# CSCI 132:
# Basic Data Structures and Algorithms

Lessons Learned so far + Intro to Stacks

Reese Pearsall

Fall 2023

# Announcements

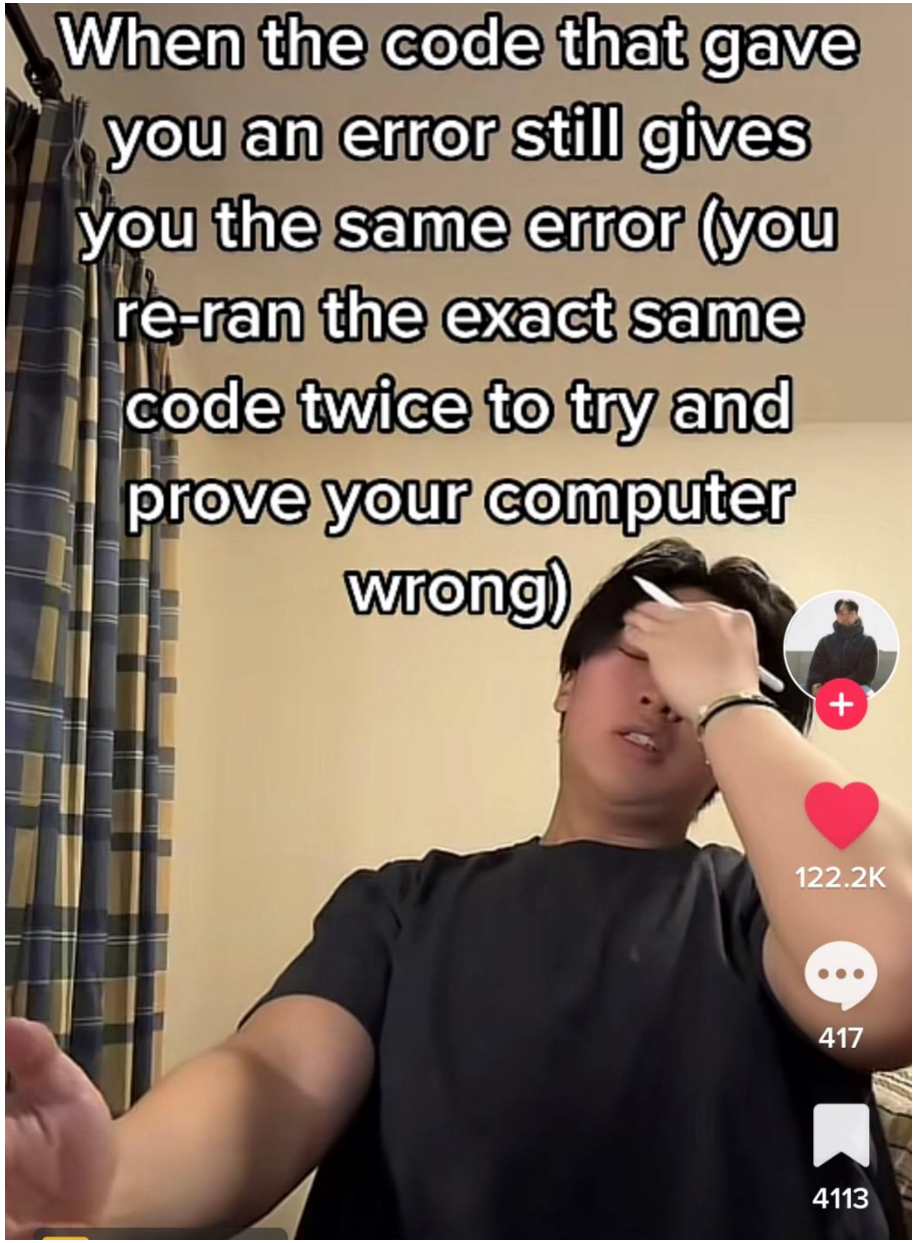
Come get your midterm exam

- Exam average was in the 80s
- Don’t stress if you didn’t do well
- Make sure I calculated your score correctly

Lab 7 and Program 3 will be posted very soon

Class Registration

→ CSCI 232
→ CS 145
→ CSCI 215
→ CSCI 246



# Big-O

Big-O notation is a way to describe the running-time/time complexity of an algorithm regarding the number of operations that are executed in the algorithm (in relation to some input **`n`**)

- Focus on worst-case scenario, and how the algorithm scales as **`n`** gets really big

Big-O

Big-O notation is a way to describe the running-time/time complexity of an algorithm regarding the number of operations that are executed in the algorithm (in relation to some input **`n`**)

- Focus on worst-case scenario, and how the algorithm scales as **`n`** gets really big

A very powerful computer and a very weak computer running the same algorithm will both execute the same number of operations (the speed at which they execute these operations will be different)

**Takeaway**: the asymptotic running time (the big-o running time) will be the same for each computer

# Big-O

Big-O notation is a way to describe the running-time/time complexity of an algorithm regarding the number of operations that are executed in the algorithm (in relation to some input **`n`**)

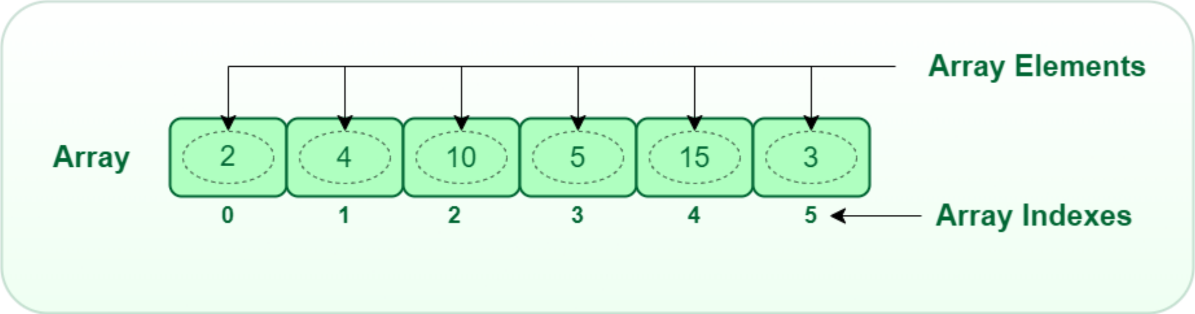
- Focus on worst-case scenario, and how the algorithm scales as **`n`** gets really big

To find the total running time of an algorithm, we calculate the running-time of each operation in the algorithm and then add everything together

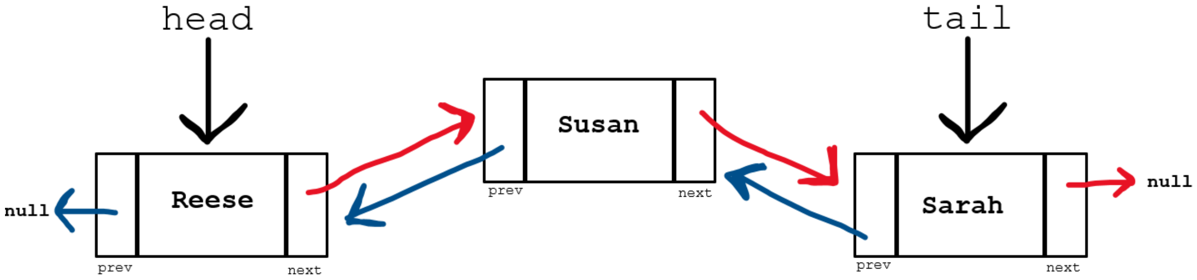
- In Big-O, we can drop non-dominant factors and multiplicative constants (coefficients)

O(n) + O(n) + O(n): Total running time = O(3n) $\in$ **O(n)**

Data Structures so far:

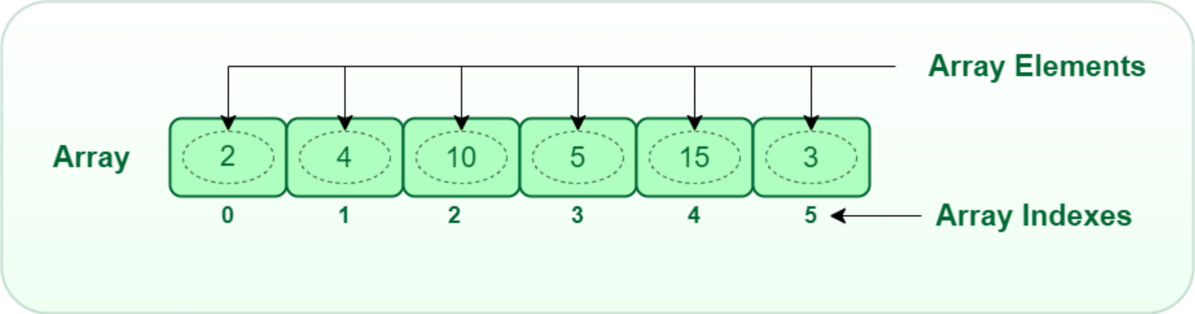


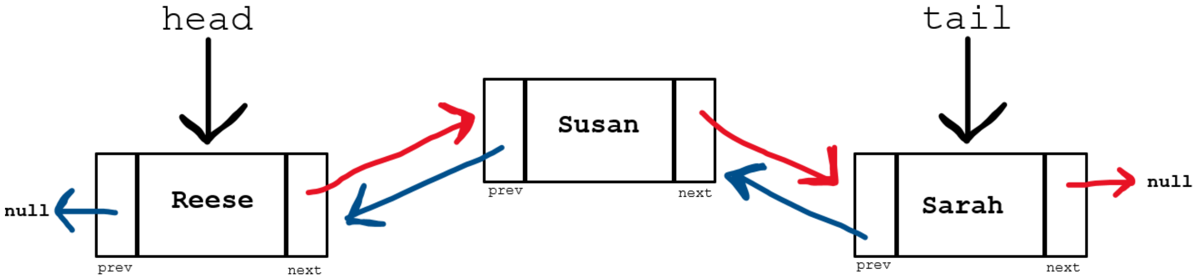

ArrayLists (Arrays)




Linked Lists

# Data Structures so far:



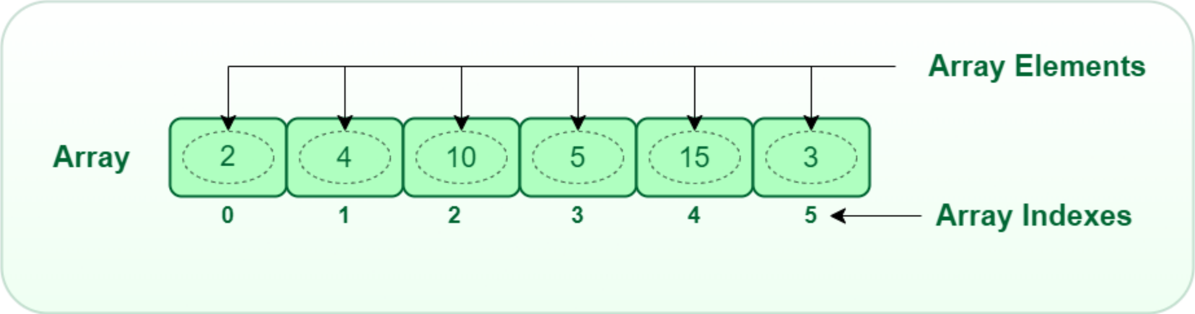

ArrayLists (Arrays)

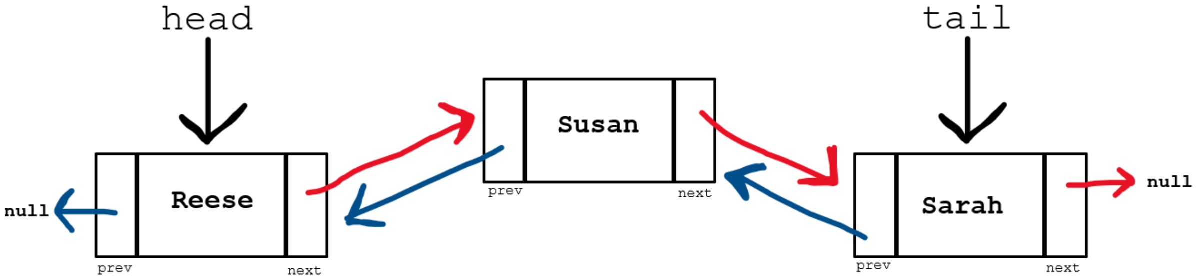
Linked Lists

Can hold one data type

~~Can hold multiple data types~~

# Data Structures so far:




## ArrayLists (Arrays)

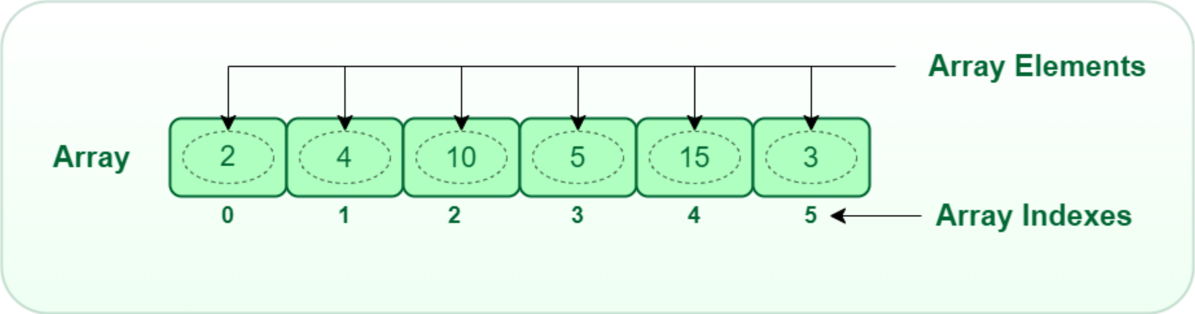
## Linked Lists

Can hold one data type

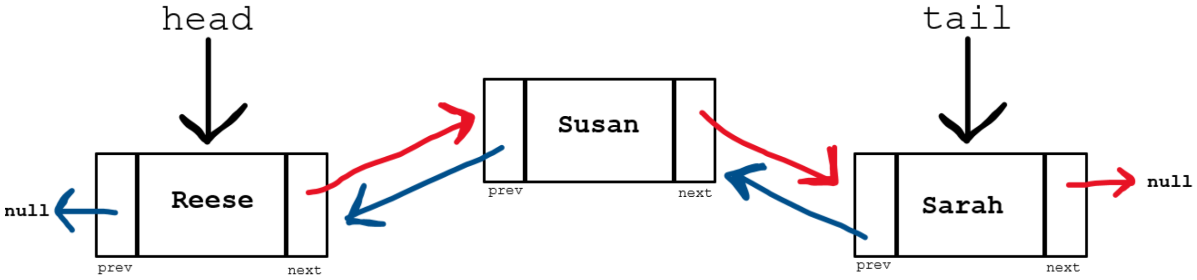
- Can also store objects, which allow for multiple data types

