# Data Structures and Algorithms

## Abstract Data Types

Dr. Lina Xu
lina.xu@ucd.ie

School of Computer Science,
University College Dublin

October 18, 2018

# Learning outcomes

After this lecture and the related practical students should...

- understand an what an abstract data type is
- understand the operations of the stack abstract data type
- be able to implement an array based stack
- be able to implement a link based stack

# Table of Contents

1. Abstract Data Types
2. Stack Abstract Data Type
3. Array Based Implementation
4. Link Based Stack Implementation

# Abstract Data Types (ADT)

- An abstract data type is the idea of a data structure
- An abstract data type states what the data structure should be able to do
    - The operations is should be able to perform
- An abstract data type does not define how the functionality should be implemented

# Abstract Data Types

- Abstract data types are used to identify
  - The charactersitics of the data within the data structure
  - The operations that can be applied to that data structure
- The issues that we face when understanding abstract data types are
  - Understanding how an abstract data type can be implemented
  - Understanding the advantages and disadvantages of different implementations
  - Knowing which of the implementation of the abstract data type to use in different situations

# How we will study Abstract Data Types

- First we will look at a conceptual overview of the abstract data type
  - We want to understand what the abstract data type is trying to achieve
- We will look at a more functional specification of the abstract data type
  - We want to understand what operations should exist in the abstract data type
- We will define the interface of the abstract data type
  - To do this we convert the functional specifications into a Java interface
- We will investigate the different implementation strategies
  - We want to understand the different ways that the abstract data type can be implemented
  - We will implement some of these strategies
  - We will compare different implementations, and look at the advantages and disadvantages of each

# Table of Contents

1 Abstract Data Types

2 Stack Abstract Data Type

3 Array Based Implementation

4 Link Based Stack Implementation

# The Stack Abstract Data Type

## Concept

The stack is one of the easiest abstract data types to understand and implement

- A stack is a container for items
- The main idea of the stack is that insertion and removal from the data structure are based on the last-in-first-out (LIFO) principle
  - This means that whenever you remove an item from the stack, it will always be the last item that was inserted and is still on the stack
- Special terminology is used to describe the operations in a stack
  - Items are pushed onto the stack (insertion)
  - Items are popped off of the stack (removal)
  - The top of the stack is that last item that was pushed onto the stack

# The Stack Abstract Data Type

## Functional Specification

- Stacks should work with any type of data
  - In our implementations we will only use integers to simplify the process
- Core operations of the stack are:
  - `push(i)`: The integer i is inserted onto the top of the stack
  - `pop()`: The object on the top of the stack is removed and returned by the method
  - `top()`: The object on the top of the stack is returned by the method but not removed
- The stack also contains some support operations that make it easier to use
  - `size()`: Returns the number of items currently stored in the stack
  - `isEmpty()`: Returns `true` if the stack has no items and `false` if there are none

# The Stack Abstract Data Type

## Java Interface

```java
public interface Stack {
  int size();

  boolean isEmpty();

  int top();

  void push(int o);

  int pop();
}
```

# The Stack Abstract Data Type

Implementation Strategies

There are two typical implementations of the stack abstract data type

- Array based implementation
  - Data is actually stored in an array
  - Extra variable required to remember what index stores the top item in the array
  - Finite capacity - limited by size of array
- Link based implementations
  - Elements are stored in custom objects called nodes
  - Object references are used to keep track of the order of the items
  - Infinite capacity - Can grow and shrink as more items are added and removed

# Table of Contents

1 Abstract Data Types

2 Stack Abstract Data Type

3 Array Based Implementation

4 Link Based Stack Implementation

# Array based stack implementation

To implement the stack abstract data type using an array first we need a class

- We will create a class called `ArrayStack`
- In this class we will create an array to store the data
- The methods we write will allow the data to be accessed in the way we want
    - We can use this to enforce the LIFO rule of the stack abstract data type

# Array based stack implementation

Instance variables required

To implement the array based stack, we will need the following variables

- An array of `ints` to store our items
  - `private int[] values;`
- An integer variable to remember where the top item is
  - `private int top;`
- It is also a good idea to store the size of the array as a variable, this way we can stop a user from inserting too many items and crashing the program
  - `private int maxSize;`
- All variables must be declared `private` so there values cannot be accessed or changed from outside this class

# Array based stack implementation

## Implementation Strategy

We must decide how we will change the variables to make sure that all items are inserted in the correct place.

