LECTURE 10

ABSTRACT DATA STRUCTURES: STACKS

COMP1002J: Introduction to Programming 2

Dr. Brett Becker (brett.becker@ucd.ie)

Beijing-Dublin International College

Introduction

- Previously, for our linked list, we used a pointer to remember where the **head node** of the list was located
- We then wrote some functions to:
  - change the structure of the list
  - find information about the list
- Each time the list's head changed, we needed to store the new head in a variable, e.g.:
  - `head_node = add_element( head_node, 0 );`

Introduction

- This approach has some drawbacks, for example:
  - In a very large list, finding the length (size) of the list can take a long time:
    - We must visit every node in the list to count it
  - It would be quicker to store the length of the list in a variable

```
node_head = add_element( node_head, 5 );
list_size++;
```

  - **BUT:** This means that any programmer using the list must remember to increase and decrease the size every time they add or remove an element from the list. This is not the best way.

Introduction

- This approach has some drawbacks, for example (continued):
  - It is easy to add a new element to the start of the list, but:
  - It is not easy to add a new element to the end of the list:
    - We must iterate through every node in the list, to find the last node
    - Then we can add a new node after this one
    - Again, with a large list this can take a long time
    - You did this in last week’s lab!
  - As well as recording the **head** node, it might be useful to also record the **tail** node, which is the last node in the list
    - **BUT:** Again, this gives programmers using the list more work to do. We don't want them to get confused between the different variables that are used to store the list.

Abstract Data Types (ADTs)

- Inserting a value into an array/list and remembering how many items are stored in it is a **common problem**

- Like in other areas of engineering, programming research has investigated what common problems are faced by programmers

- They have also identified the most common solutions to those problems

- This area of study is known as **data structures and algorithms**
  - This includes problems like **searching** and **sorting**
    It also includes a number of **abstract data types (ADTs)**

Abstract Data Types (ADTs)

- An **Abstract Data Type (ADT)** is a typical solution for storing data that combines:
  - At least one structure
  - A number of functions that manipulate the structure(s)

- The functions will take care of typical tasks that involve storing and organising data:
  - Adding new items
  - Removing items
  - Searching for items
  - Finding the number of items

    ... and many more

Abstract Data Types(ADT): Stack

- Today we will look at a **stack** data structure
  - This is not the same as stack memory, where variables are created
  - But stack memory is called stack because it operates like a stack!
- A stack contains values
- Inserting and removing values is based on a last-in, first-out (LIFO) policy
  - A value can be inserted at any time**, but only the last (the most recently inserted) value can be accessed (for removal, or inspection)**
- Terminology:
  - Values can be **pushed** onto the stack (insertion)
  - Values can be **popped** off the stack (removal)
  - The **top** of the stack is the last value that was pushed
  - Therefore when the stack is popped, the value at the top is removed

Stacks: Main Algorithms / Operations

- Creating and Destroying:
  - `new_stack()`: Create a new (empty) stack
  - `dispose_stack()`: Destroy the stack and free its memory
- Core Operations (that change the data in the stack):
  - `push(e)`: Inserts element e onto top of stack
  - `pop()`: Removes the top object of stack and returns it
- Support Operations (that give information about the stack):
  - `size()`: Returns the number of objects in stack
  - `is_empty()`: Return a boolean indicating if stack is empty
  - `is_full()`: Return a Boolean indicating if stack is full
  - `top()`: Return the top object of the stack, without removing it (in other words ask what is at the top without changing the stack)

Stacks: Main Algorithms / Operations

- Creating and Destroying:
  - `new_stack()`: Create a new (empty) stack
  - `dispose_stack()`: Destroy the stack and free its memory
- Core Ope
  - `push(e`
  - `pop()`: d returns it

Some of these functions (like is_full and is_empty) can be used by other functions to make their jobs easier.

- Support Operations (that give information about the stack):
  - `size()`: Returns the number of objects in stack
  - `is_empty()`: Return a boolean indicating if stack is empty
  - `is_full()`: Return a Boolean indicating if stack is full
  - `top()`: Return the top object of the stack, without removing it (in other words ask what is at the top without changing the stack)

Stacks: Main Algorithms / Operations

- We might also modify our stack to do some other things. Strictly speaking though, these are not "pure stack" operations. So if we added these, we might call our stack a "modified stack"
  - `max()`: Returns the max value in the stack
  - `min()`: Returns the min value in the stack
  - Etc.

Stacks: Main Algorithms / Operations

Note that these are not specific to C.

A programmer working in any language would expect to have functions like this available.

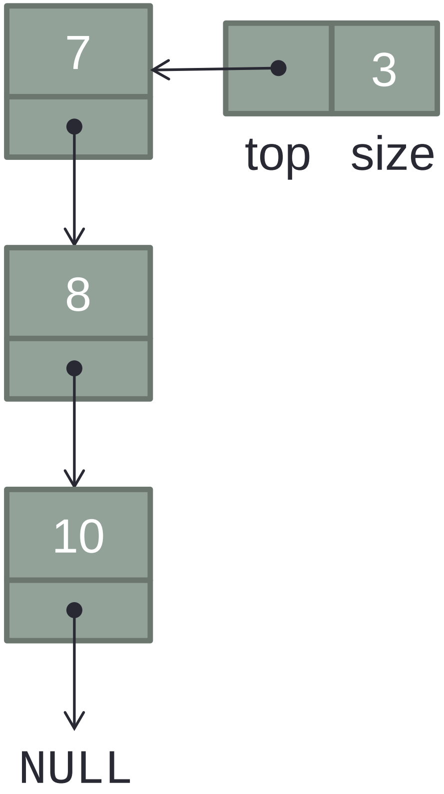
The specific details of implementing these in C will be seen later.

