control goes to the second call.<br><br>A stack frame for the second call of the function `sum_to_n` which takes in the argument `(n - 1)` is pushed onto the call stack.<br><br>The function runs to line `6` where the statement evaluates to `FALSE`.<br><br>It goes to line `9`, then line `10`. In line `10`, the function `sum_to_n` calls itself a third time with argument `(n - 1)` i.e. `2 - 1 = 1` |  |
| Control goes to the third call.<br><br>The function runs to line `6`, where the statement evaluates to `TRUE`.<br><br>It moves to line `7`, then line `8`. The value `1` is returned.<br><br>This causes the third call to terminate. |  |

| | |
|---|---|
| Control returns to the second call.<br><br>The return address is read and the return value `1` is returned to this address.<br><br>Line `6` continues to run by adding the return value `1` to `2`. The result `3` is returned.<br><br>This causes the topmost stack frame (stack frame of the second call) to be popped from the stack. |  |
| Control returns to the first call.<br><br>The return address is read and the return value `3` is returned to this address.<br><br>Line `6` continues to run by adding the return value `3` to `3`. The result `6` is returned.<br><br>This causes the topmost stack frame (stack frame of the first call) to be popped from the stack. |  |
| The return address in `MAIN` is read and the result `6` is returned as the `OUTPUT`.<br><br>The return address in MAIN is popped from the call stack and it is now empty. The function terminates. | |

### §2.6.3 Interrupt and Exception Handling

The same mechanism as describe in section 2.6.1 is used for handling interrupts and exceptions in programs.

Interrupts and exceptions are events that cause the current program to stop as there is an immediate demand for the attention of the processor This could be something happening inside the program that is running or it could be an external event, such as a power failure, or a printer running out of paper.

When this happens, special blocks of code called interrupt handlers and exceptions handlers are loaded into memory and executed. Whilst the new demand is being dealt with, the data of the current program are pushed onto a stack.

As soon as the interrupt or exception has been dealt with, the data will be popped off the stack and returned to the first program for execution from where it left off.

**Assignment**

Refer to Jupyter Notebook titled "`Data Structures 2 - Stacks.ipynb`"