- The initial value of the variable top is 0
- Every time we push a new item we will insert it into index top in the array values
  - After inserting the item we need to increment top, so that the next item is inserted in the next place in the array
- Every time we pop an item from the stack, we return the value in index top - 1
  - After returning the item we need to decrement top, so that the same item is not returned again

# Push Pseudocode

```
Algorithm push(i):
  Input: An integer to be stored in the stack

if(top < maxSize) then
    values[top] ← i
    top ← top + 1
```

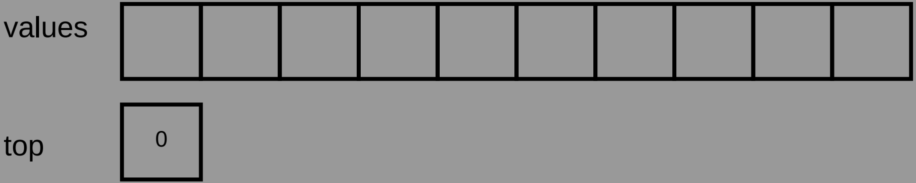
# Pop Pseudocode

```
Algorithm pop():
  Output: The integer value stored in the top
   of the stack

i ← values[top - 1]
top ← top     1
return i
```

# Array Based Stack Example

- Operation:



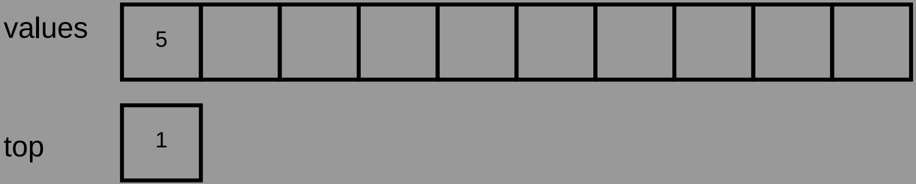
values

top

0

# Array Based Stack Example

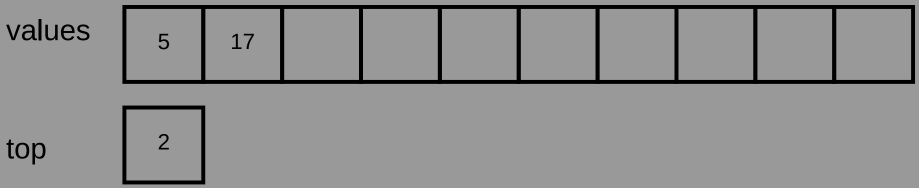
- Operation: `push(5)`



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| values | 5 | | | | | | | | | |
| top | 1 | | | | | | | | | |

# Array Based Stack Example

- Operation: `push(17)`

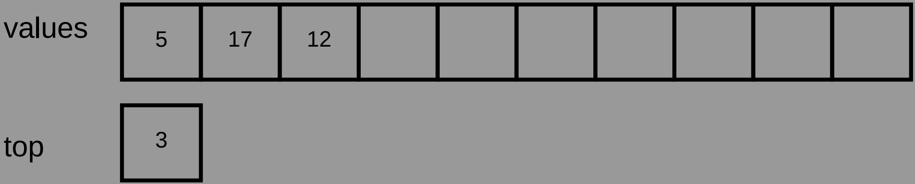


values 5 17

top 2

# Array Based Stack Example

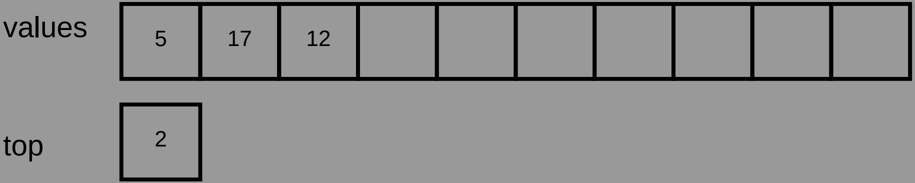
- Operation: `push(12)`



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| values | 5 | 17 | 12 | | | | | | | |
| top | 3 | | | | | | | | | |