Stack ADT

- The reason we call this an **abstract** data type is because programmers using it do not need to know how it is implemented (how it works)
- They are only interested in the functions that are available, and that they work they way they are supposed to
- What would we expect to be in the stack if we do the following operations?
  - push(10), push(7), push(8), pop(), push(0), push(12), push(6), pop(), pop(), push(3)

Link-Based Stack

- Link based approaches combine: pointers and dynamic memory allocation to define an efficient implementation for Stacks (and other ADTs)
- The idea is that you think of the Stack as a linked list (i.e. a collection of linked nodes), with each node linked to the node beneath it in the Stack
- We use a variable to keep track of the **top** node in the Stack and can access the other nodes via this top node…
  - Just like the way we remembered the **head** node in our linked list

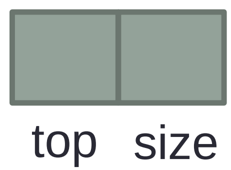


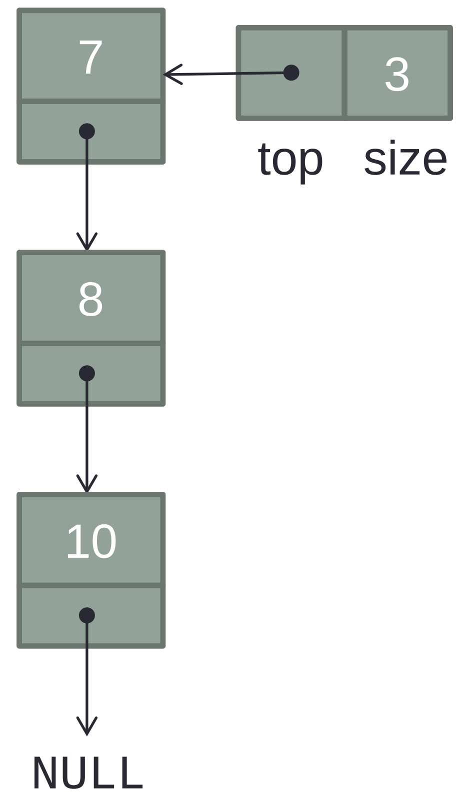
Linked Stack Types

- In developing this implementation, we must define two structures (the Node and the Stack):

```
typedef struct Node {
     int data;
     struct Node *next;
} Node;

typedef struct {
     Node *top;
     int size;
} Stack;
```





- *NOTE: The order of definition is important here: the Stack type depends on the Node type, so node comes first*
- *NOTE: Again, we will assume that our stack is to store* `int` *values only*

Linked Stack Types

- Programmers using our stack do not need to know that it is made up of nodes
- They only need to know that they have a pointer to a `Stack` structure that they can use with the functions provided
- Let's look at the functions again to see how they should look when implemented using C

Implementing a Stack in C

- Creating and destroying:
  - `Stack* new_stack()`
    - Creates a new (empty) stack. Returns a pointer to the new stack
  - `dispose_stack( Stack *sptr )`
    - Destroy the stack and free its memory.
    - Parameter:
      - A pointer to the stack that should be destroyed

Implementing a Stack in C

- Core operations (that change the data in the stack):
  - `bool push( Stack *sptr, int element)`
    - Inserts element onto top of stack. Returns true if the element was successfully inserted, or false otherwise.
    - Parameters:
      - A pointer to the stack
      - The element to be inserted

  - `int pop( Stack *sptr )`
    - Removes the top object of stack and returns it
    - Parameters:
      - A pointer to the stack

Implementing a Stack in C

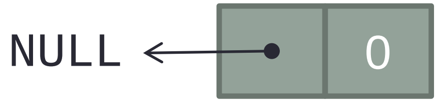
- Support Operations (that give information about the stack):
  - `int size( Stack *sptr )`:
    - Returns the number of elements in stack
    - Parameters:
      - A pointer to the stack
  - `bool is_empty( Stack *sptr)`:
    - Return a boolean indicating if stack is empty (true if it is empty, false otherwise)
    - Parameters:
      - A pointer to the stack
  - `int top( Stack *sptr)`:
    - Return the top element of the stack, without removing it
    - Parameters:
      - A pointer to the stack

New Stack

- Creating a new Stack:

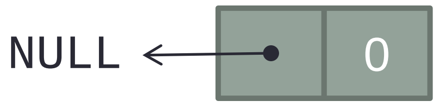
```
Stack* new_stack() {
    Stack *sptr = malloc(sizeof(Stack));
    if (sptr == NULL) return NULL;
    sptr->size = 0;
    sptr->top = NULL;
    return sptr;
}
```

- So the initial state of the stack is:



Pushing

- Pushing on to the stack:




```
bool push(Stack *sptr, int value) {
    Node *node = malloc(sizeof(node));
    if (node == NULL) return false;

    node->data = value;
    node->next = sptr->top;
    sptr->top = node;
    sptr->size++;
    return true;
}
```

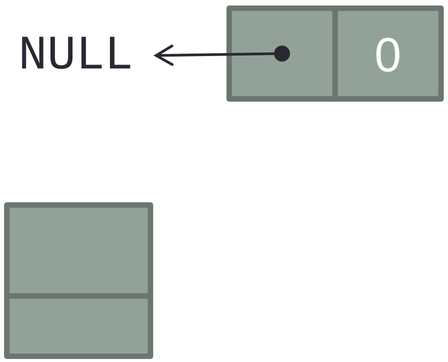
- Example: push(15), push(2)

Pushing

- Pushing on to the stack:

```
bool push(Stack *sptr, int value) {
    Node *node = malloc(sizeof(Node));
    if (node == NULL) return false;

    node->data = value;
    node->next = sptr->top;
    sptr->top = node;
    sptr->size++;
    return true;
}
```



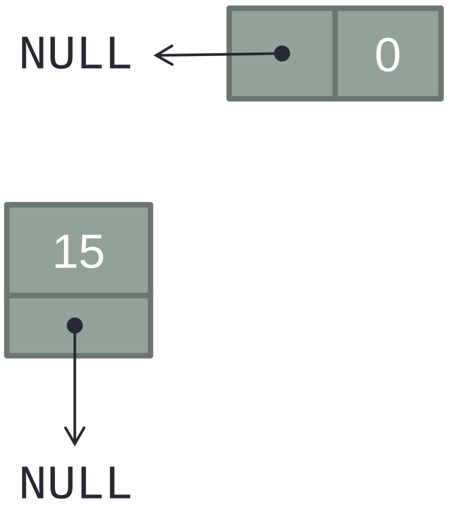
- Example: **push(15)**, push(2)

Pushing

- Pushing on to the stack:

```
bool push(Stack *sptr, int value) {
     Node *node = malloc(sizeof(node));
     if (node == NULL) return false;

     node->data = value;
     node->next = sptr->top;
     sptr->top = node;
     sptr->size++;
     return true;
}
```



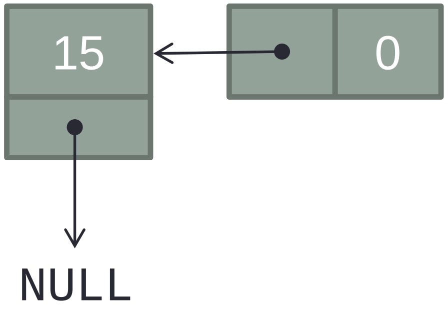
- Example: **push(15)**, push(2)