*Nodes* in the linked list can hold multiple data types

# Data Structures so far:




## ArrayLists (Arrays)




## Linked Lists

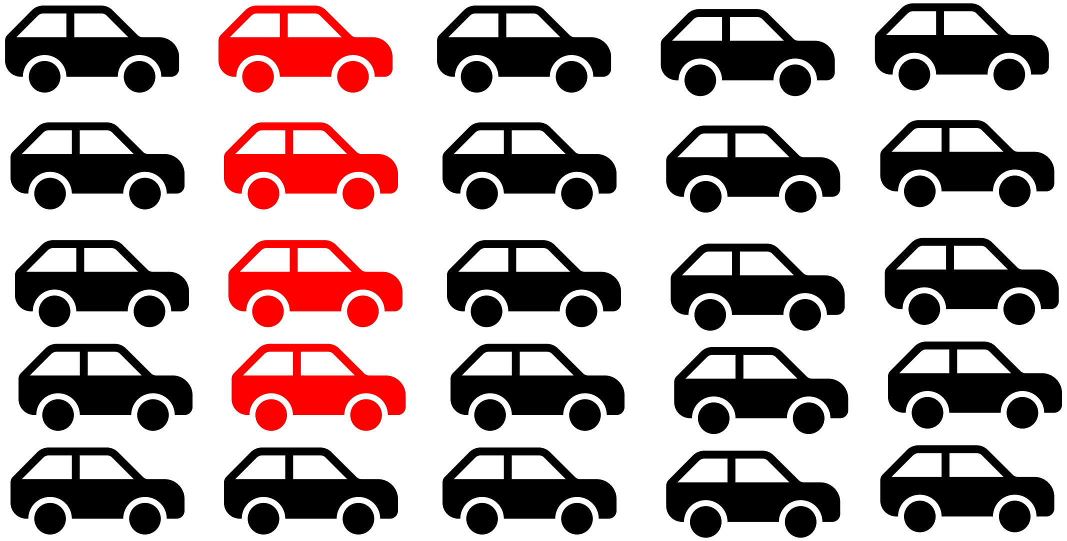
Can hold one data type

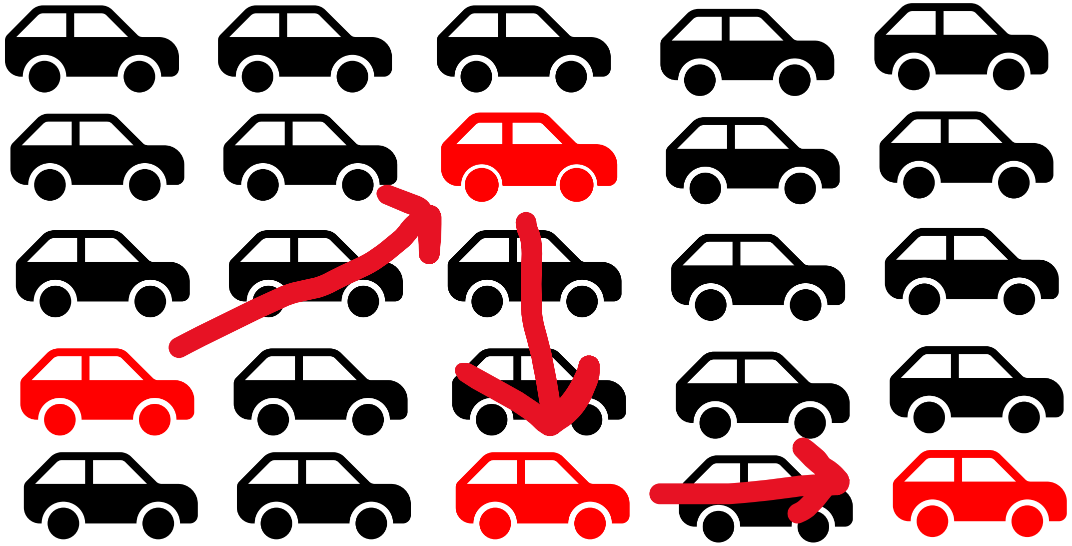
- Can also store objects, which allow for multiple data types

*Nodes* in the linked list can hold multiple data types

Entire array is stored at a **contiguous** spot in memory

Linked list nodes are stored at **non-contiguous** spots in memory





Can hold one data type

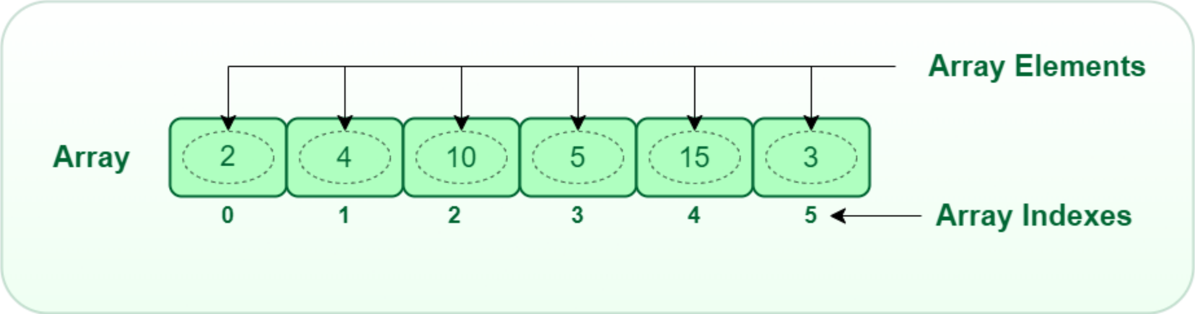
- Can also store objects, which allow for multiple data types

*Nodes* in the linked list can hold multiple data types

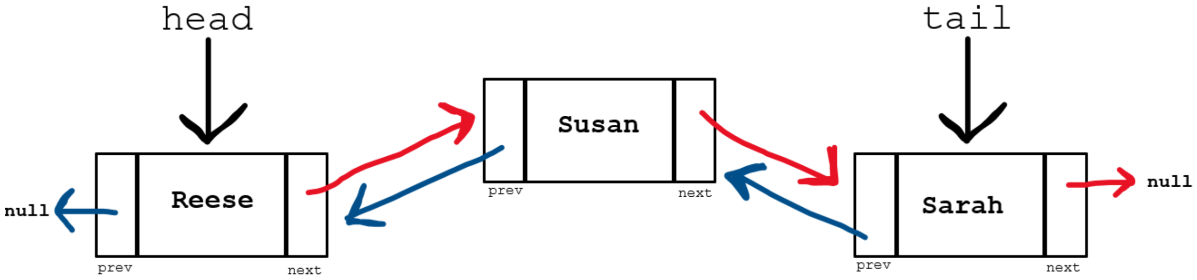
Entire array is stored at a **contiguous** spot in memory

Linked list nodes are stored at **non-contiguous** spots in memory

## Data Structures so far:




ArrayLists (Arrays)




Linked Lists

Can hold one data type

- Can also store objects, which allow for multiple data types

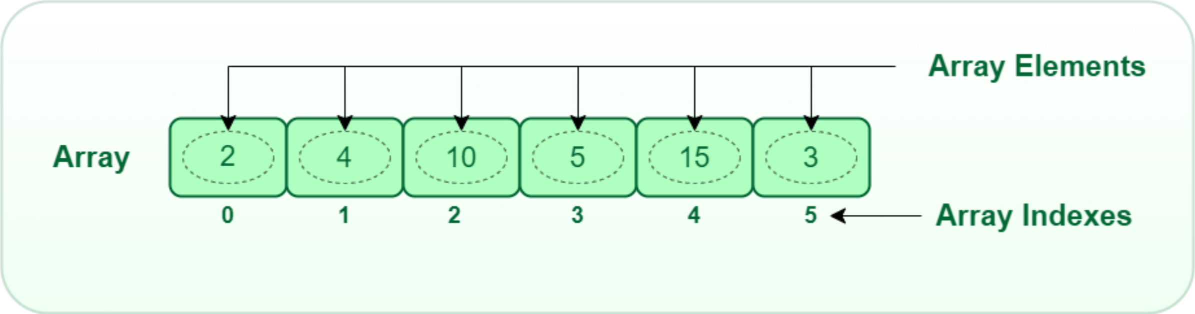
*Nodes* in the linked list can hold multiple data types

Entire array is stored at a **contiguous** spot in memory

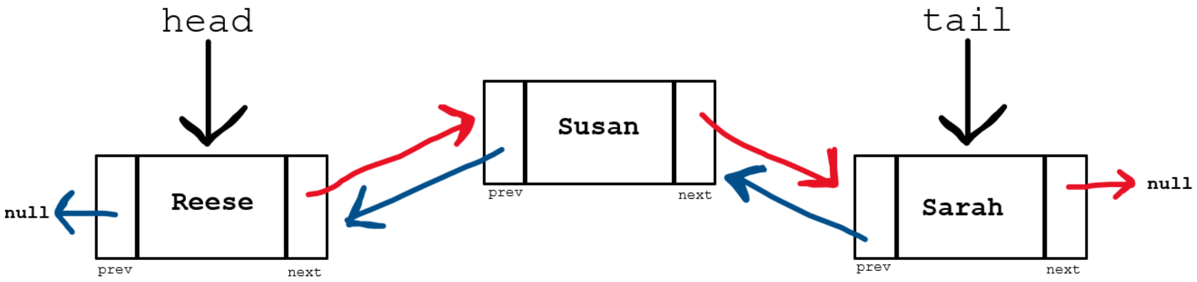
Linked list nodes are stored at **non-contiguous** spots in memory

*Traversing a linked list requires more work than traversing an array*

Data Structures so far:




ArrayLists (Arrays)



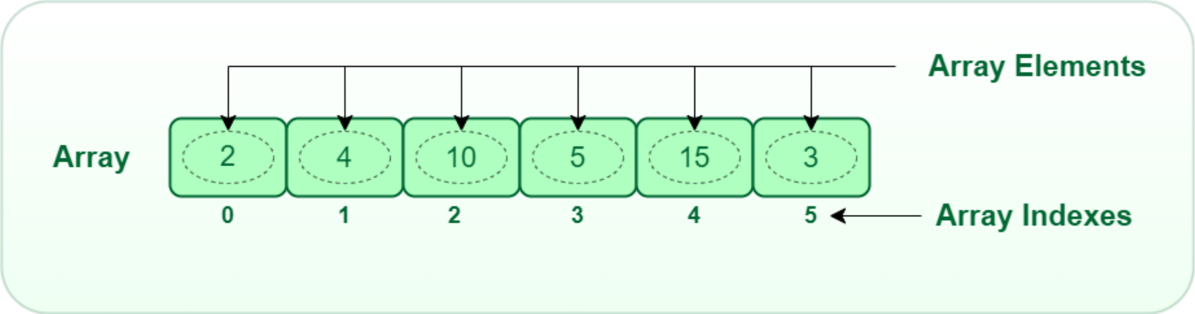

Linked Lists

Can add new elements to data structure (resizable)

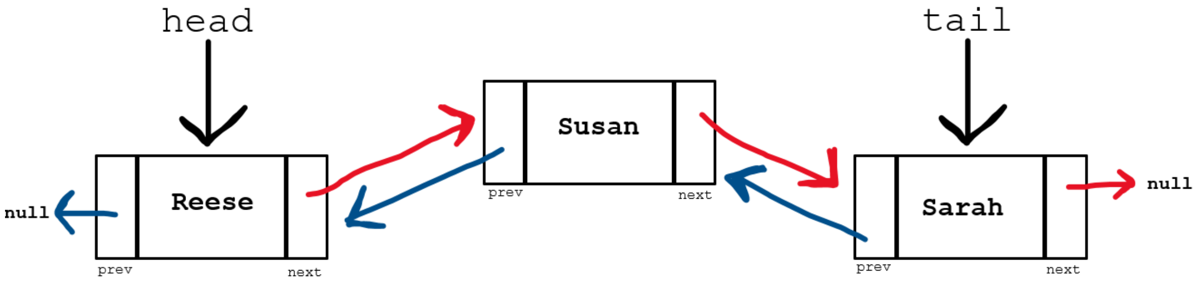
Can add new elements to data structure (resizable)

Both data structures can grow dynamically, and new elements can be added, but they way they add new elements is **drastically** different

# Data Structures so far:




## ArrayLists (Arrays)




## Linked Lists

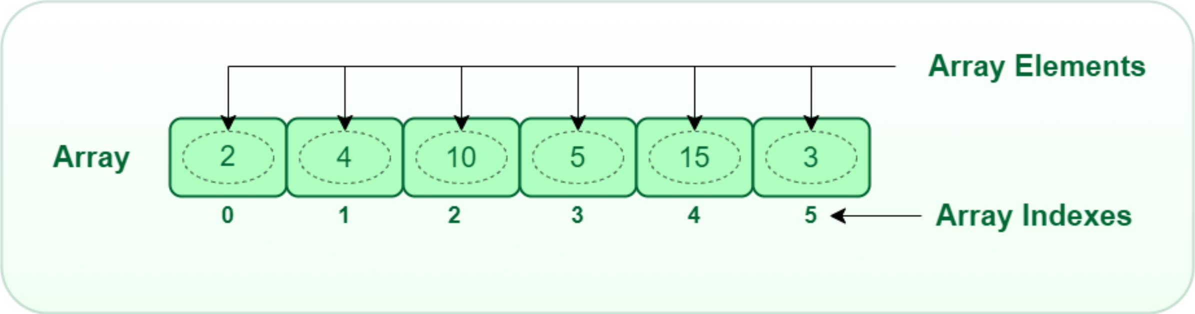
```java
int[] newArray = new int[myArray.length + 1];
for(int i = 0; i < myArray.length; i++) {
        newArray[i] = myArray[i];
}

int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

Create a brand-new array, copy everything over from old array

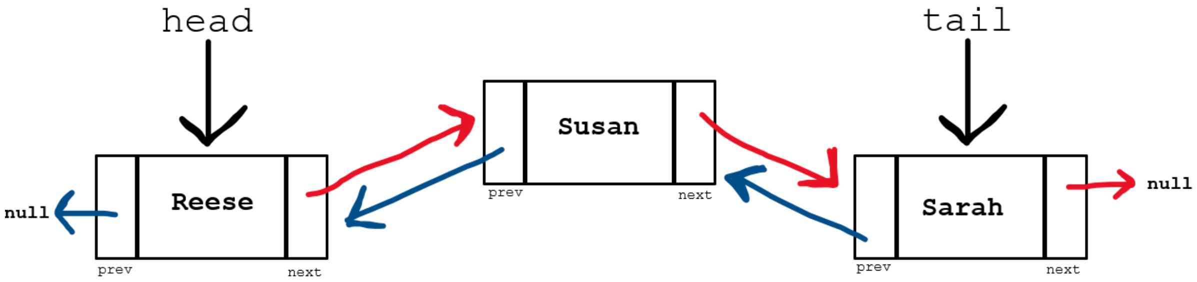
```java
public void addToFront(Node newNode) {
    if(head == null) {
        head = newNode;
    }
    else {
        newNode.setNext(head);
        head = newNode;
    }
}
```

Update pointers

# Data Structures so far:




ArrayLists (Arrays)




Linked Lists

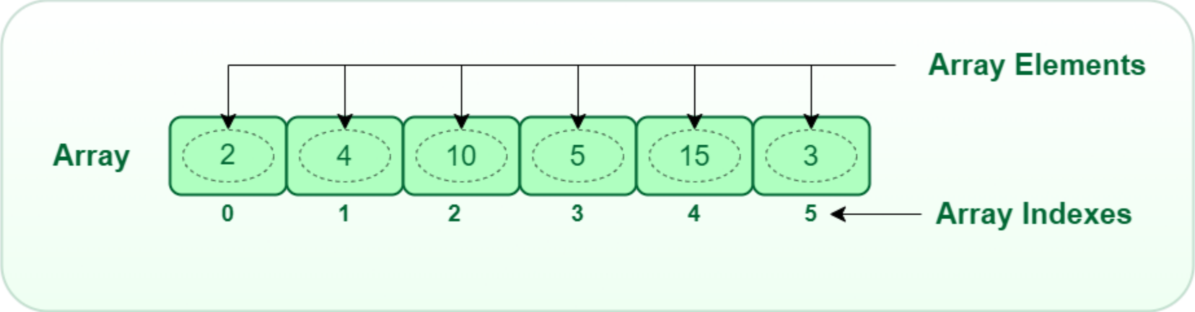
```
int[] newArray = new int[myArray.length + 1];
for(int i = 0; i < myArray.length; i++) {
        newArray[i] = myArray[i];
}

int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

Create a brand-new array, copy everything over from old array O(n)