# Array Based Stack Example

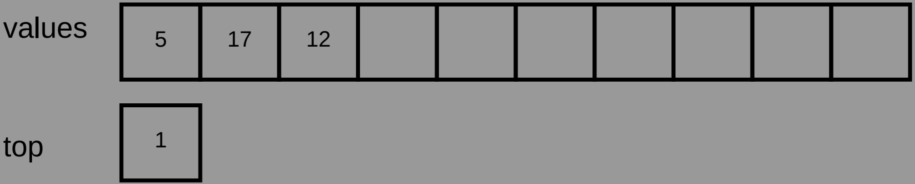
- Operation: `pop()`



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| values | 5 | 17 | 12 | | | | | | | |
| top | 2 | | | | | | | | | |

# Array Based Stack Example

- Operation: `pop()`



| values | 5 | 17 | 12 |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|

| top | 1 |
|---|---|

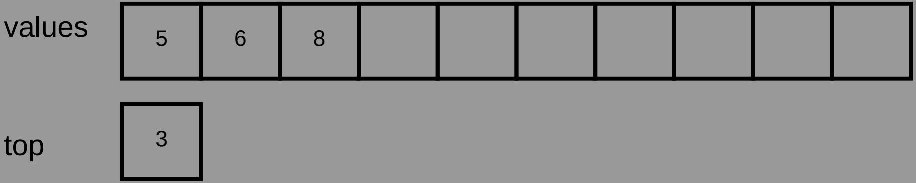
# Array Based Stack Example

- Operation: `push(6)`

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| values | 5 | 6 | 12 | | | | | | | |
| top | 2 | | | | | | | | | |

# Array Based Stack Example

- Operation: `push(8)`

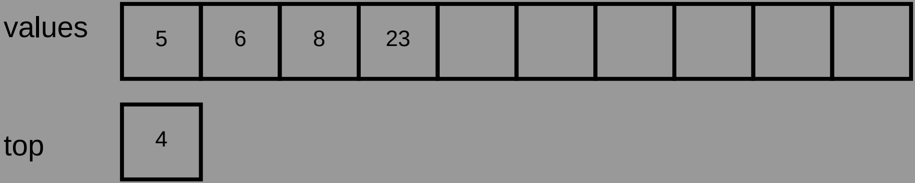


| values | 5 | 6 | 8 |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|

| top | 3 |
|---|---|

# Array Based Stack Example

- Operation: `push(23)`



|  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| values | 5 | 6 | 8 | 23 |  |  |  |  |  |  |
| top | 4 |  |  |  |  |  |  |  |  |  |

# Array Based Stack Example

In our example we saw how the data is stored in the array as it is pushed and popped

- There are some points that we should note:
    - Items are not actually removed when we call `pop`
    - The item will remain in the array, but we cannot access it
    - The item is only ever removed when another item is inserted in the same index

# Array Based Stack Implementation

Before we can implement the array based stack, we should think about what steps and decisions have to be made

- What size will the internal array be?
  - It is easier to have the user decide this by passing a parameter to the constructor
- What has to be defined in the constructor?
  - We need to create the array
  - We need to set the value of `top` to 0
  - We need to set the `maxSize` value

# Array Based Stack Implementation

```java
public class ArrayStack implements Stack {
  private int maxSize;
  private int[] values;
  private int top;

  public ArrayStack(int size) {
    maxSize = size;
    values = new int[size];
    top = 0;
  }
  public int size() {
    return top;
  }
  public boolean isEmpty() {
    return top == 0;
  }
```

# Array Based Stack Implementation

```
  public int top() {
    return values[top - 1];
  }
  public void push(int i) {
    if (top < maxSize) {
      values[top] = i;
      top = top + 1;
    }
  }
  public int pop() {
    int i = values[top - 1];
    top = top - 1;
    return i;
  }
}
```

# Table of Contents

1. Abstract Data Types

2. Stack Abstract Data Type

3. Array Based Implementation

4. Link Based Stack Implementation

# Link Based Stack Implementation

To implement the stack abstract data type without using an array first we need a class

- We will create a class called `LinkStack`
- This version of the stack abstract data type does not make use of an array to store data
- Each item is stored separately in an object based on a class called `Node`
- Each `Node` stores a reference to the `Node` that comes after it in the stack
  - This `Node` is usually called `next`
- The `LinkStack` class only keeps a reference to the top `Node`