Pushing

- Pushing on to the stack:

```
bool push(Stack *sptr, int value) {
    Node *node = malloc(sizeof(node));
    if (node == NULL) return false;

    node->data = value;
    node->next = sptr->top;
    sptr->top = node;
    sptr->size++;
    return true;
}
```



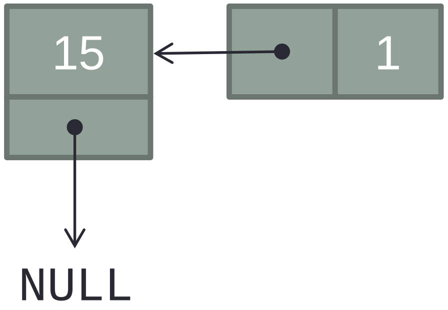
- Example: **push(15)**, push(2)

Pushing

- Pushing on to the stack:

```
bool push(Stack *sptr, int value) {
    Node *node = malloc(sizeof(node));
    if (node == NULL) return false;

    node->data = value;
    node->next = sptr->top;
    sptr->top = node;
    sptr->size++;
    return true;
}
```



- Example: **push(15)**, push(2)

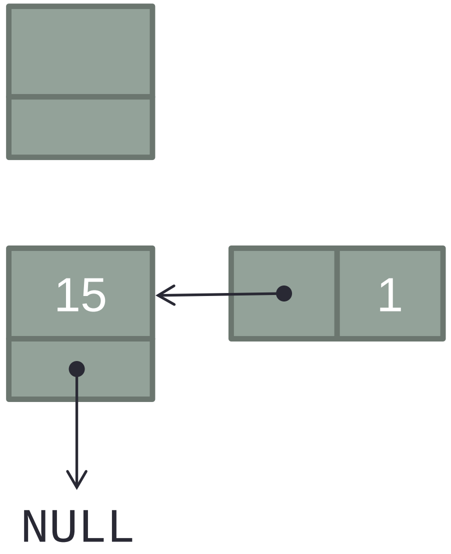
Pushing

- Pushing on to the stack:

```
bool push(Stack *sptr, int value) {
    Node *node = malloc(sizeof(node));
    if (node == NULL) return false;

    node->data = value;
    node->next = sptr->top;
    sptr->top = node;
    sptr->size++;
    return true;
}
```

- Example: push(15), **push(2)**



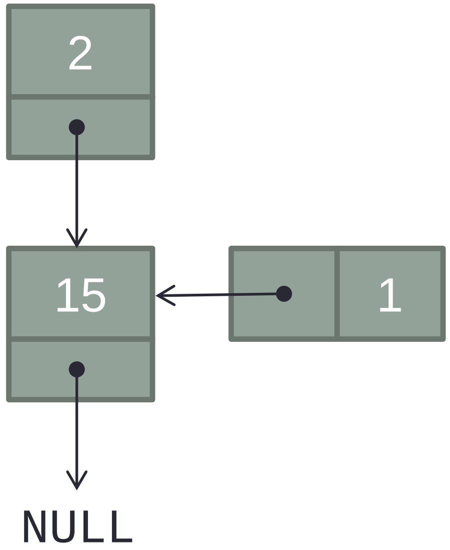
Pushing

- Pushing on to the stack:

```
bool push(Stack *sptr, int value) {
    Node *node = malloc(sizeof(node));
    if (node == NULL) return false;

    node->data = value;
    node->next = sptr->top;
    sptr->top = node;
    sptr->size++;
    return true;
}
```

- Example: push(15), **push(2)**

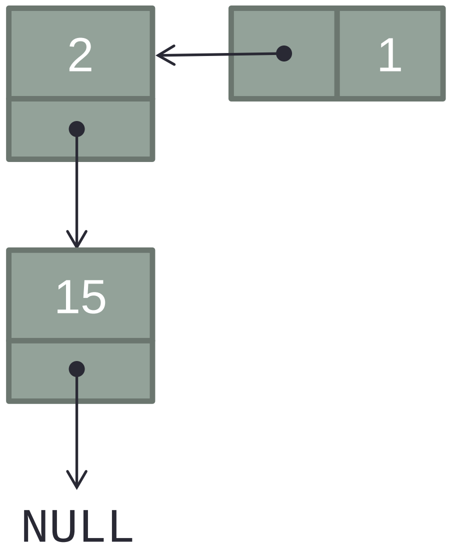


Pushing

- Pushing on to the stack:

```
bool push(Stack *sptr, int value) {
     Node *node = malloc(sizeof(node));
     if (node == NULL) return false;

     node->data = value;
     node->next = sptr->top;
     sptr->top = node;
     sptr->size++;
     return true;
}
```



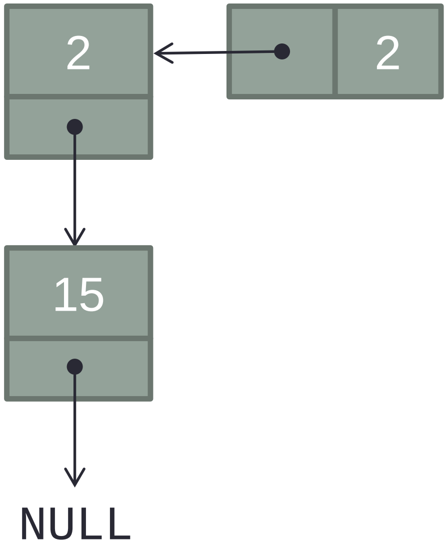
- Example: push(15), **push(2)**

Pushing

- Pushing on to the stack:

```
bool push(Stack *sptr, int value) {
    Node *node = malloc(sizeof(node));
    if (node == NULL) return false;

    node->data = value;
    node->next = sptr->top;
    sptr->top = node;
    sptr->size++;
    return true;
}
```