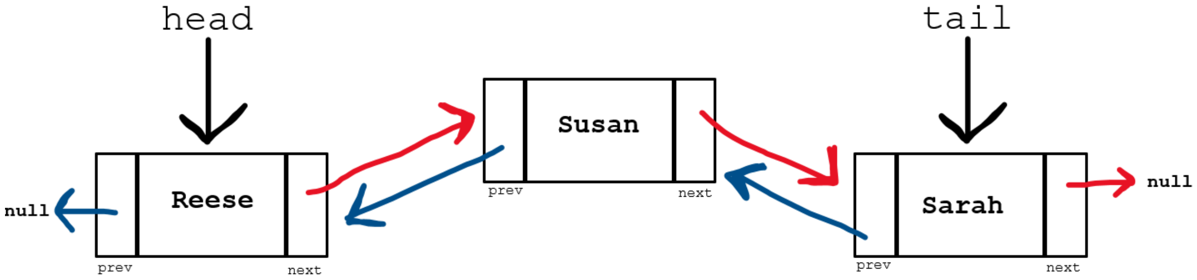
```
public void addToFront(Node newNode) {
    if(head == null) {
        head = newNode;
    }
    else {
        newNode.setNext(head);
        head = newNode;
    }
}
```

Update pointers O(1)

# Data Structures so far:




ArrayLists (Arrays)



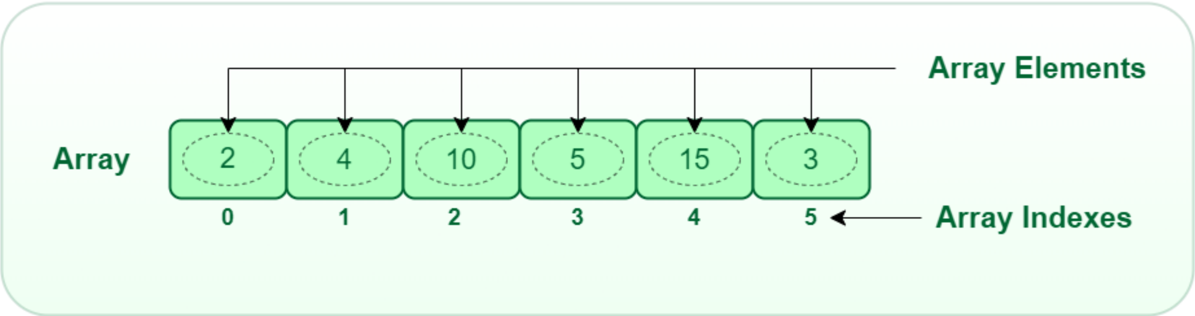

Linked Lists

```
int[] newArray = new int[myArray.length + 1];
for(int i = 0; i < myArray.length; i++) {
          newArray[i] = myArray[i];
}

int new_value = 4;
newArray[myArray.length] = new_value;
myArray = newArray;
```

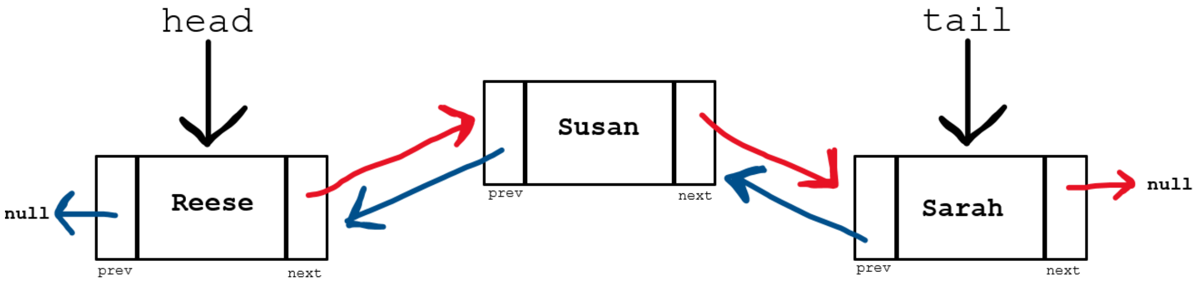
```
public void addToFront(Node newNode) {
    if(head == null) {
        head = newNode;
    }
    else {
        newNode.setNext(head);
        head = newNode;
    }
}
```

***Takeaway***: Adding a new element to an ArrayList requires much more work than adding a new element to a Linked List

# Data Structures so far:




ArrayLists (Arrays)



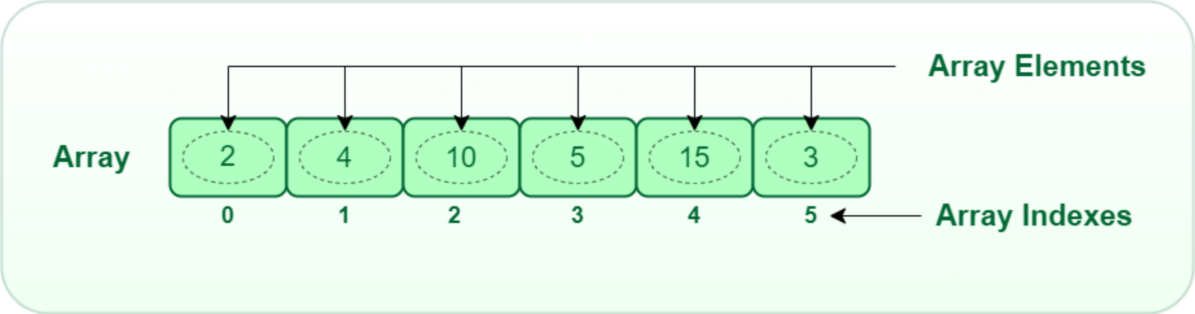

Linked Lists

Arrays are generally much easier to sort than Nodes in a Linked List

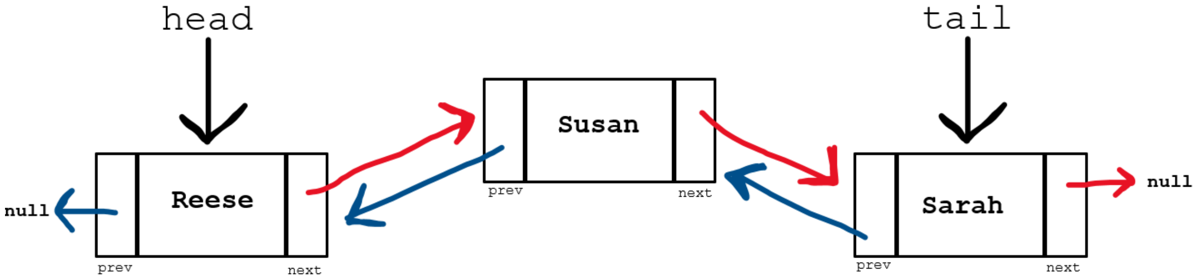
If you are constantly needing to add new elements to the data structure, using a Linked List requires much less work in the long run

Arrays are more memory efficient (adding is not very memory efficient though)

# Data Structures so far:




ArrayLists (Arrays)




Linked Lists

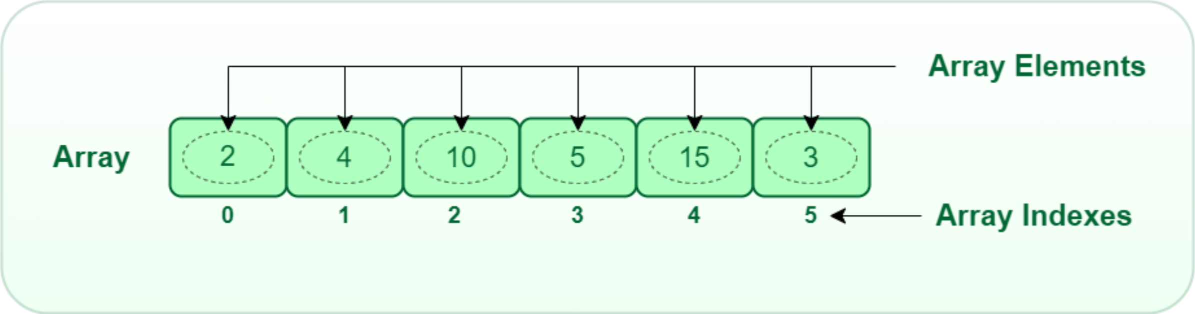
When to use each data structure?

It depends on ***how you are using your data*** and ***if you know how much data you have***

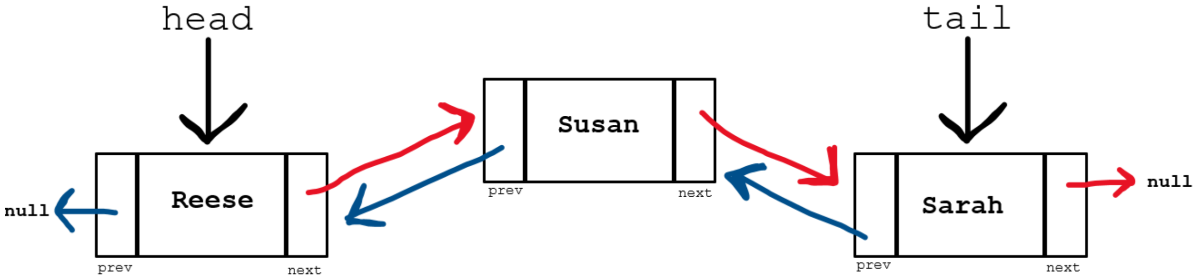
If you don't know how much data you need to store, or if you are constantly needing to add new elements to the data structure → **Linked Lists**

If you know how much data you need to store, and if you can add all your data at once → **Arrays/ArrayLists**

# Data Structures so far:




ArrayLists (Arrays)




Linked Lists

These two data structures are <u>implementations</u> of a **List** Abstract Data Type (ADT)

ADT is a class whose behavior is defined by a set of operations and how a user interacts with it.

A list data type must be able to **get** an element, **add** an element, **remove** an element, etc
→ How they do these operations is up to the subclass (LL and AL)

As programmers, we use handy methods that were written by other people that allows us to use these data structures

# The Linked List Class

We will no longer be writing our own Linked List class, instead we will now import the Java-provided Linked List Class

```java
import java.util.LinkedList;
```

We will no longer be writing our own Linked List class, instead we will now import the Java-provided Linked List Class

```java
import java.util.LinkedList;
```

```java
LinkedList<String> names = new LinkedList<String>();
```

The data type the linked list will be holding

Reference variable for LL

# The Linked List Class

https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Doubly-linked list implementation of the `List` and `Deque` interfaces. Implements all optional list operations, and permits all elements (including `null`).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

The **documentation** describe how the LinkedList class was implemented, and all the methods/operations we can do with the Linked List class

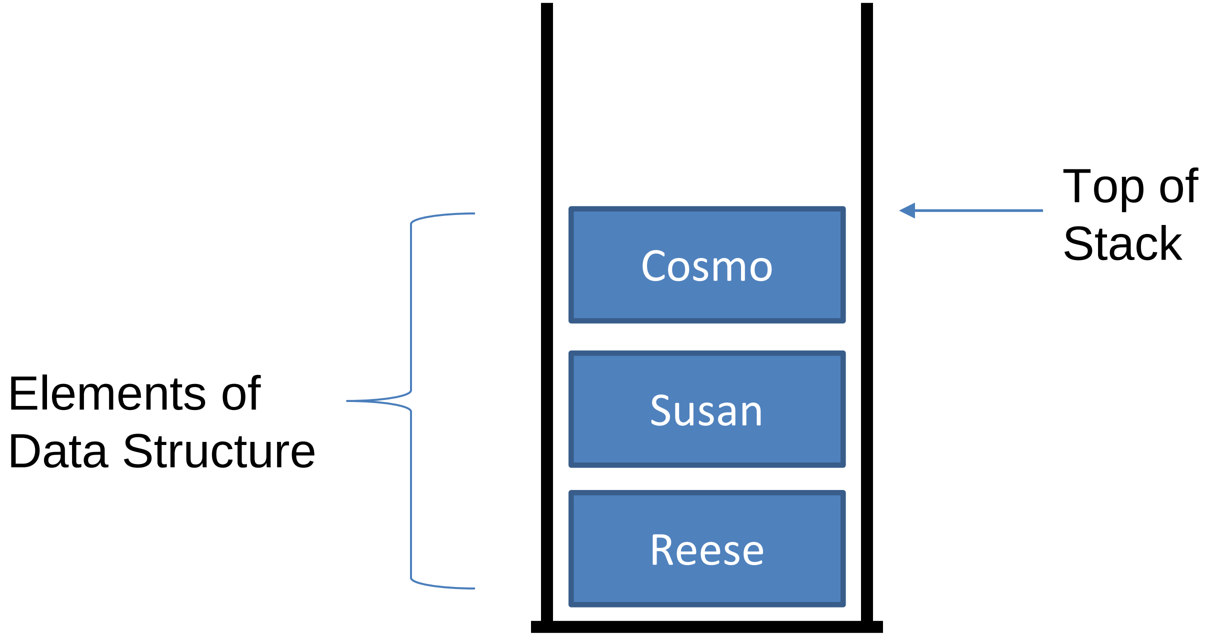
**Methods**

| Modifier and Type | Method and Description |
|---|---|
| `boolean` | `add(E e)`<br>Appends the specified element to the end of this list. |
| `void` | `add(int index, E element)`<br>Inserts the specified element at the specified position in this list. |
| `boolean` | `addAll(Collection<? extends E> c)`<br>Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the sp |
| `boolean` | `addAll(int index, Collection<? extends E> c)`<br>Inserts all of the elements in the specified collection into this list, starting at the specified position. |
| `void` | `addFirst(E e)`<br>Inserts the specified element at the beginning of this list. |
| `void` | `addLast(E e)`<br>Appends the specified element to the end of this list. |
| `void` | `clear()`<br>Removes all of the elements from this list. |
| `Object` | `clone()`<br>Returns a shallow copy of this `LinkedList`. |
| `boolean` | `contains(Object o)`<br>Returns `true` if this list contains the specified element. |
| `Iterator<E>` | `descendingIterator()`<br>Returns an iterator over the elements in this deque in reverse sequential order. |
| `E` | `element()`<br>Retrieves, but does not remove, the head (first element) of this list. |
| `E` | `get(int index)`<br>Returns the element at the specified position in this list. |
| `E` | `getFirst()`<br>Returns the first element in this list. |

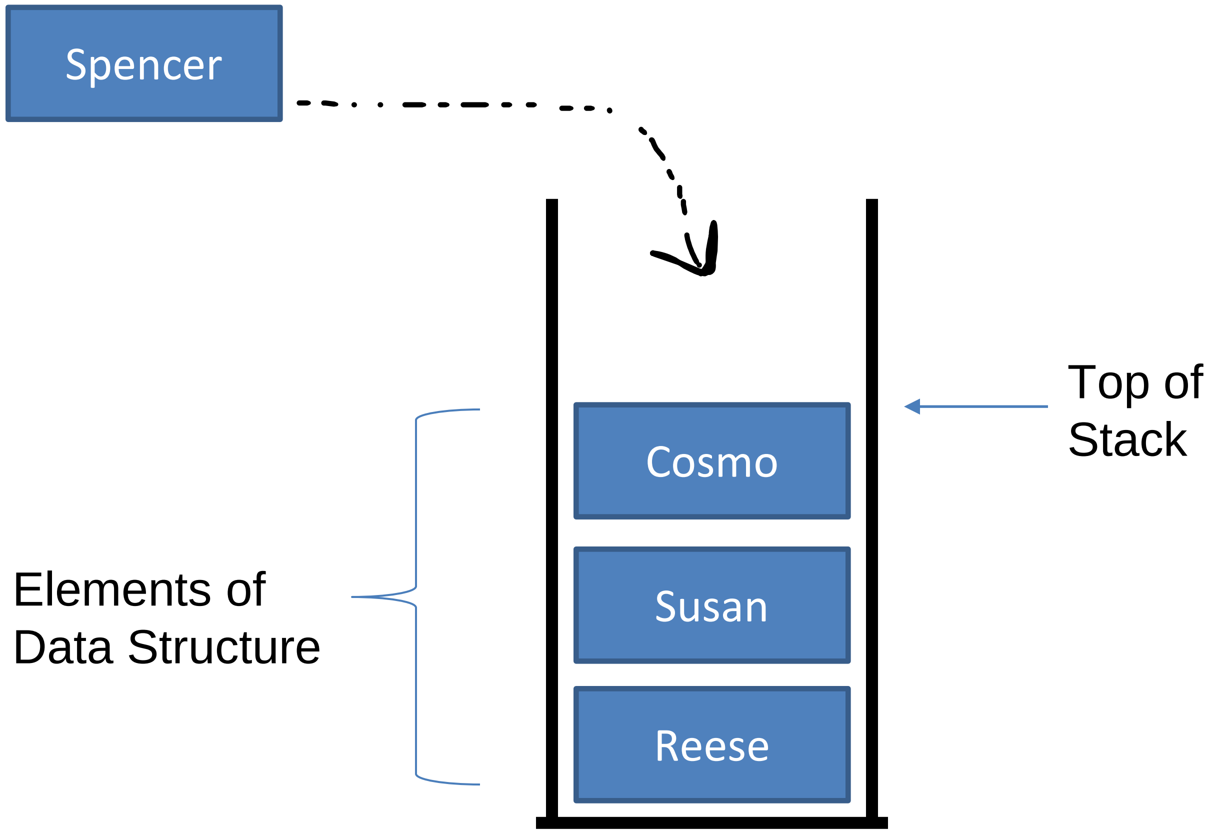
# The Linked List Class

```java
import java.util.LinkedList;

public class march20demo {

    public static void main(String[] args) {

        LinkedList<String> names = new LinkedList<String>();

        names.add("Reese");
        names.add("Spencer");
        names.add("Susan");

        System.out.println(names);


    }

}
```