# Link Based Stack Implementation

Node class

The `Node` class is very simple, it only needs to contain two instance variables

- A variable to store the data
  - `private int data;`
- A variable to store a reference to the next `Node` in the stack
  - `private Node next;`

# Node Class Implementation

```java
public class Node {
   int data;
   Node next;

   public Node(int i){
      data = i;
   }
}
```

# Link Based Stack Implementation

Instance variables required

To implement the link based stack, we need the following variables

- A reference to the node that is the top of the stack
  - `Node top;`
- An int to count the number of items in the stack
  - `int size;`

# Link Based Stack Implementation

Implementation strategy

We must decide how we will change the variables to make sure that all items are inserted in the correct place

- The initial value of the variable `size` is 0
- Every time we `push` a new item we need to do the following:
  - Create a new `Node` object `n`, containing the item to be inserted
  - We change the reference of the `next` variable in `n` to the value of `top`
  - We change the reference `top` to the value of `n`
- Every time we `pop` an item from the stack, we need to do the following:
  - Copy the data stored in the `top` variable to a temporary variable
  - Change the reference `top`, to the value of the next variable inside `top`

# Push Pseudocode

```
Algorithm push(i):
   Input: An integer to be stored in the stack.

N ← new Node(i)
N.next ← top
top ← N
size ← size + 1
```

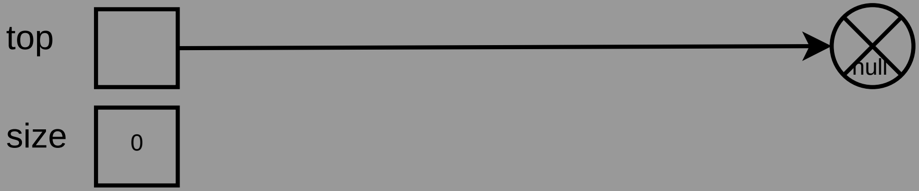
# Pop Pseudocode

```
Algorithm pop():
  Output: The integer that is on  the top of the stack

i ←  top.element
top ←  top.next
size ←  size - 1
return i
```

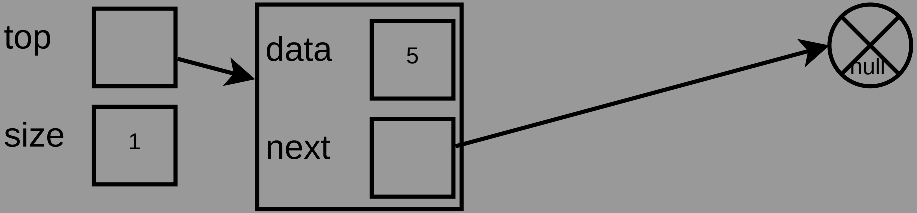
# Link Based Stack Example

- Operation:



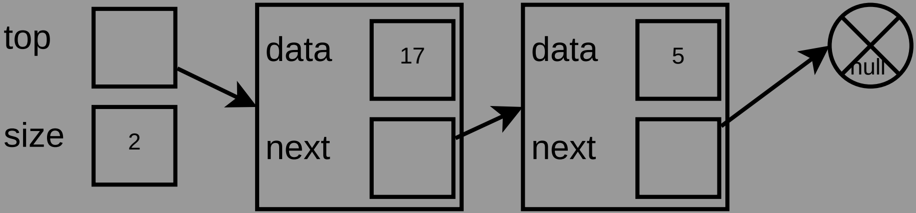
# Link Based Stack Example

- Operation: `push(5)`



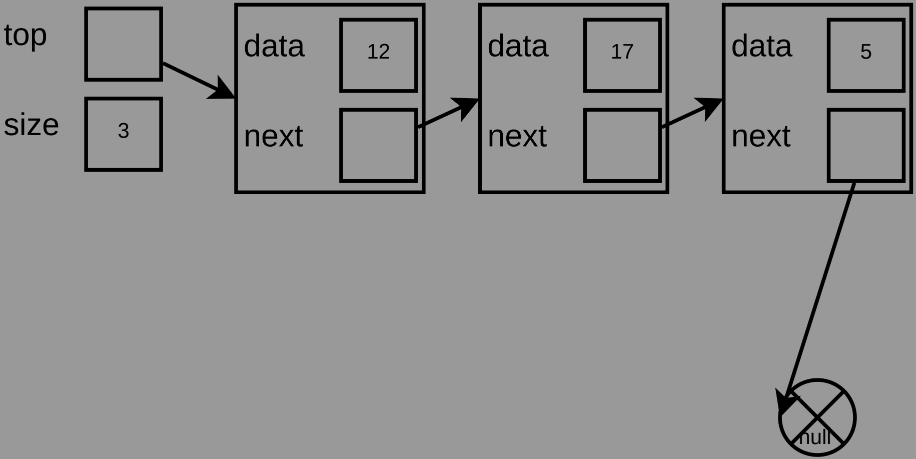
# Link Based Stack Example

- Operation: `push(17)`



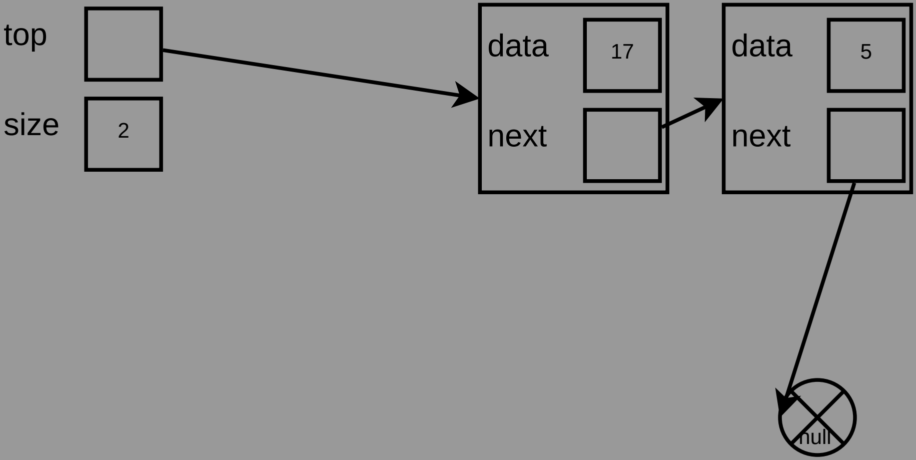
# Link Based Stack Example

- Operation: `push(12)`



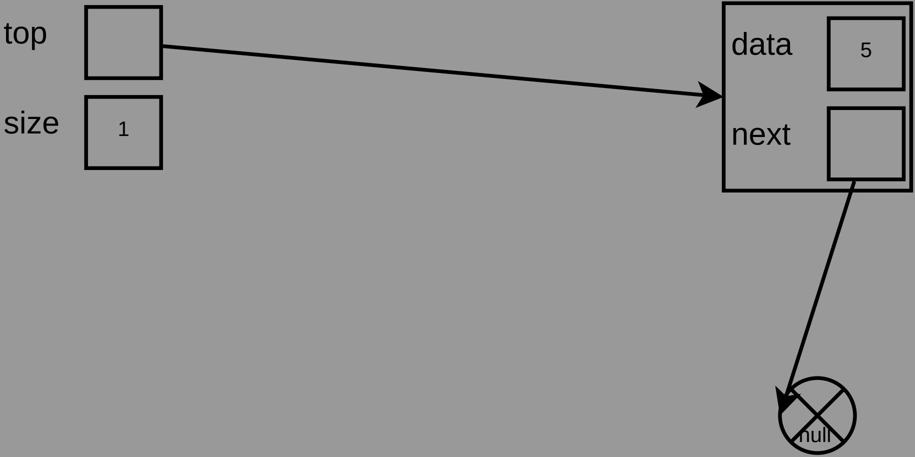
# Link Based Stack Example

- Operation: `pop()`



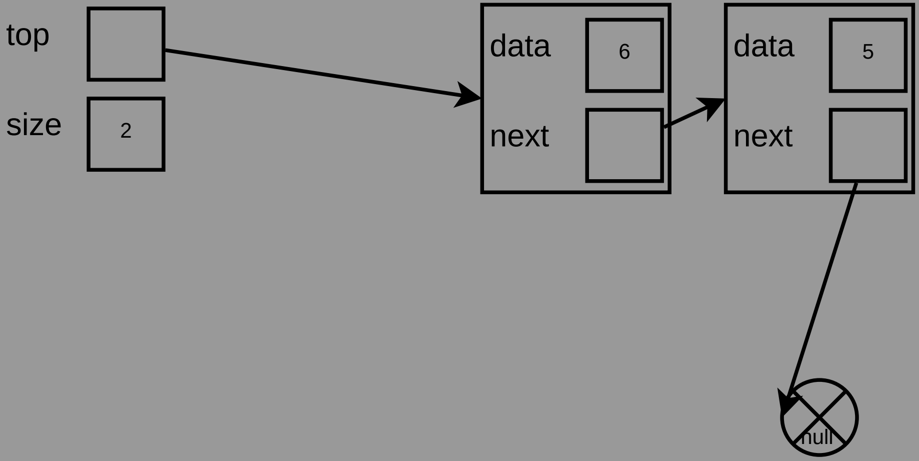
# Link Based Stack Example

- Operation: `pop()`