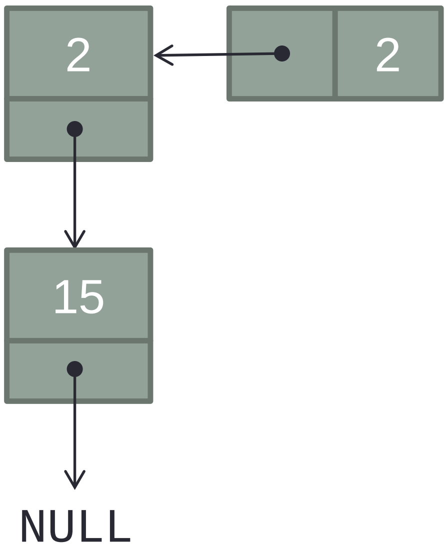


- Example: push(15), **push(2)**

Popping

- Popping from the stack:

```
int pop(Stack *sptr) {
    Node *node = sptr->top;
    int data = sptr->top->data;
    sptr->top = sptr->top->next;
    sptr->size--;
    free(node);
    return data;
}
```



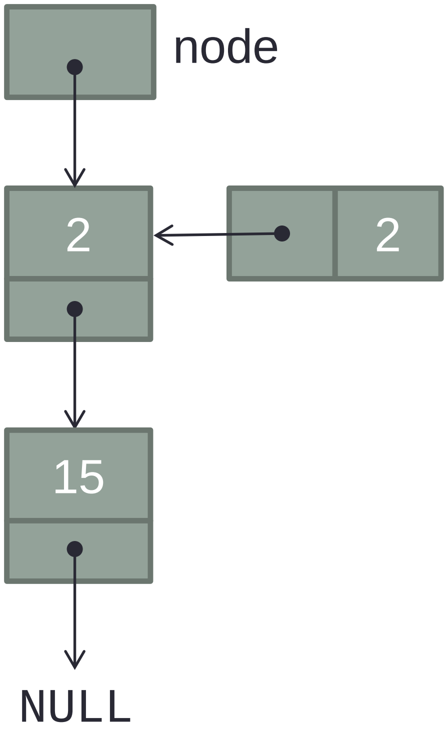
- Example: pop(), pop()

Popping

- Popping from the stack:

```
int pop(Stack *sptr) {
    Node *node = sptr->top;
    int data = sptr->top->data;
    sptr->top = sptr->top->next;
    sptr->size--;
    free(node);
    return data;
}
```

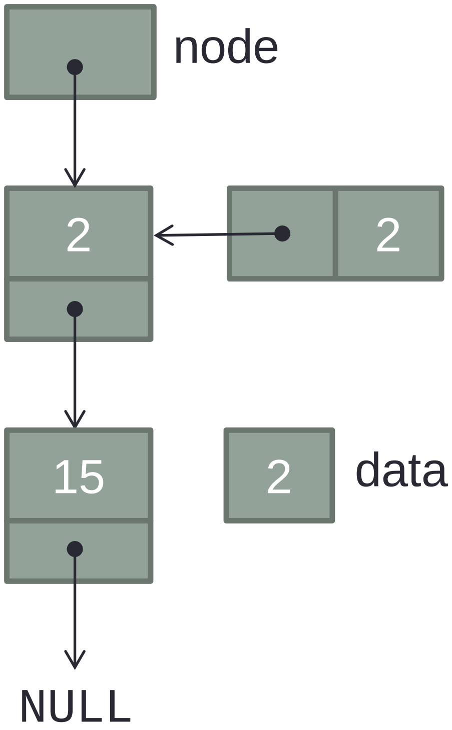
- Example: **pop()**, pop()



Popping

- Popping from the stack:

```
int pop(Stack *sptr) {
    Node *node = sptr->top;
    int data = sptr->top->data;
    sptr->top = sptr->top->next;
    sptr->size--;
    free(node);
    return data;
}
```



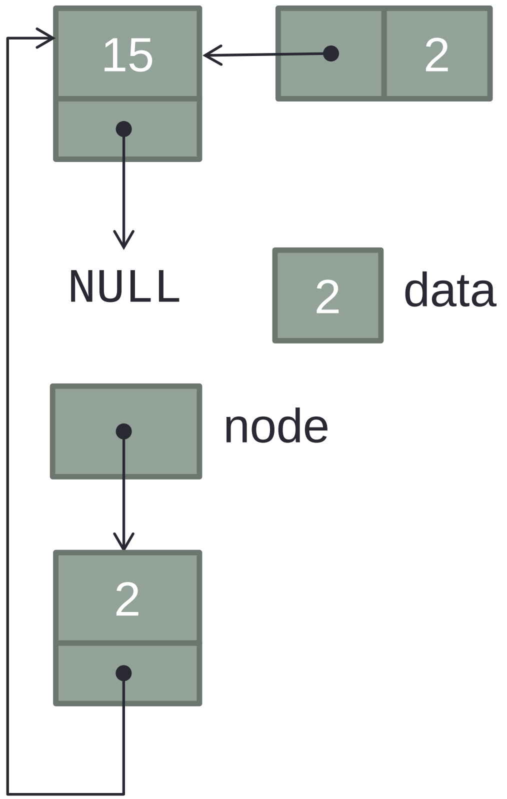
- Example: **pop()**, pop()

Popping

- Popping from the stack:

```
int pop(Stack *sptr) {
     Node *node = sptr->top;
     int data = sptr->top->data;
     sptr->top = sptr->top->next;
     sptr->size--;
     free(node);
     return data;
}
```

- Example: **pop()**, pop()

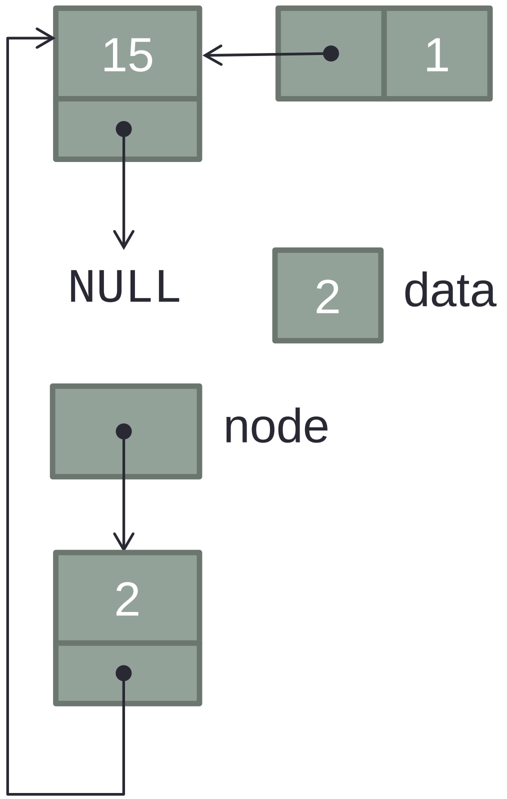


Popping

- Popping from the stack:

```
int pop(Stack *sptr) {
    Node *node = sptr->top;
    int data = sptr->top->data;
    sptr->top = sptr->top->next;
    sptr->size--;
    free(node);
    return data;
}
```