A **stack** is a data structure that can hold data, however the way we interact with a stack is a little bit different



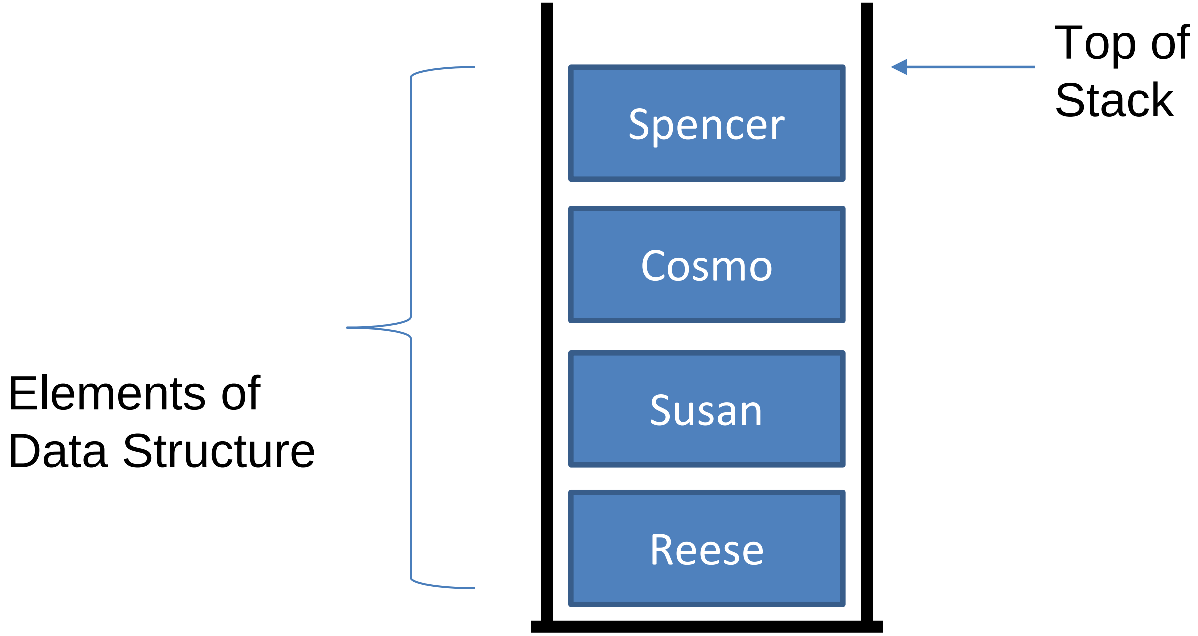
A **stack** is a data structure that can hold data, however the way we interact with a stack is a little bit different




When only interact with the top of the stack.

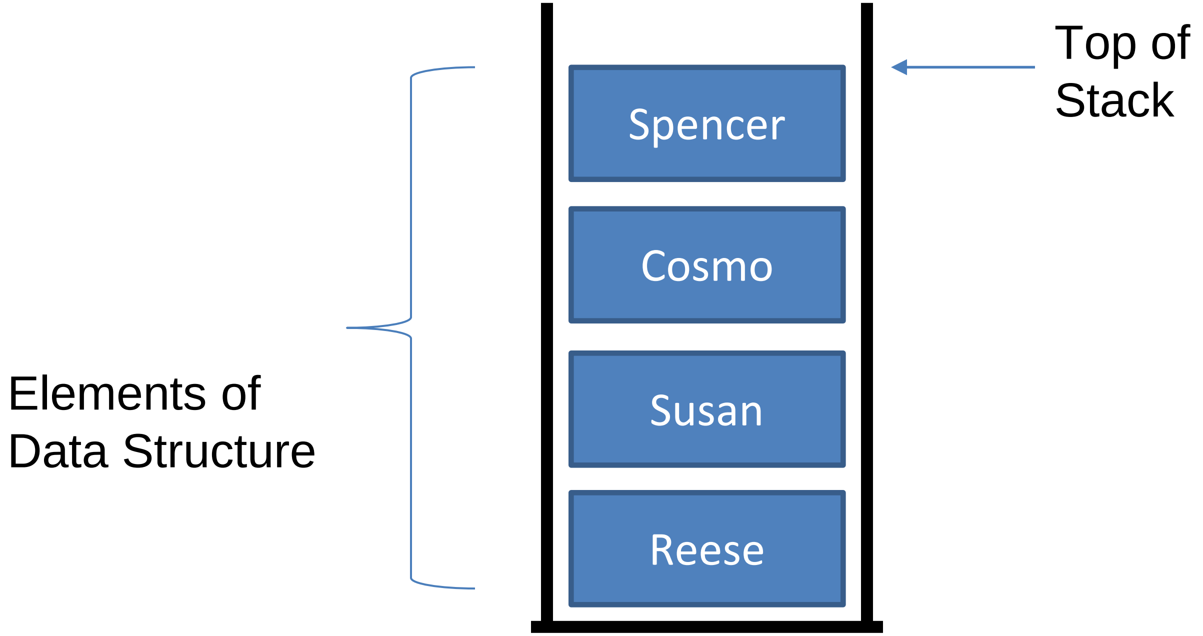
If we want to add a new element, we must put it on the top of the stack

A **stack** is a data structure that can hold data, however the way we interact with a stack is a little bit different



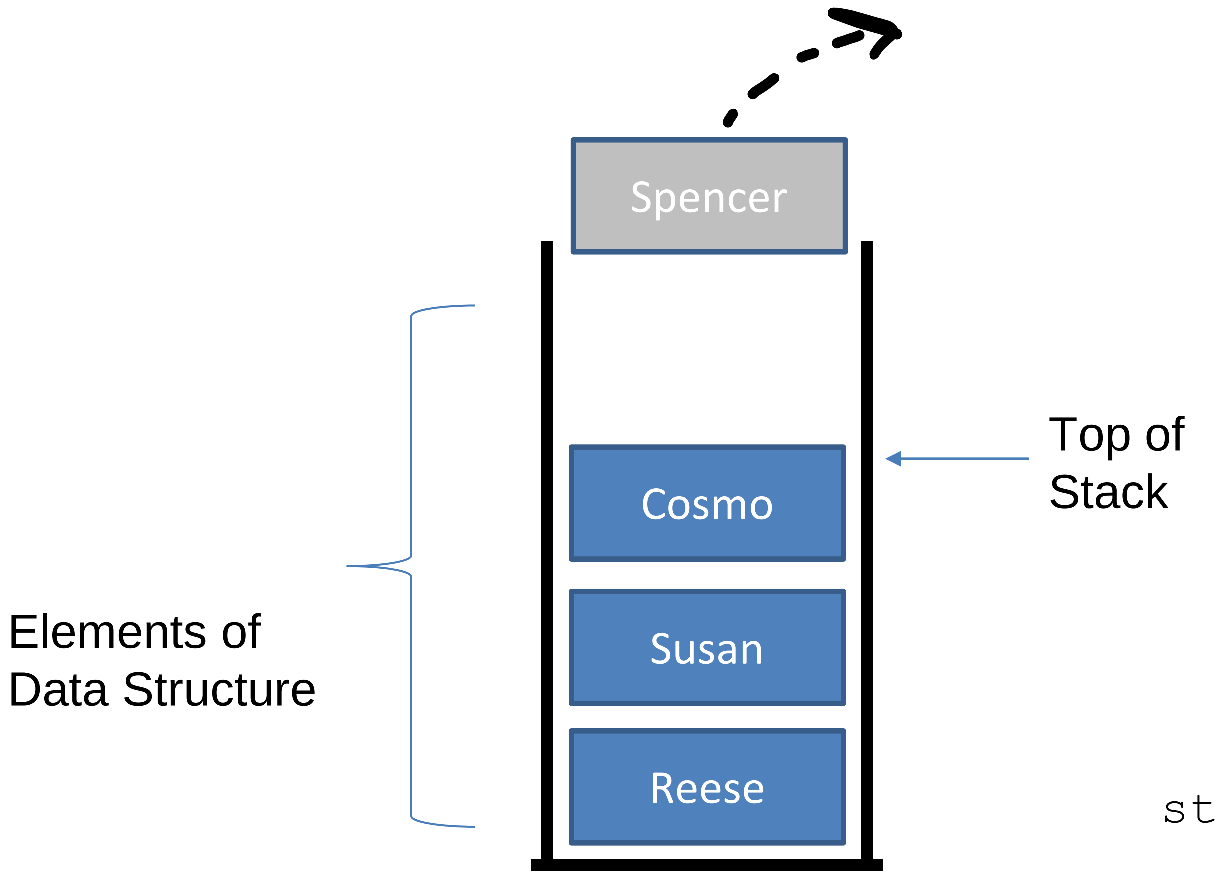

Adding something to a stack is known as the **push** operation

A **stack** is a data structure that can hold data, however the way we interact with a stack is a little bit different




If we want to remove something, we must always remove the element on the top of the stack

A **stack** is a data structure that can hold data, however the way we interact with a stack is a little bit different



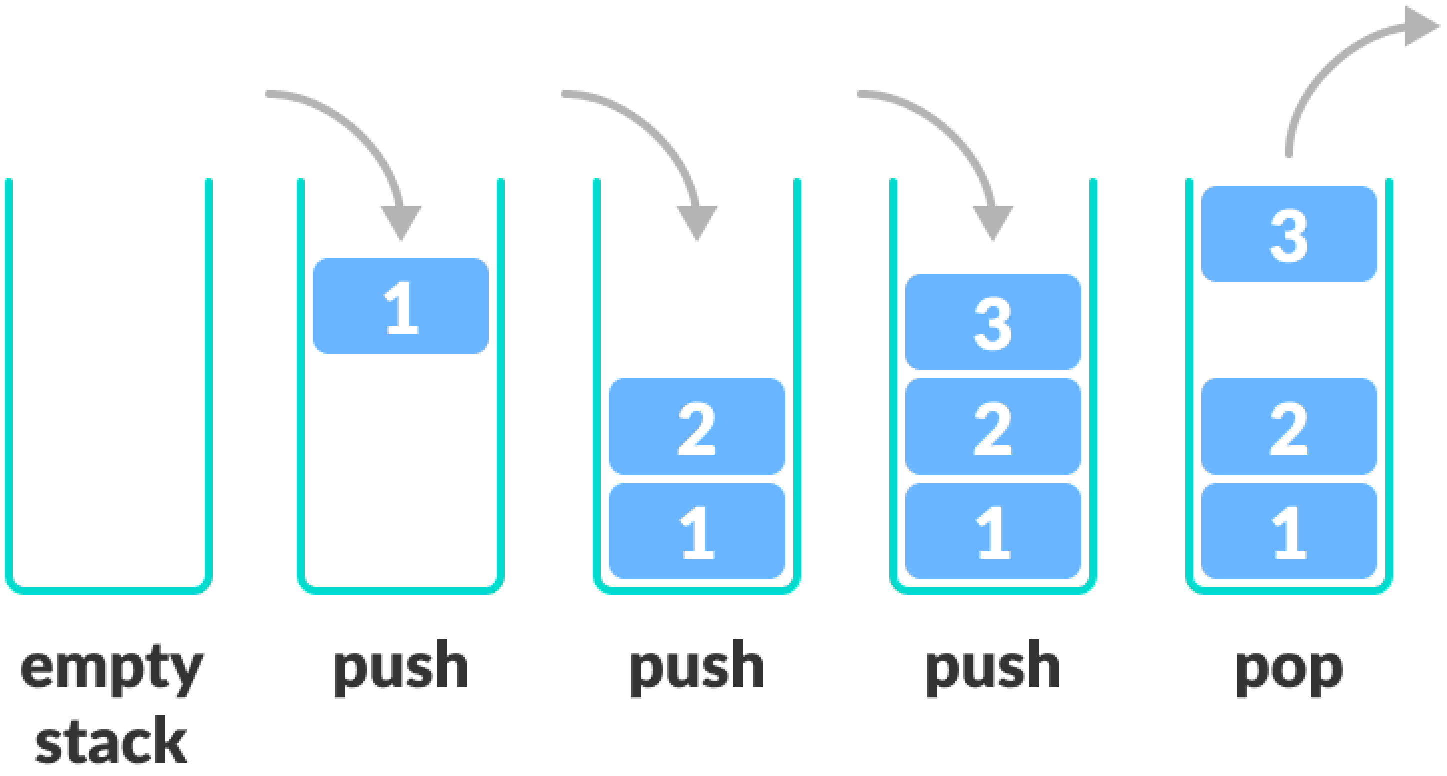

Removing an element is known as the **pop** operation

`stack.pop()` → Top node (spencer) is removed

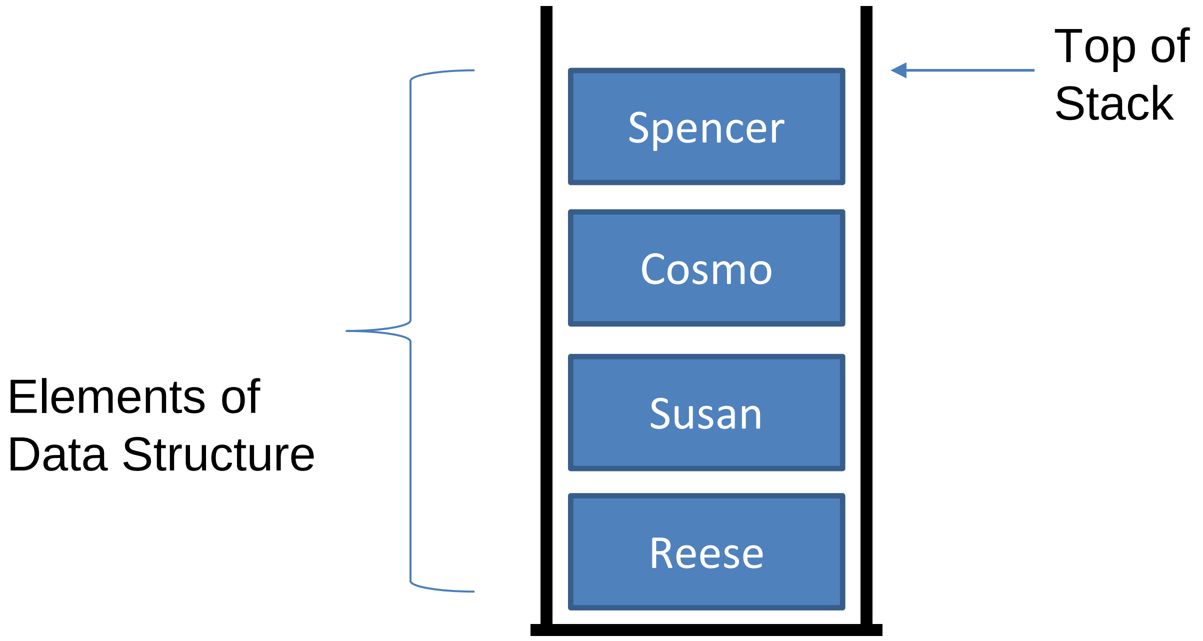
A **stack** is a data structure that can hold data, and follows the **last in first out (LIFO)** principle