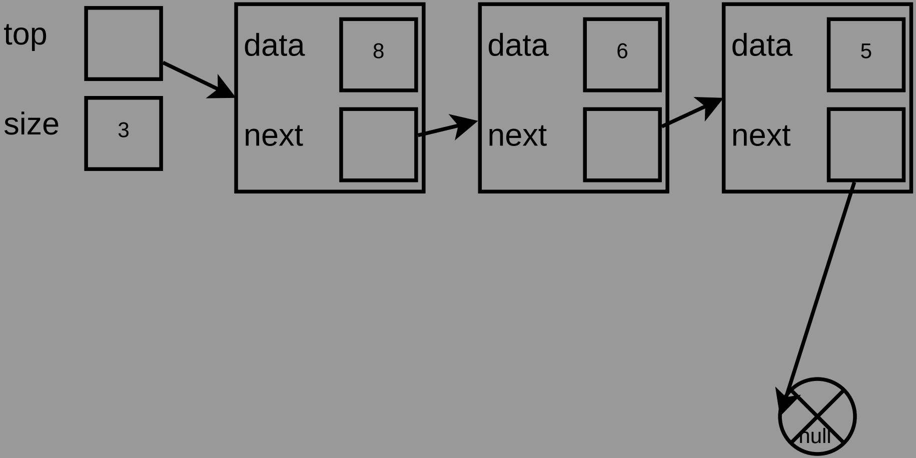
# Link Based Stack Example

- Operation: `push(6)`



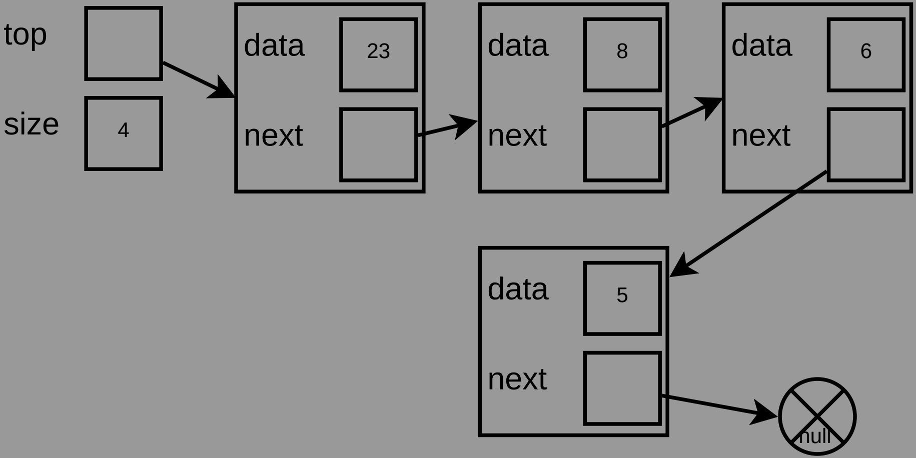
# Link Based Stack Example

- Operation: `push(8)`



# Link Based Stack Example

- Operation: `push(23)`



# Further Information and Review

If you wish to review the materials covered in this lecture or get further information, read the following sections in Data Structures and Algorithms textbook.

- 5.1 - Stacks