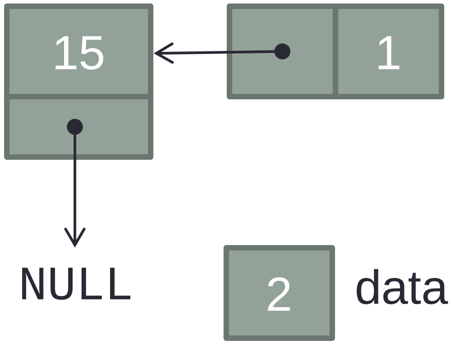
- Example: **pop()**, pop()



Popping

- Popping from the stack:

```
int pop(Stack *sptr) {
    Node *node = sptr->top;
    int data = sptr->top->data;
    sptr->top = sptr->top->next;
    sptr->size--;
    free(node);
    return data;
}
```

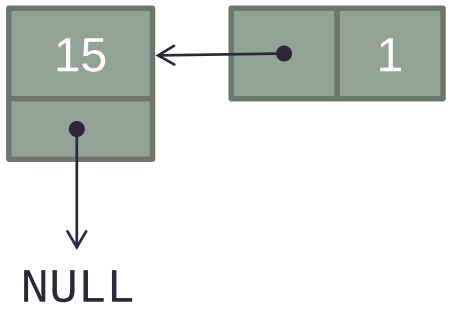


- Example: **pop()**, pop()

Popping

- Popping from the stack:

```
int pop(Stack *sptr) {
     Node *node = sptr->top;
     int data = sptr->top->data;
     sptr->top = sptr->top->next;
     sptr->size--;
     free(node);
     return data;
}
```



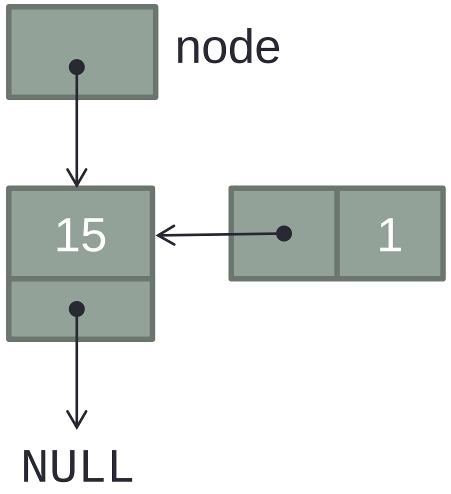
- Example: **pop()**, pop()

Popping

- Popping from the stack:

```
int pop(Stack *sptr) {
    Node *node = sptr->top;
    int data = sptr->top->data;
    sptr->top = sptr->top->next;
    sptr->size--;
    free(node);
    return data;
}
```

- Example: **pop()**, **pop()**

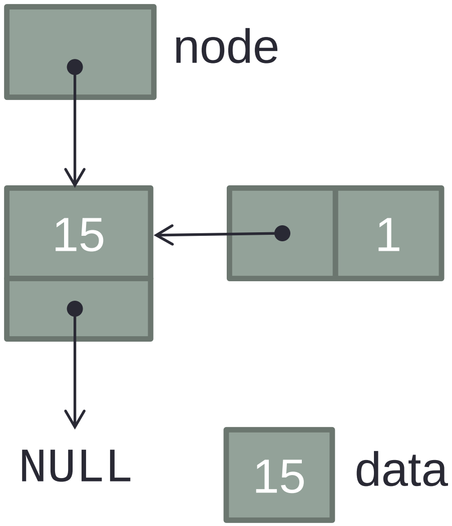


Popping

- Popping from the stack:

```
int pop(Stack *sptr) {
    Node *node = sptr->top;
    int data = sptr->top->data;
    sptr->top = sptr->top->next;
    sptr->size--;
    free(node);
    return data;
}
```

- Example: **pop()**, **pop()**

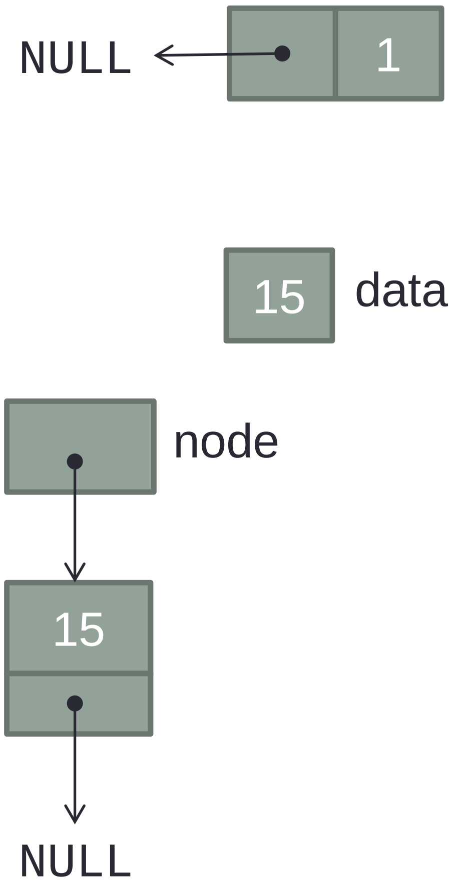


Popping

- Popping from the stack:

```
int pop(Stack *sptr) {
    Node *node = sptr->top;
    int data = sptr->top->data;
    sptr->top = sptr->top->next;
    sptr->size--;
    free(node);
    return data;
}
```