We can:

- Add an element to the top of the stack (push)
- Remove the top element (pop)



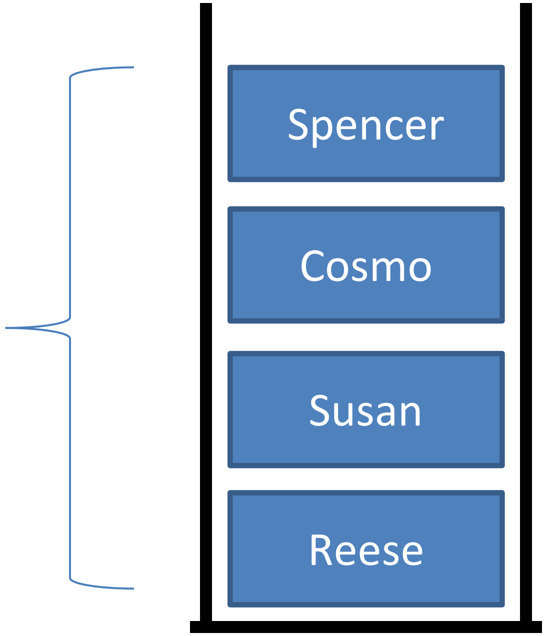

A **stack** is a data structure that can hold data, and follows the **last in first out (LIFO)** principle




Our stack data structure needs to keep track of a few things

1. Something to hold our stack elements

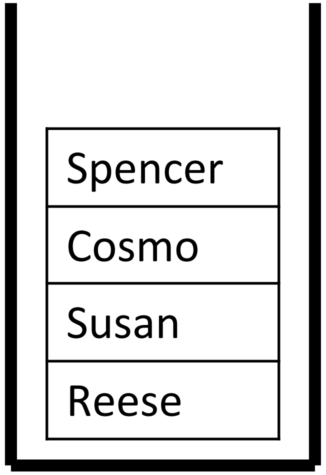
A **stack** is a data structure that can hold data, and follows the **last in first out (LIFO)** principle




Our stack data structure needs to keep track of a few things

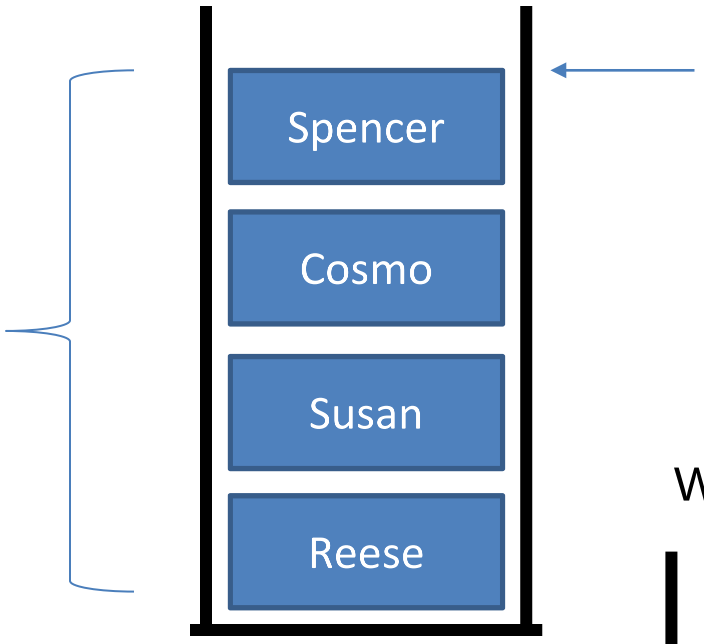
1. Something to hold our stack elements

We have a few options:




1. Array

A **stack** is a data structure that can hold data, and follows the **last in first out (LIFO)** principle



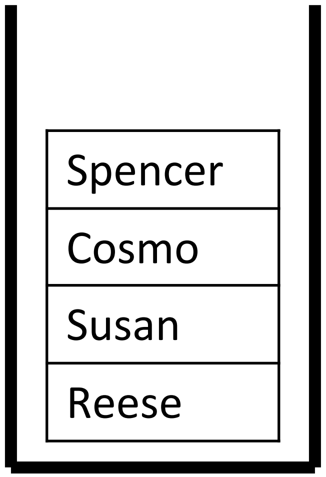

Top of Stack

Elements of Data Structure

Our stack data structure needs to keep track of a few things

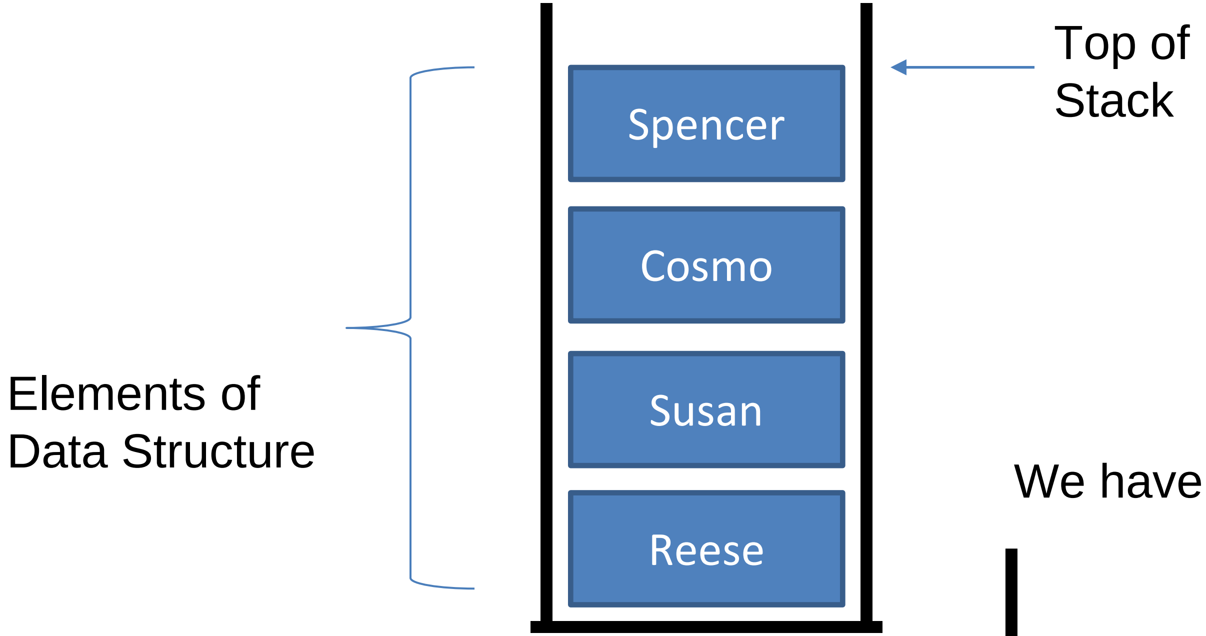
1. Something to hold our stack elements

We have a few options:




1. Array
2. ArrayList

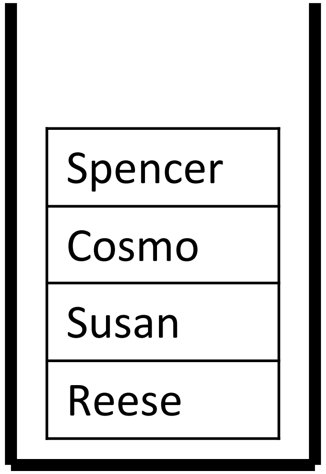
A **stack** is a data structure that can hold data, and follows the **last in first out (LIFO)** principle



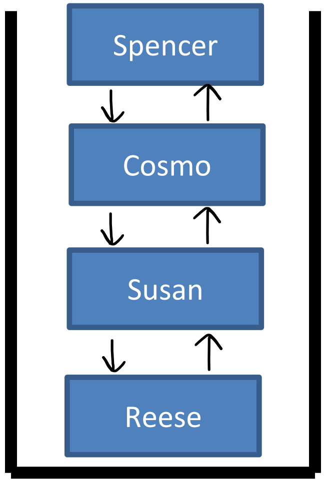

Our stack data structure needs to keep track of a few things

1. Something to hold our stack elements

We have a few options:

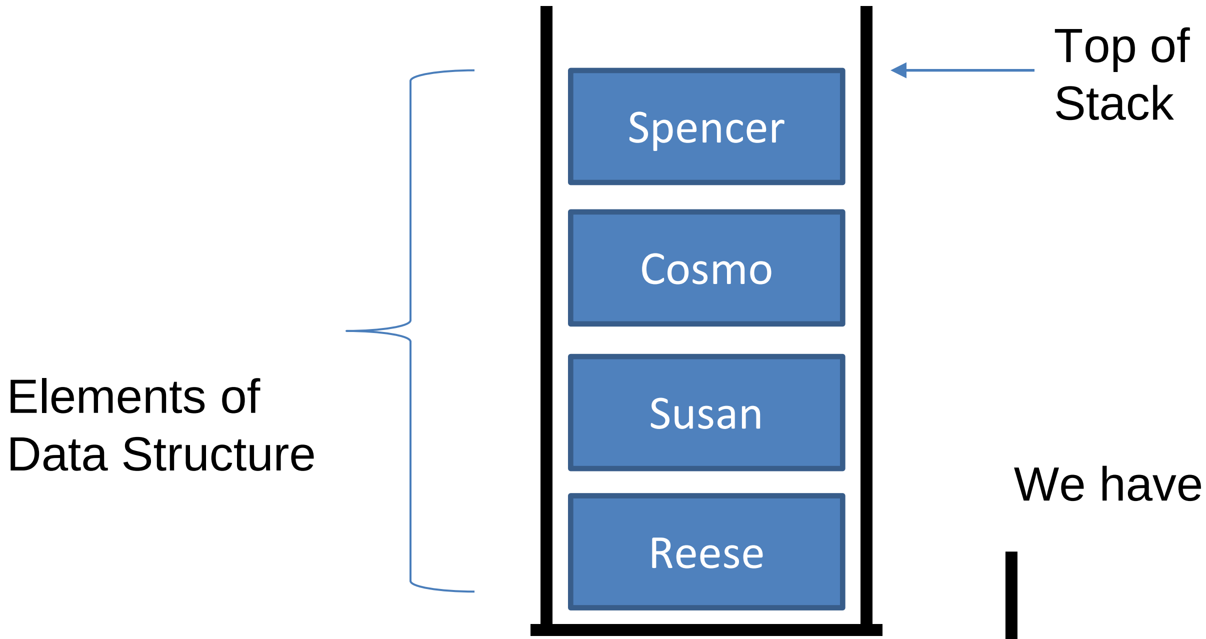



1. Array
2. ArrayList




3. Linked List

A **stack** is a data structure that can hold data, and follows the **last in first out (LIFO)** principle

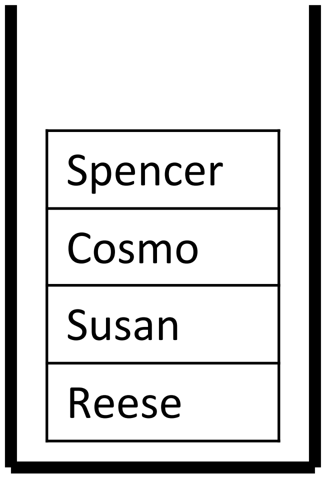



Our stack data structure needs to keep track of a few things

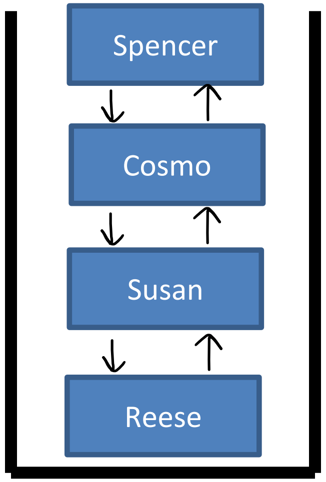
1. Something to hold our stack elements

Which should you pick?

We have a few options:

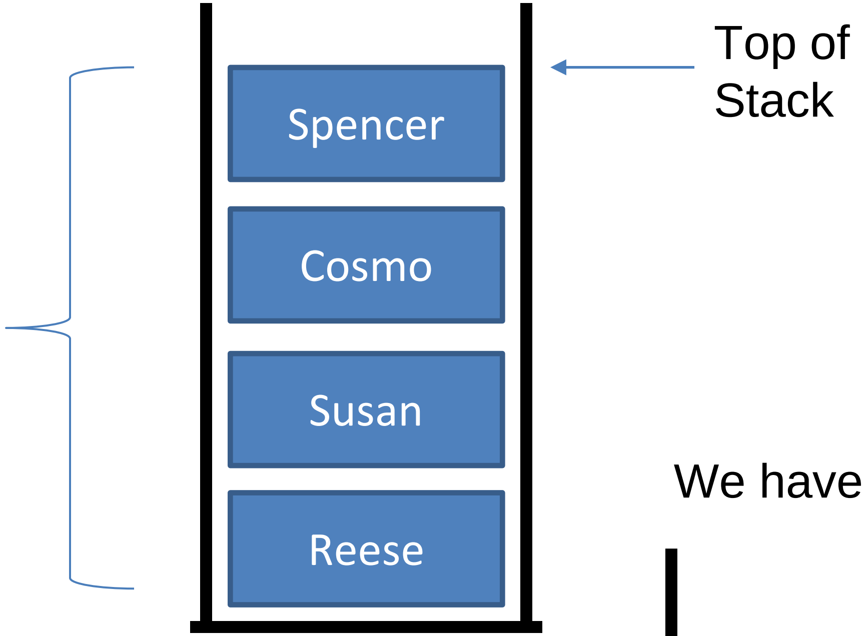



1. Array
2. ArrayList



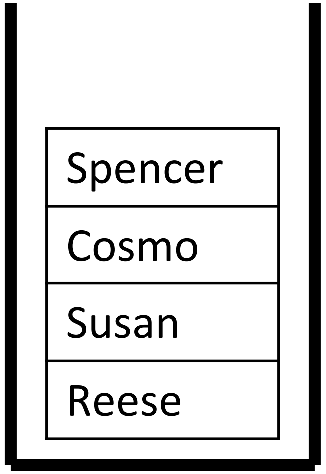

3. Linked List

A **stack** is a data structure that can hold data, and follows the **last in first out (LIFO)** principle

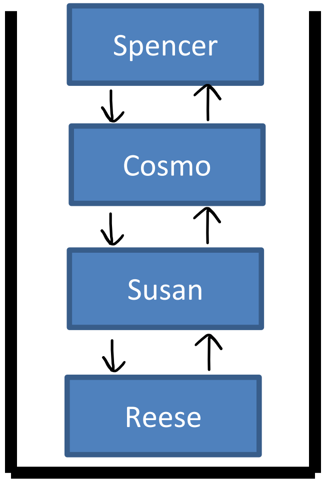



Our stack data structure needs to keep track of a few things
1. Something to hold our stack elements

We have a few options:




1. Array
2. ArrayList

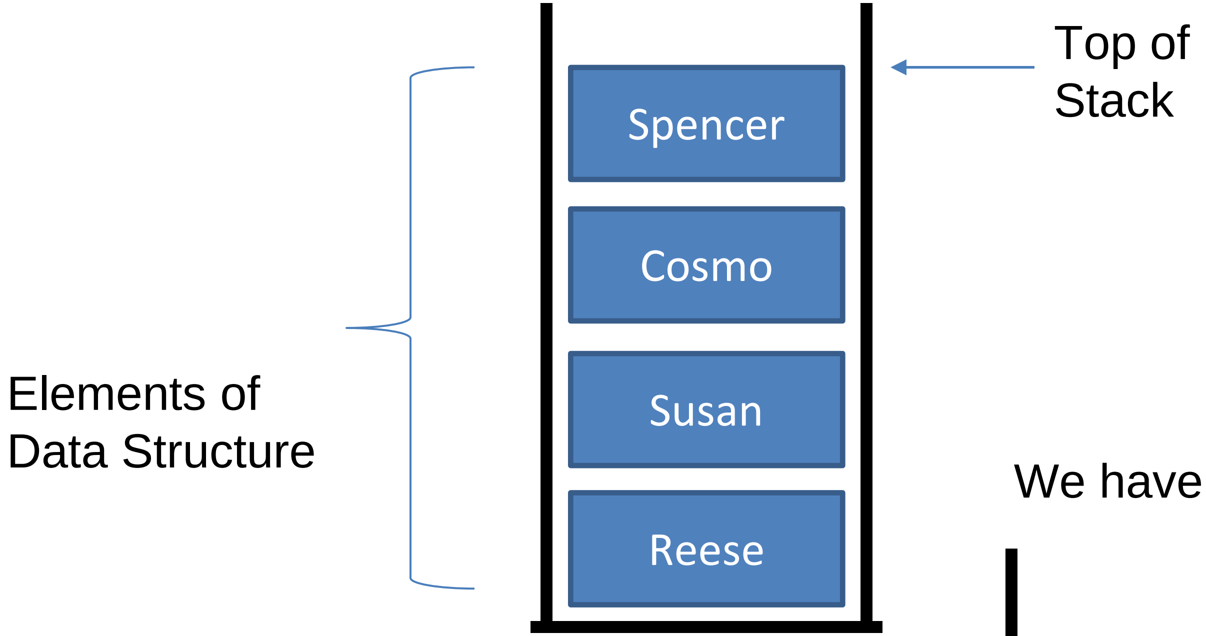



3. Linked List

Which should you pick?