- Example: **pop()**, **pop()**

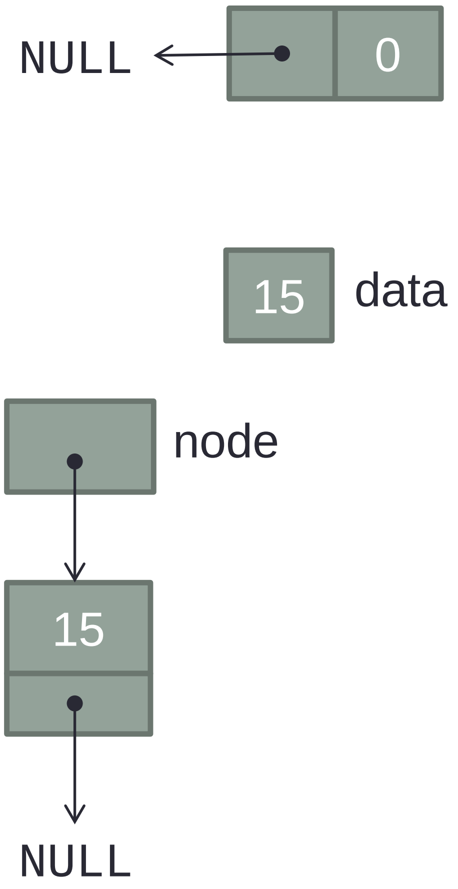


Popping

- Popping from the stack:

```
int pop(Stack *sptr) {
    Node *node = sptr->top;
    int data = sptr->top->data;
    sptr->top = sptr->top->next;
    sptr->size--;
    free(node);
    return data;
}
```

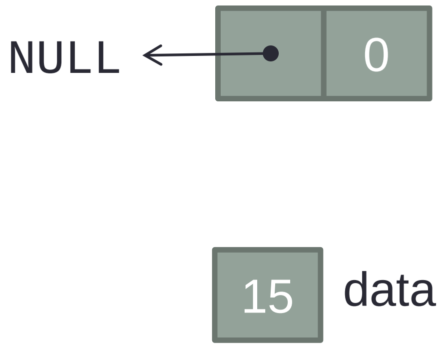
- Example: **pop()**, **pop()**



Popping

- Popping from the stack:

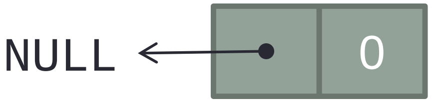
```
int pop(Stack *sptr) {
    Node *node = sptr->top;
    int data = sptr->top->data;
    sptr->top = sptr->top->next;
    sptr->size--;
    free(node);
    return data;
}
```



- Example: **pop()**, **pop()**

Popping

- Popping from the stack:




```
int pop(Stack *sptr) {
    Node *node = sptr->top;
    int data = sptr->top->data;
    sptr->top = sptr->top->next;
    sptr->size--;
    free(node);
    return data;
}
```

- Example: **pop()**, **pop()**

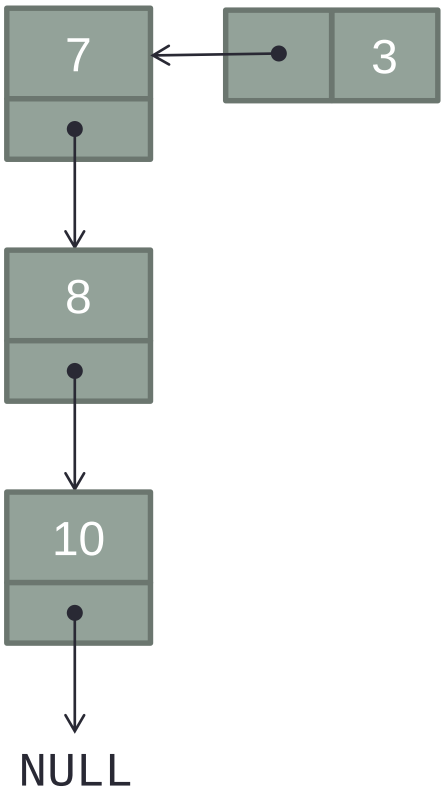
Other Functions

- Is Empty:

```
bool is_empty(Stack *sptr) {
     return sptr->size == 0;
}
```

- Dispose Stack:

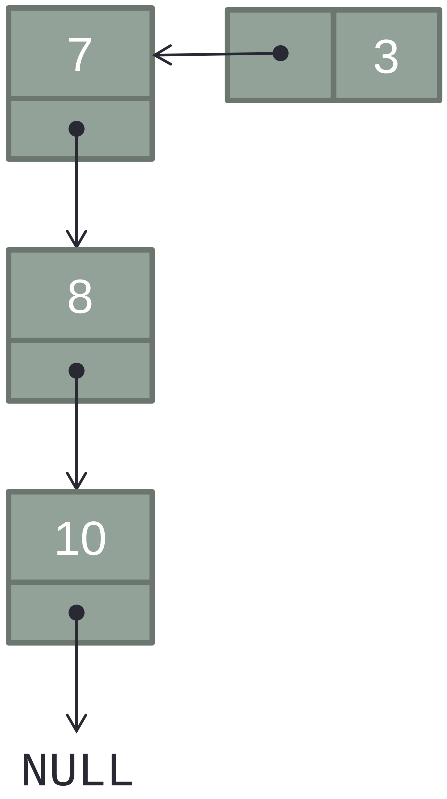
```
void dispose_stack(Stack *sptr) {
     while (!is_empty(sptr)) {
            pop(sptr);
     }
     free(sptr);
}
```



Other Functions

- Display Stack:

```
void display(Stack *sptr) {
     printf("values: ");
     Node *cur = sptr->top;
     while (cur != NULL) {
          printf("%d ", cur->data);
          cur = cur->next;
     }
     printf("\nsize: %d\n\n", sptr->size);
}
```



Example Program

```
int main() {
	Stack *stack = new_stack();

	printf("pushing 1");
	push(stack, 1);
	display(stack);

	printf("pushing 7");
	push(stack, 7);
	display(stack);

	printf("popping");
	pop(stack);
	display(stack);

	printf("pushing 4");
	push(stack, 4);
	display(stack);

	printf("popping");
	pop(stack);
	display(stack);

	printf("pushing 3");
	push(stack, 3);
	display(stack);

	dispose_stack(stack);
	return(0);
}
```

Output:

pushing 1
values: 1
size: 1

pushing 7
values: 7 1
size: 2

popping
values: 1
size: 1

pushing 4
values: 4 1
size: 2

popping
values: 1
size: 1

pushing 3
values: 3 1
size: 2

file: stack_example.c

Stacks and Arrays

- A different way to implement a stack is to use arrays
- Remember, a programmer using a stack doesn't care how it is implemented
- As long as the functions work, and do what is expected, we can implement it any way we like
- For many ADTs, there are many different ways we can implement them
  - Often, no implementation is better than all the others for all of the operations
- Let’s design a stack-based array to store **positive** integers.

Stacks and Arrays

- The main structure combines an array (to store the values) and an integer value to keep track of how full the array is
- We must also track "capacity": the number of items that the stack can hold

- Lets look at an integer stack:

```
typedef struct {
    int capacity;
    int *data;
    int top;
} Stack;
```

- The `top` field is the one used to keep track of how full the array is… basically this is the ‘position’ in the array where the top item is

Stacks and Arrays

- Imagine that the capacity is set to 10
  - We can represent an IntegerStack as follows

0	1	2	3	4	5	6	7	8	9	top	capacity
											10

- An empty stack looks like this

0	1	2	3	4	5	6	7	8	9	top	capacity
										0	10

Stacks and Arrays

- If we push a value (10) onto the stack, we want the stack to look like this afterwards:

0	1	2	3	4	5	6	7	8	9	top	capacity
10										1	10

- The idea here is that top records the number of items stored in the stack
- It also happens that top is also **the index of the cell at which the <u>next</u> item should be stored** in the array…
- For example: if we push 7 onto the stack we get:

0	1	2	3	4	5	6	7	8	9	top	capacity
10	7									2	10

Stacks and Arrays

- Lets see what happens with the following operations:
  - Push(10), Push(7), Push(8), Pop(), Push(0), Push(12), Push(6), Pop(), Pop(), Push(3).
- Draw the state of the stack after EACH OPERATION.
- Some more exercises you can try:
  - Push(5), Pop(), Push(11), Push(5), Push(2), Push(13), Pop(), Push(3), Push(4), Push (8)
  - Push(13), Pop(), Push(11), Push(9), Push(1), Pop(), Pop(), Push(3), Pop(), Pop()

Stacks and Arrays

- So, pushing involves inserting a value into a cell in the array and updating top.
- The function implementation of this operation is:

```
void push(Stack *sptr, int value) {
        sptr->data[sptr->top++] = value;
}
```

- What is wrong with this code?
- What happens if we push a value on to the following stack?

0	1	2	3	4	5	6	7	8	9	top	capacity
10	7	3	8	2	10	9	6	3	1	10	10

Stacks and Arrays

- So, we need to check whether or not the stack is full.

- To do this, we can introduce a new operation (function) that checks whether or not the stack is full:

```
bool is_full(Stack *sptr) {
   return (sptr->top == sptr->capacity);
}
```

- We can call this function inside the push function to check whether or not we can push
  - We will worry about pushing to a full stack later.

Stacks and Arrays

- When we pop, we remove a value from the stack and return the value.
- For example, consider what happens if we pop from this stack:

0	1	2	3	4	5	6	7	8	9	top	capacity
10	7	8	9							4	10

- In this case, we want to remove the value 9 from the stack and return it…
- Afterwards, the state should be:

0	1	2	3	4	5	6	7	8	9	top	capacity
10	7	8	9							3	10

Stacks and Arrays

- Note that we do not remove the 9 from position 3. **WHY NOT?**

- What does the array contain from position 4 to position 9?
  - **Anything/rubbish!**

- The only important cells are the ones that are in the stack (from 0 to 2 in this example). For the others, it is OK for them to contain any values at all, including 9.

0	1	2	3	4	5	6	7	8	9	top	capacity
10	7	8	9							3	10

Stacks and Arrays

- We can write this as the following function:

```
int pop(Stack *sptr) {
        sptr->top--;
        return sptr->data[sptr->top];
}
```

- Again, what happens if we try to pop in this scenario:

0	1	2	3	4	5	6	7	8	9	top	capacity
										0	10

Stacks and Arrays

- We need to check that the stack is not empty before we perform a pop.

- We can do this check using the following function (which is already an operation):

```
bool is_empty(Stack *sptr) {
        return (sptr->top == 0);
}
```

- Failure to call this function before a pop operation can lead to your program working incorrectly…

Stacks and Arrays

- So a better pop is:

```
int pop(Stack *sptr) {
        if(is_empty( sptr )){
                printf( "The stack is empty!" );
                return -1;
        }
        else{
                sptr->top--;
                return sptr->data[sptr->top];
        }
}
```

Stacks and Arrays

- We need to also write the new_stack() function.
- This is a little different to a link-based stack, because this time we need to allocate memory for the stack structure but also for the array to store the data.
- Let's assume that every stack begins with a capacity of 5.

Stacks and Arrays

```
Stack* new_stack() {
  int capacity = 5;
  Stack *stack = malloc( sizeof( Stack ) );



  // we will fill in this bit later!
  return stack;
}
```

Set the initial capacity to 5.

Allocate enough memory for the Stack structure.

Stacks and Arrays

```
Stack* new_stack() {
  int capacity = 5;
  Stack *stack = malloc( sizeof( Stack ) );
  if ( stack == NULL ) return NULL;



  // we will fill in this bit later!
  return stack;
}
```

If malloc() could not give us the memory we need, return NULL.

Stacks and Arrays

```
Stack* new_stack() {
  int capacity = 5;
  Stack *stack = malloc( sizeof( Stack ) );
  if ( stack == NULL ) return NULL;
  stack->top = 0;
  stack->capacity = capacity;



  // we will fill in this bit later!
  return stack;
}
```

Set the values of top and capacity.

Stacks and Arrays

```
Stack* new_stack() {
   int capacity = 5;
   Stack *stack = malloc( size
   if ( stack == NULL ) return
   stack->top = 0;
   stack->capacity = capacity;
   stack->data = malloc(sizeof(int)* capacity);



   // we will fill in this bit later!
   return stack;
}
```

Now allocate enough memory for the data.

What happens if malloc() fails this time?

Stacks and Arrays

```
Stack* new_stack() {
  int capacity = 5;
  Stack *stack = malloc( sizeof( Stack ) );
  if ( stack == NULL ) return NULL;
  stack->top = 0;
  stack->capacity = capacity;
  stack->data = malloc(sizeof(int)* capacity);
  if ( stack->data == NULL ) {
    free( stack );
    return NULL;
  }
  return stack;
}
```

If the second malloc() fails, then we cannot create the stack.

We must free all the memory we got before, and then return NULL.

Stacks and Arrays

```
Stack* new_stack() {
  int capacity = 5;
  Stack *stack = malloc( sizeof( Stack ) );
  if ( stack == NULL ) return NULL;
  stack->top = 0;
  stack->capacity = capacity;
  stack->data = malloc(sizeof(int)* capacity);
  if ( stack->data == NULL ) {
    free( stack );
    return NULL;
  }
  return stack;
}
```

If everything was OK, we can return the new (empty) stack.