- Depends on how you are using the stack

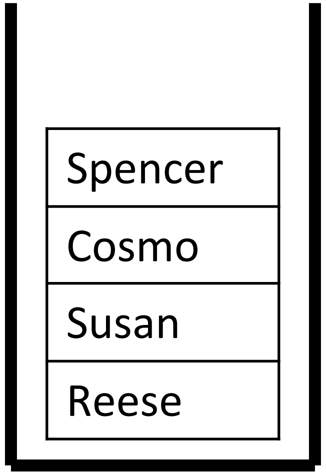
A **stack** is a data structure that can hold data, and follows the **last in first out (LIFO)** principle



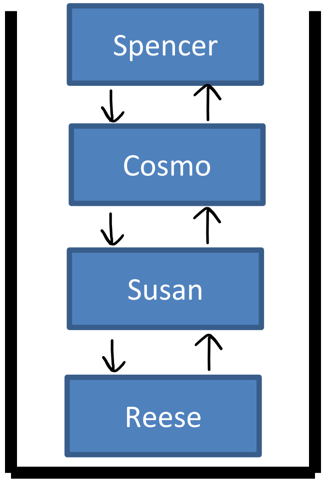

Our stack data structure needs to keep track of a few things

1. Something to hold our stack elements

We have a few options:




1. Array
2. ArrayList

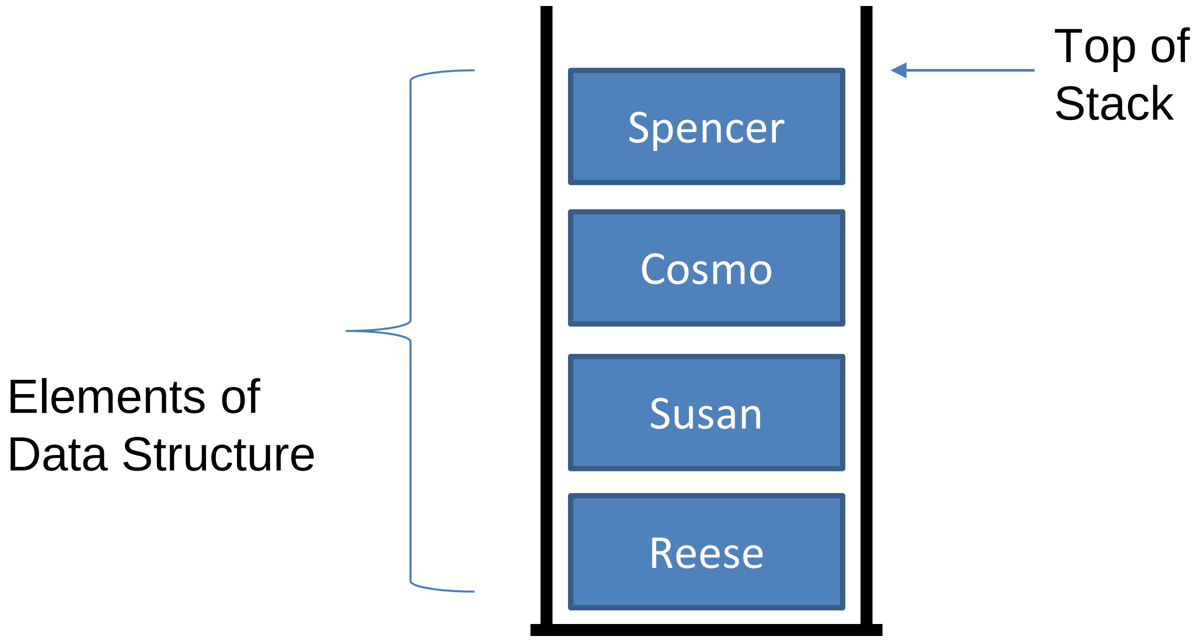



3. Linked List

## Which should you pick?

- If you know how big the stack needs to be
  → Array
- If you don't know how big the stack needs to be
  → Linked List

A **stack** is a data structure that can hold data, and follows the **last in first out (LIFO)** principle



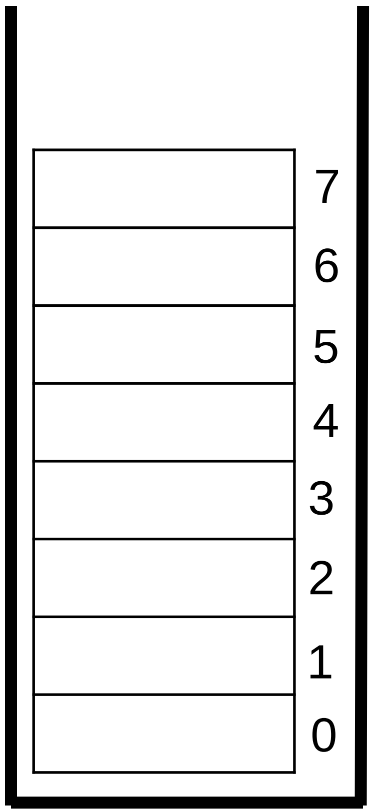

Our stack data structure needs to keep track of a few things

1. Something to hold our stack elements (Array/LinkedList)
2. Something that points the current top element of the stack
3. The size of the stack

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

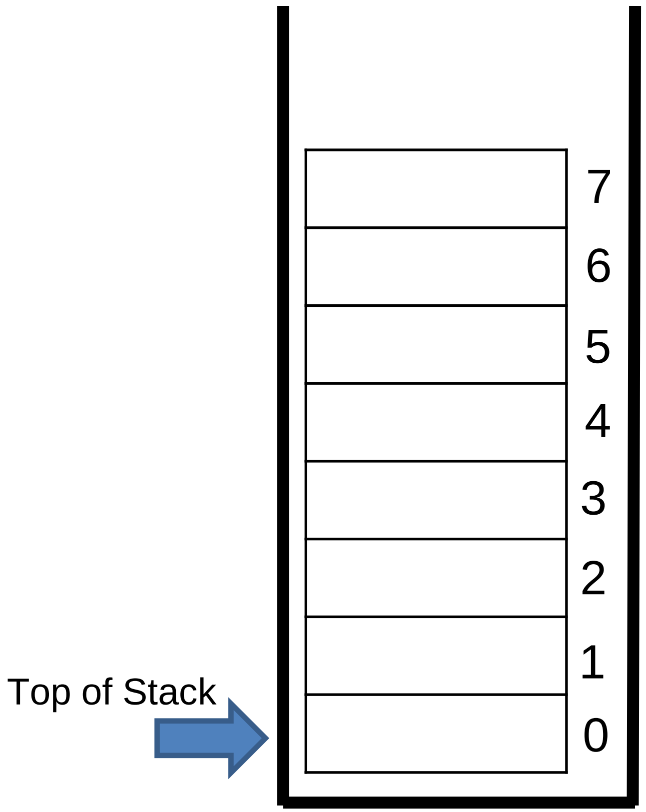
Here, we've created an array of size 8 to hold our stack data



# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




The bottom of the stack will always be at index 0, and grows towards the higher indices

```
String[] data = new String[8]
```

When the stack is empty, the index of the bottom of the stack, and the index of the top of the stack will be the same

```
top_of_stack = 0
```

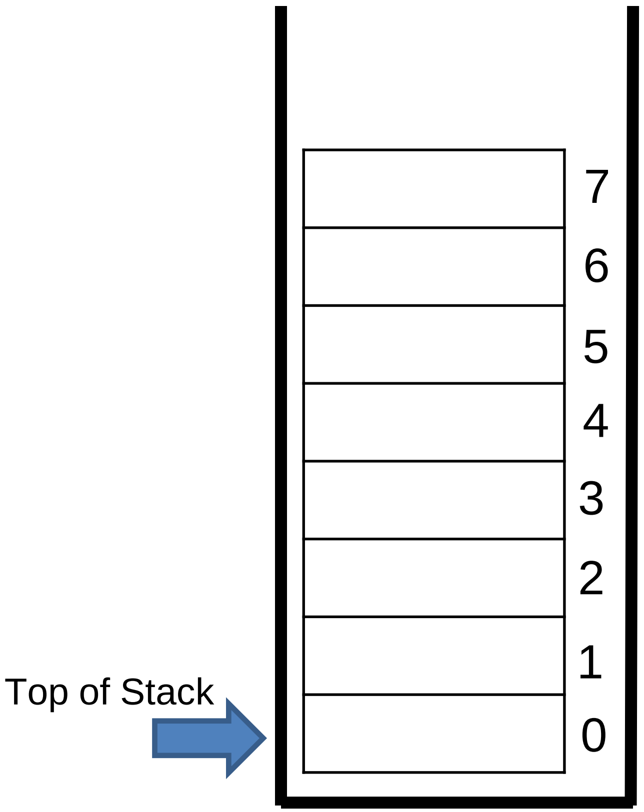
The size of the stack will start at 0

```
size = 0
```

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public void push(newElement){



}
```

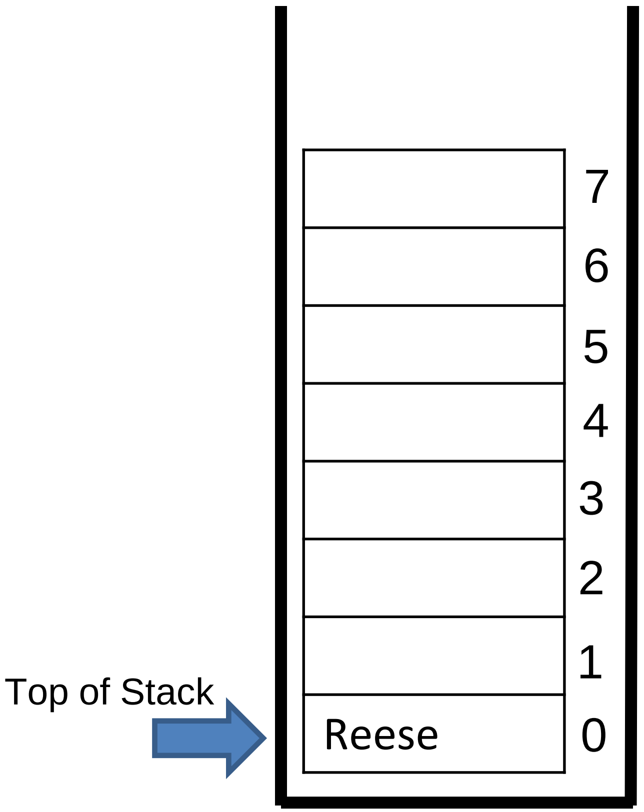
Stack Instance Fields

```
String[] data = new String[8]
         top_of_stack = 0
                 size = 0
```

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public void push(newElement){

  if stack is empty:
      place newElement at current top_of_stack
      size++


  if stack if full:
      return



}
```

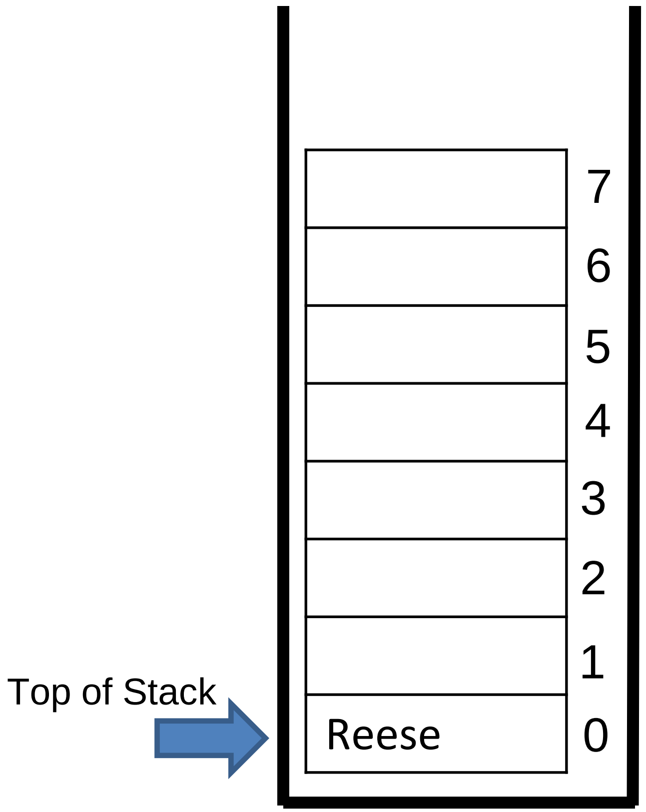
Stack Instance Fields

```
String[] data = new String[8]
        top_of_stack = 0
                size = 1
```

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public void push(newElement){

  if stack is empty:
      place newElement at current top_of_stack
      size++

  if stack if full:
      return

  else:
      top_of_stack++;
      place newElement at index top_of_stack
      size++
}
```

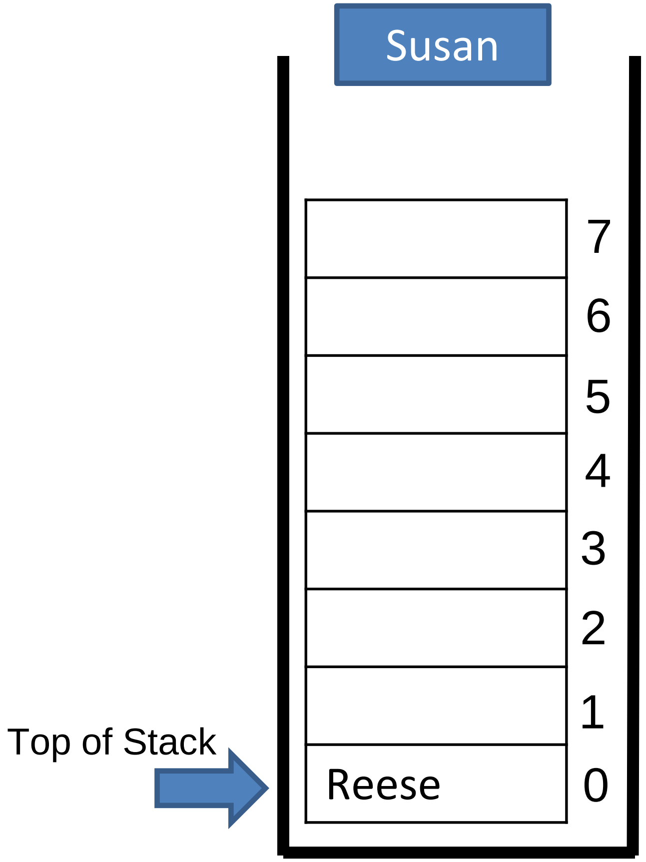
Stack Instance Fields