Stacks and Arrays

```
Stack* new_stack() {
    int capacity = 5;
    Stack *stack = malloc( sizeof( Stack ) );
    if ( stack == NULL ) return NULL;
    stack->top = 0;
    stack->capacity = capacity;
    stack->data = malloc(sizeof(int)* capacity);
    if ( stack->data == NULL ) {
```

Note that if we want an if statement to only have one statement, we can skip the brackets.
if (stack == NULL) return NULL;
Is exactly the same as
if (stack == NULL) {return NULL};

You only NEED the brackets if you want your if to have more than one statement:

```
if(something){
    Statement 1;
    Statement 2;
}
```

Stacks and Arrays

- In addition to the operations discussed so far, here are 3 more operations we might want (note, some of these are not *strictly* stack operations, so our stack might be called a *modified stack)*:

```
int size(Stack *sptr) {
         //should return the number of items in the stack
}

int top(Stack *sptr) {
         //returns the data at the top of the stack
         //without popping the stack
}

int max(Stack *sptr) {
        //returns the max value in the stack
}
```

Stacks and Arrays

- Earlier, we asked what would happen when somebody tries to push to a stack that is full
- We need to allocate more memory to the stack
- Let's try to double the capacity of the stack each time this happens

Stacks and Arrays

```
bool push( Stack *sptr, int value ) {
  if ( is_full( sptr ) ) {



  }
  sptr->data[ sptr->top++ ] = value;
  return true;
}
```

We can use is_full(...) to check if the stack is full.

Stacks and Arrays

```
bool push( Stack *sptr, int value ) {
  if ( is_full( sptr ) ) {
    int *new_data;
    new_data = realloc( sptr->data,
           sizeof( int )*sptr->capacity*2);



  }
  sptr->data[ sptr->t
  return true;
}
```

realloc(...) can be used to get a new block of memory on the heap.

The contents of the old memory are copied here, so the data is kept.

I have created a new pointer variable: **WHY**?

Just like malloc(...), realloc(...) can fail too!

If it does fail, return false to show that the operation was not successful.

But if it fails, the old memory is not changed, so sptr->data still points to the old (full) memory.

If I had done sptr->data = realloc(...) then my stack's contents would be gone if it failed.

Stacks and

```
bool push( St
  if ( is_ful
    int *new_
    new_data
           sizeof( int )*sptr->capacity*2);
    if ( new_data == NULL ) return false;



  }
  sptr->data[ sptr->top++ ] = value;
  return true;
}
```

Aside: realloc(...) the wrong way

- What if I just did this instead?
  - `sptr->data = realloc( sptr->data, sizeof(int)*sptr->capacity * 2);`
- Below is a diagram of a stack that is full.
- We will see what happens using the above line of code in two situations:
  1. The call to realloc(...) succeeds.
  2. The call to realloc(...) fails.

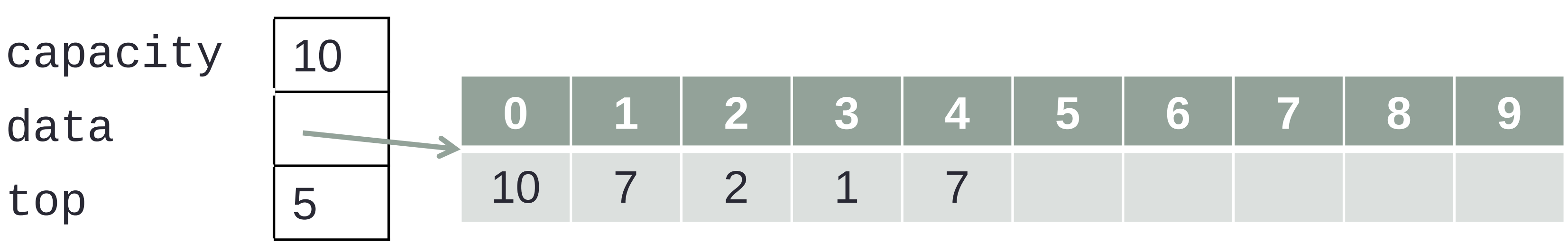
`capacity`	5
`data`	
`top`	5

0	1	2	3	4
10	7	2	1	7

realloc(...) success

- If `realloc(...)` succeeds, it allocates enough memory (in this case, space for 10 `ints`).
- The data that was previously stored in the memory `sptr->data` points to is copied into the new memory.
- It returns a pointer to this memory, which is stored in `sptr->data`. Everying is OK!

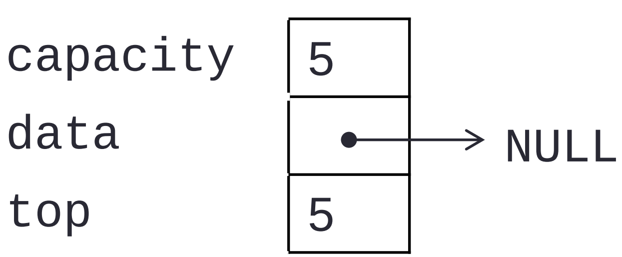
```
sptr->data = realloc( sptr->data,
    sizeof( int )*sptr->capacity * 2);
```



realloc(...) failure

- If `realloc(...)` fails, it returns a null pointer, just like `malloc(...)`.
- This null pointer is stored in `sptr->data`.
- **BUT:** The memory that sptr->data previously pointed to has not been deallocated, and now we have no pointer to it.
- A **memory leak** has occurred.

```
sptr->data = realloc( sptr->data,
    sizeof( int )*sptr->capacity * 2);
```



0	1	2	3	4
10	7	2	1	7

Stacks and Arrays

```
bool push( Stack *sptr, i
  if ( is_full( sptr ) ) 
    int *new_data;
    new_data = realloc( s
           sizeof( int )*sptr->capacity*2);
    if ( new_data == NULL ) return false;
    sptr->data = new_data;
    sptr->capacity *= 2;
  }
  sptr->data[ sptr->top++ ] = value;
  return true;
}
```

Since the realloc(...) succeeded, we can set sptr->data to point at the new memory, and the capacity has doubled.

file: array_stack_example.c

Summary

- This lecture has discussed **Abstract Data Types (ADTs)**.
- These are common ways of storing and organising data, consisting of:
  - One or more structures.
  - A collection of functions to carry out the common operations (adding, removing, searching, etc.).
- The ADT we have studied is the **stack**.
- Even though the function names don't change, it is possible to implement a stack in two different ways: one using a linked-list and the other using an array to store the data.
- There are many other data structures: queues, trees, etc.