```
String[] data = new String[8]
          top_of_stack = 0
                  size = 1
```

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public void push(newElement){

  if stack is empty:
      place newElement at current top_of_stack
      size++


  if stack if full:
      return


  else:
      top_of_stack++;
      place newElement at index top_of_stack
      size++
}
```

Stack Instance Fields

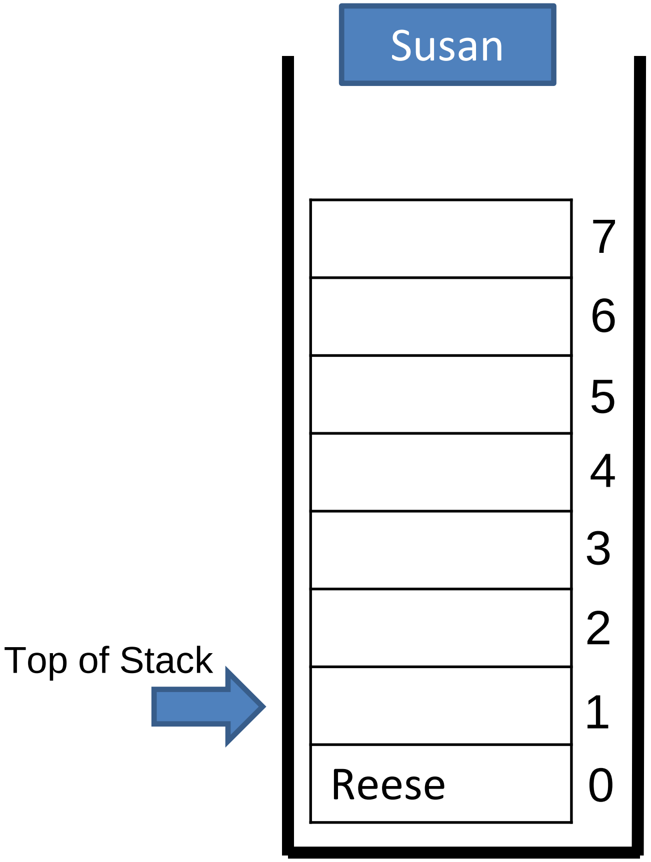
```
String[] data = new String[8]
          top_of_stack = 0
                  size = 1
```

```
stack.push("Susan")
```

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public void push(newElement){

  if stack is empty:
      place newElement at current top_of_stack
      size++

  if stack if full:
      return

  else:
      top_of_stack++;
      place newElement at index top_of_stack
      size++
}
```

Stack Instance Fields

```
String[] data = new String[8]
         top_of_stack = 1
                 size = 1
```

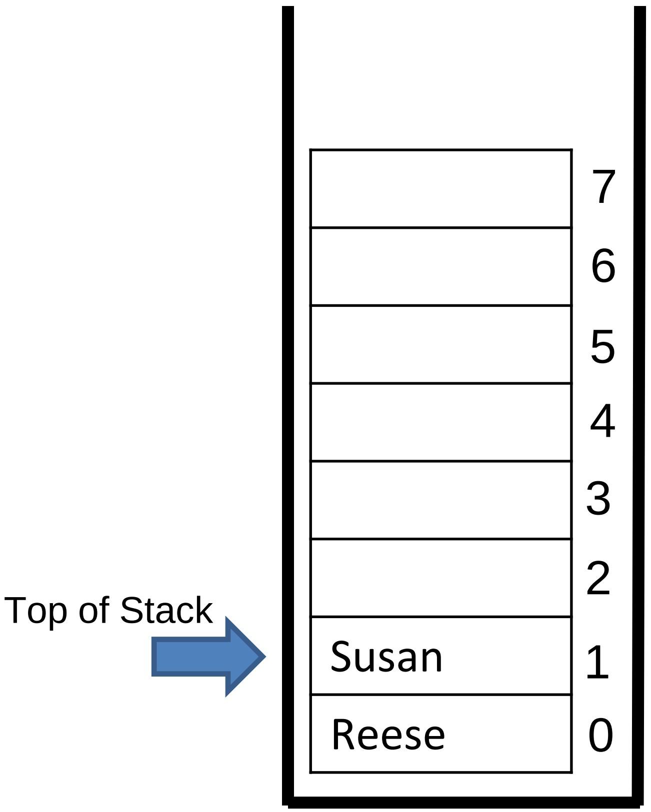
```
stack.push("Susan")
```

# Stack Implementation (Array)

To Do List:

- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public void push(newElement){

  if stack is empty:
      place newElement at current top_of_stack
      size++

  if stack if full:
      return

  else:
      top_of_stack++;
      place newElement at index top_of_stack
      size++
}
```

stack.push("Susan")

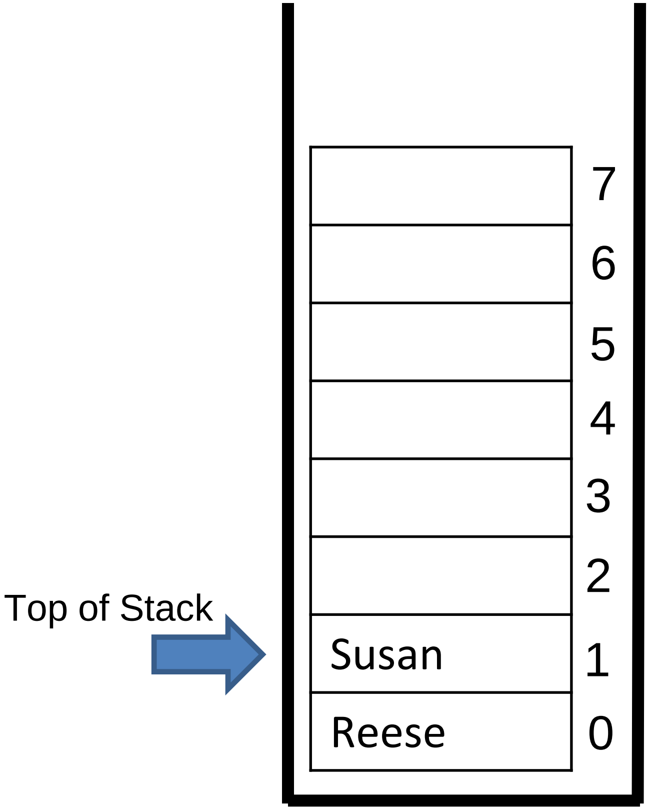
Stack Instance Fields

```
String[] data = new String[8]
         top_of_stack = 1
                 size = 1
```

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public void push(newElement){

  if stack is empty:
      place newElement at current top_of_stack
      size++


  if stack if full:
      return


  else:
      top_of_stack++;
      place newElement at index top_of_stack
      size++
}
```

stack.push("Susan")

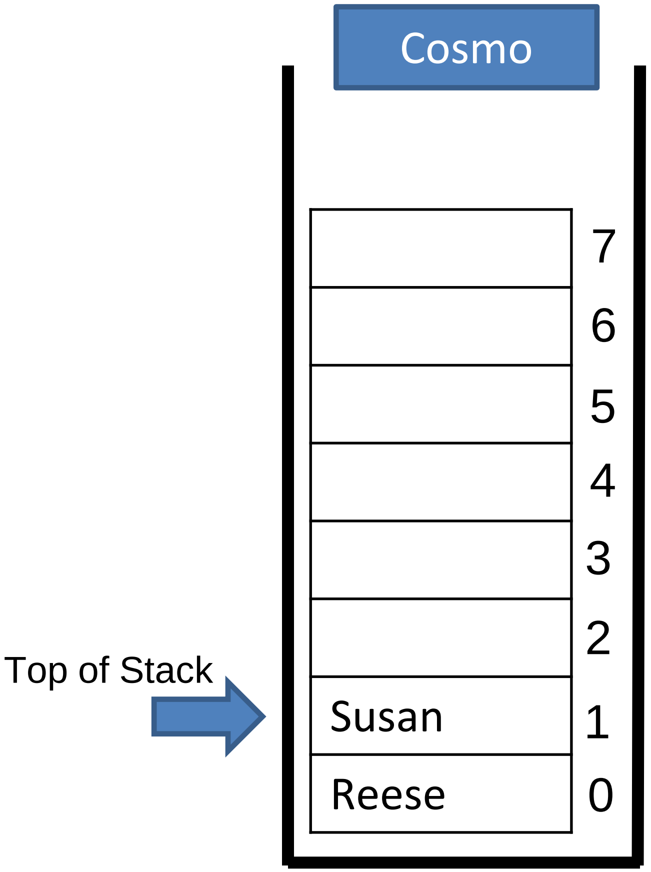
Stack Instance Fields

```
String[] data = new String[8]
          top_of_stack = 1
                  size = 2
```

# Stack Implementation (Array)

Here, we've created an array of size 8 to hold our stack data

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()




```
public void push(newElement){

  if stack is empty:
      place newElement at current top_of_stack
      size++


  if stack if full:
      return


  else:
      top_of_stack++;
      place newElement at index top_of_stack
      size++
}
```

```
stack.push("Cosmo")
```

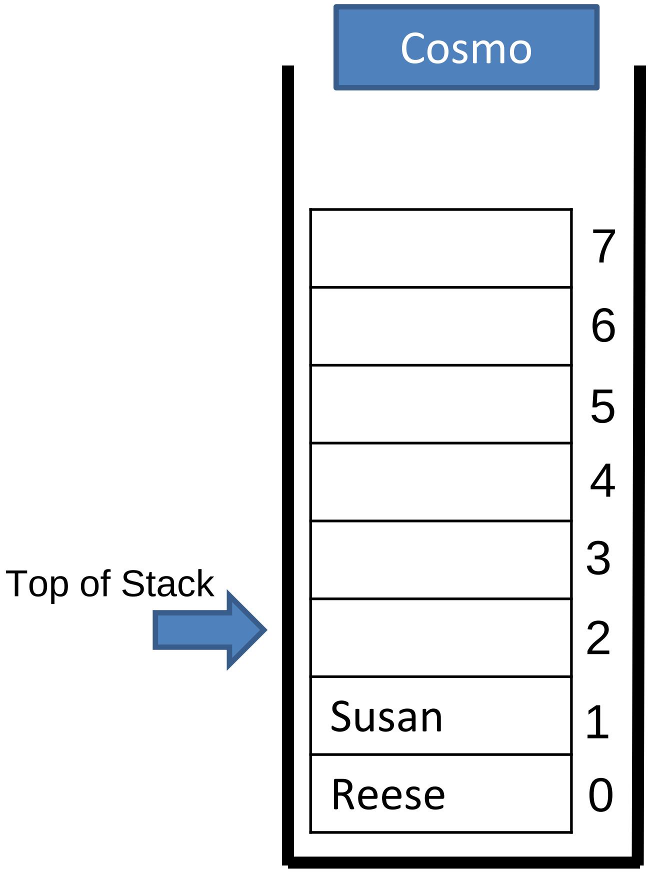
Stack Instance Fields

```
String[] data = new String[8]
         top_of_stack = 1
                 size = 2
```

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public void push(newElement){

  if stack is empty:
      place newElement at current top_of_stack
      size++



  if stack if full:
      return


  else:
      top_of_stack++;
      place newElement at index top_of_stack
      size++
}
```

Stack Instance Fields

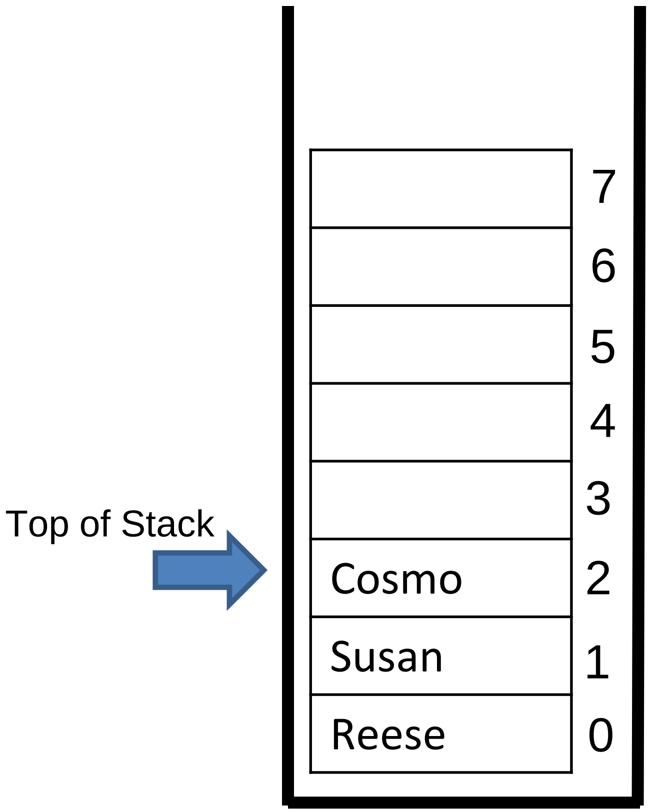
```
String[] data = new String[8]
         top_of_stack = 2
                 size = 2
```

```
stack.push("Cosmo")
```

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public void push(newElement){

  if stack is empty:
      place newElement at current top_of_stack
      size++



  if stack if full:
      return


  else:
      top_of_stack++;
      place newElement at index top_of_stack
      size++
}
```

Stack Instance Fields

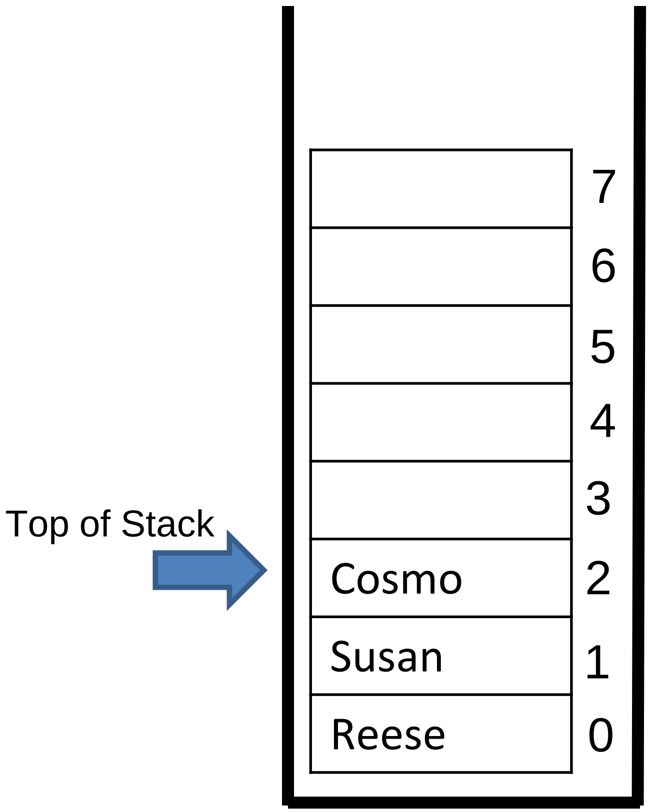
```
String[] data = new String[8]
          top_of_stack = 2
                  size = 3
```

```
stack.push("Cosmo")
```

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public void pop(){



}
```

The pop method will always remove the element on the top of the stack

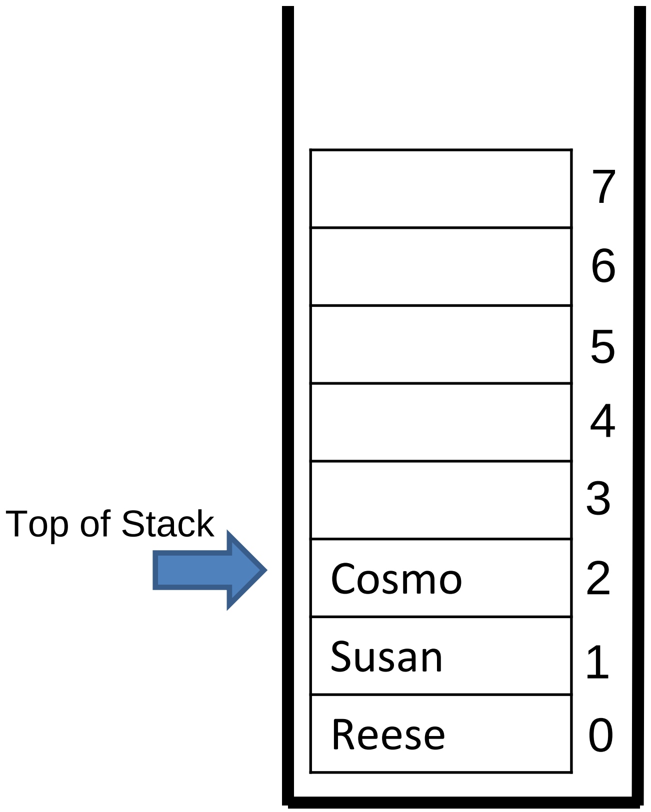
Stack Instance Fields

```
String[] data = new String[8]
         top_of_stack = 2
                 size = 3
```

# Stack Implementation (Array)

Here, we've created an array of size 8 to hold our stack data

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()




```
public void pop(){

   if stack is empty:
       return

   Set index top_of_stack to be null
   top_of_stack--
   size--

}
```

Stack Instance Fields

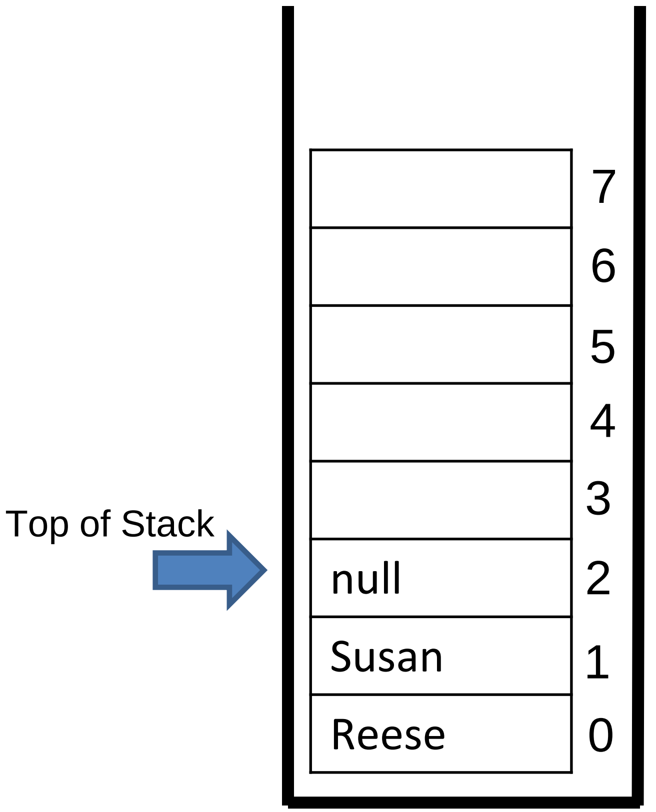
```
String[] data = new String[8]
            top_of_stack = 2
                    size = 3
```

```
stack.pop()
```

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public void pop(){

   if stack is empty:
       return

   Set index top_of_stack to be null
   top_of_stack--
   size--

}
```

Stack Instance Fields

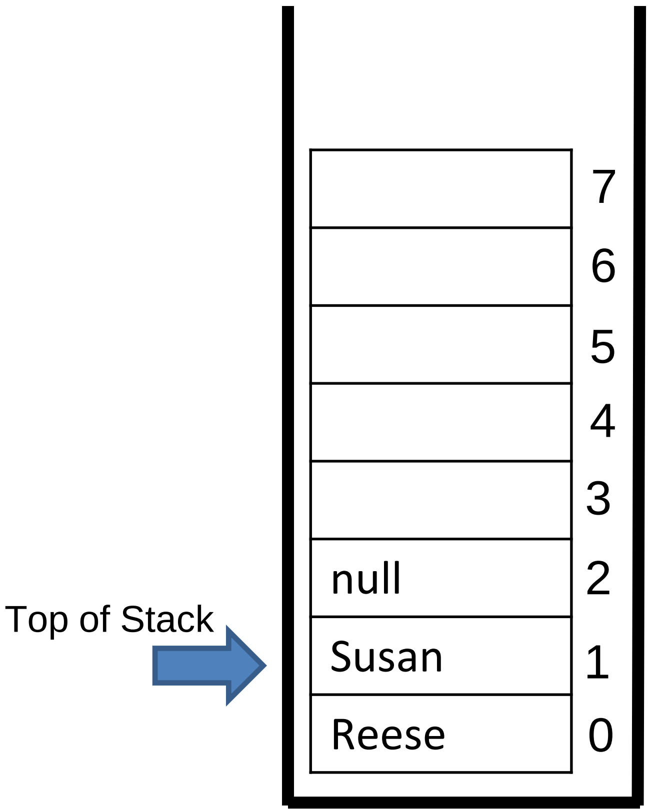
```
String[] data = new String[8]
             top_of_stack = 2
                     size = 3
```

```
stack.pop()
```

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public void pop(){

   if stack is empty:
       return

   Set index top_of_stack to be null
   top_of_stack--
   size--

}
```

Stack Instance Fields

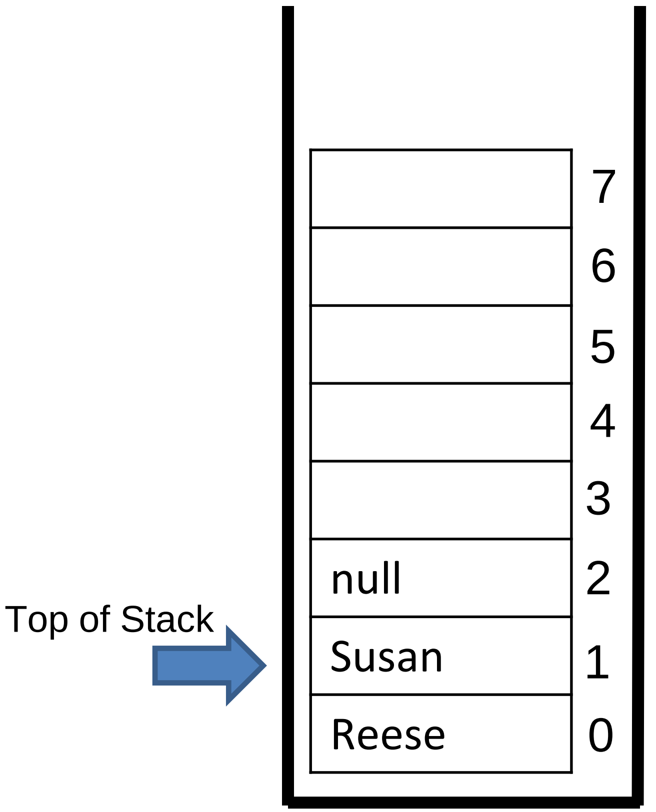
```
String[] data = new String[8]
              top_of_stack = 1
                      size = 3
```

```
stack.pop()
```

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public void pop(){

   if stack is empty:
       return

   Set index top_of_stack to be null
   top_of_stack--
   size--

}
```

Stack Instance Fields

```
String[] data = new String[8]
             top_of_stack = 1
                     size = 2
```

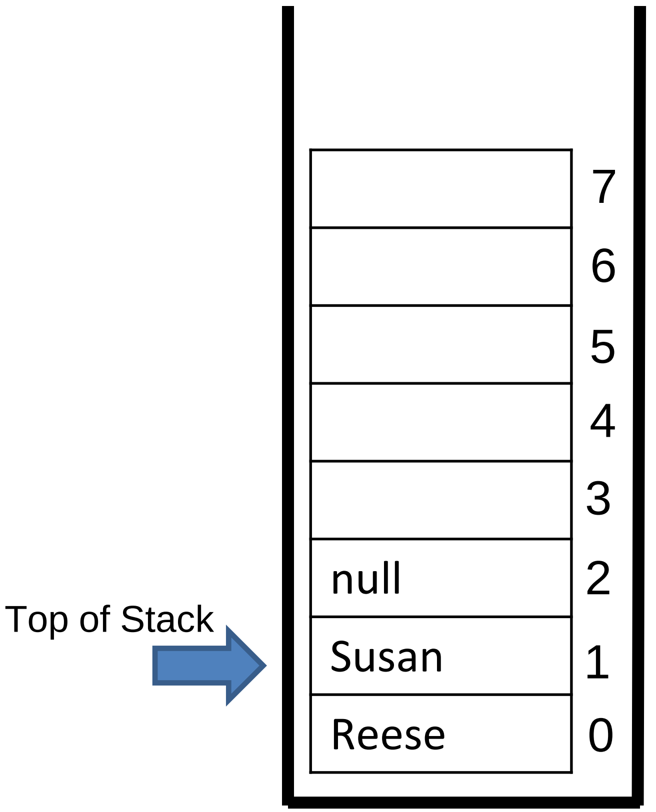
```
stack.pop()
```

# Stack Implementation (Array)

To Do List:

- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public void pop(){

   if stack is empty:
       return

   Set index top_of_stack to be null
   top_of_stack--
   size--

}
```

Stack Instance Fields

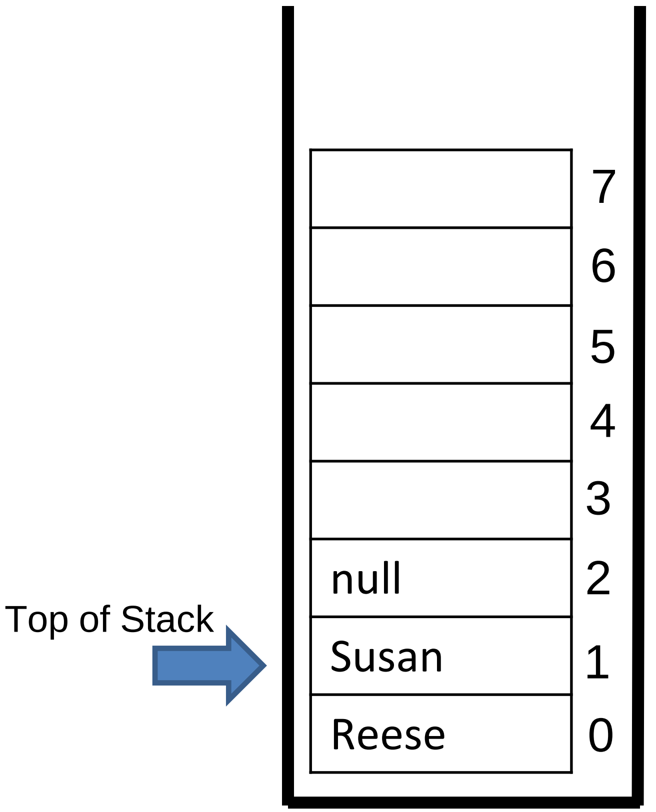
```
String[] data = new String[8]
            top_of_stack = 1
                    size = 2
```

Note: This method does not return the element that was removed, however there may be times where the pop() method returns the element that got removed

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public String peek(){



}
```

Stack Instance Fields

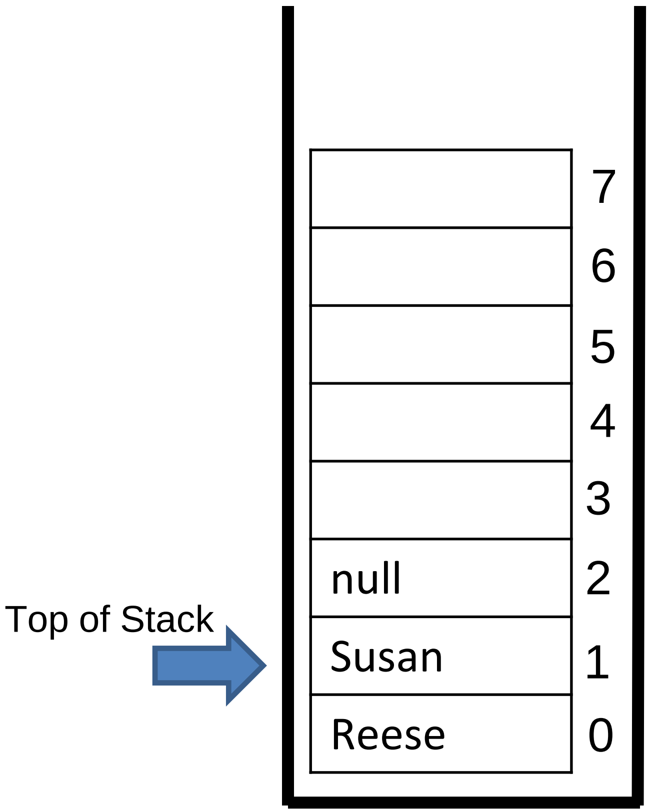
```
String[] data = new String[8]
         top_of_stack = 1
                 size = 2
```

The `peek()` method returns the element that is currently on the top of the stack

# Stack Implementation (Array)

Here, we've created an array of size 8 to hold our stack data

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()




```
public String peek(){

  If stack is not empty:
     return data[top_of_stack]

}
```

Stack Instance Fields

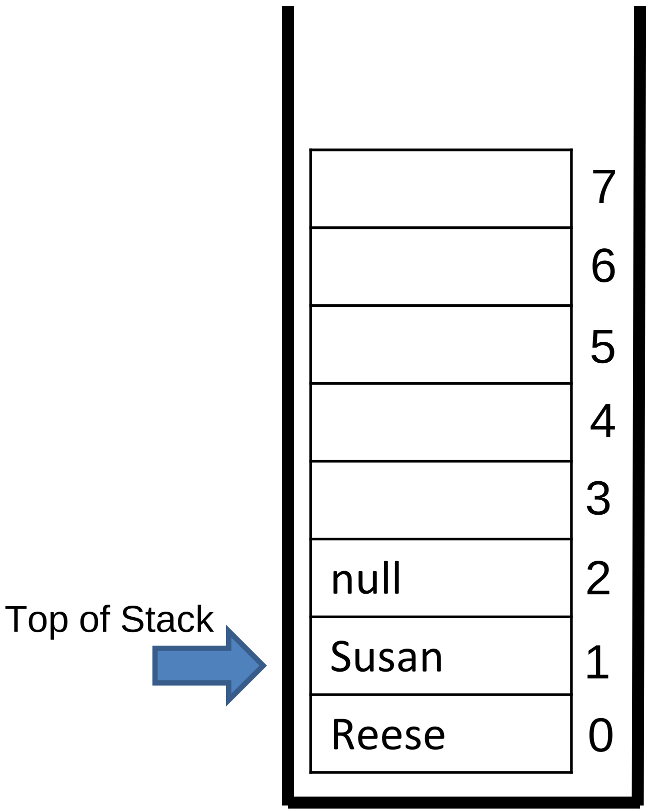
```
String[] data = new String[8]
         top_of_stack = 1
                 size = 2
```

The `peek()` method returns the element that is currently on the top of the stack

# Stack Implementation (Array)

To Do List:
- Push()
- Pop()
- Peek()
- IsEmpty()

Here, we've created an array of size 8 to hold our stack data




```
public boolean isEmpty(){

    if size == 0:
        return true

    else:
        return false
}
```

Stack Instance Fields

```
String[] data = new String[8]
            top_of_stack = 1
                    size = 2
```

The `isEmpty()` method returns a boolean: true if the stack is empty, false if the stack is not empty