CSCI 132:
Basic Data Structures and Algorithms

Lessons Learned so far + Intro to Stacks

Reese Pearsall

Spring 2023

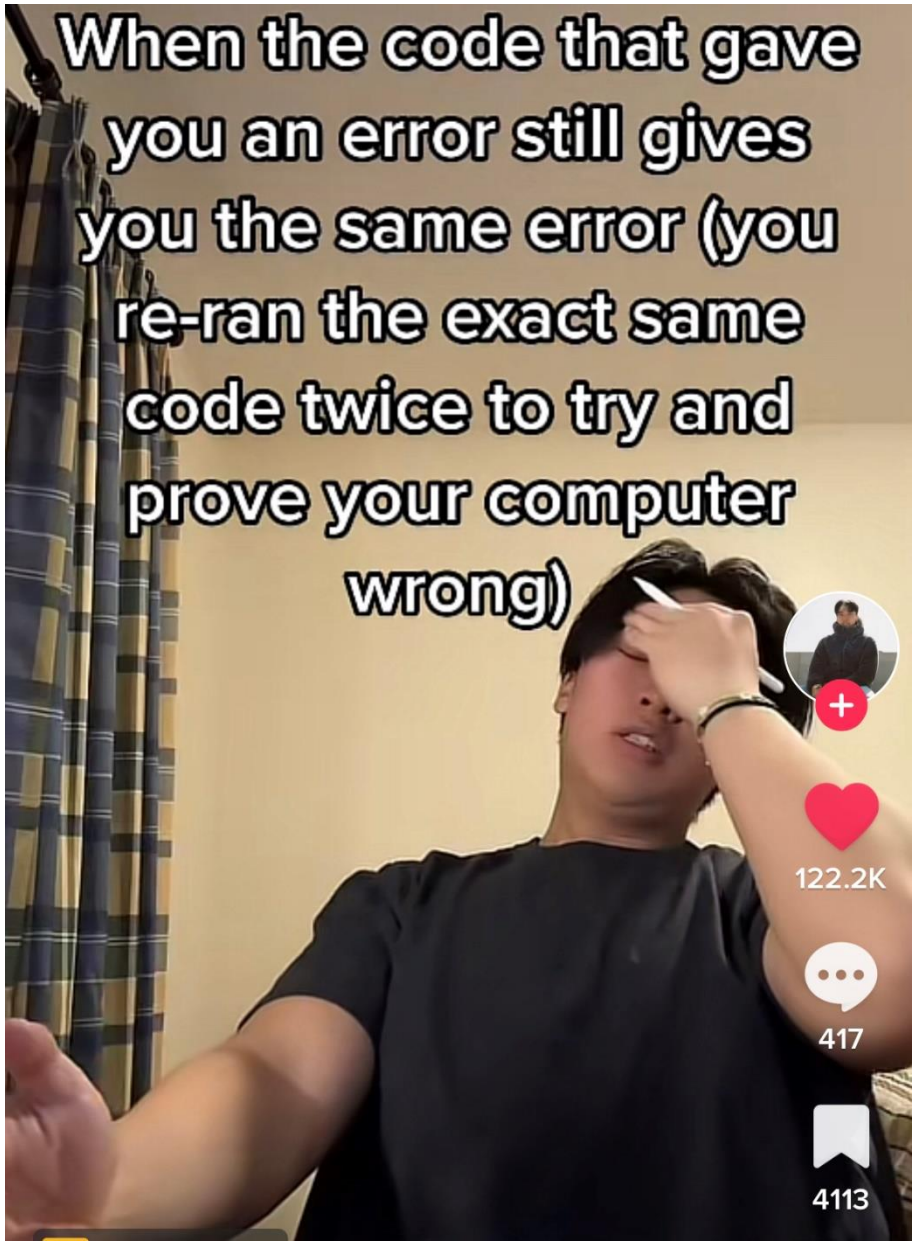
Announcements

Come get your midterm exam

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

Lab 7 due tomorrow @ 11:59 PM

Program 3 will be posted very soon



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`**)

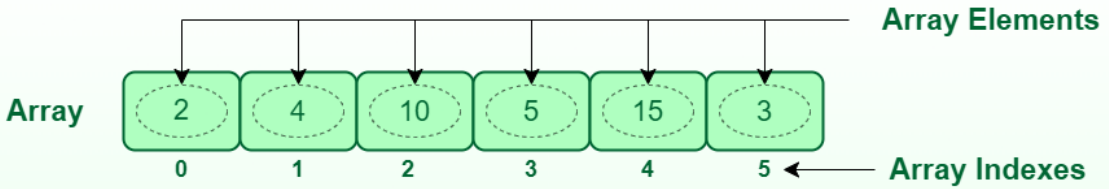
- Focus on worst-case scenario, and how the algorithm scales as **`n`** gets really big

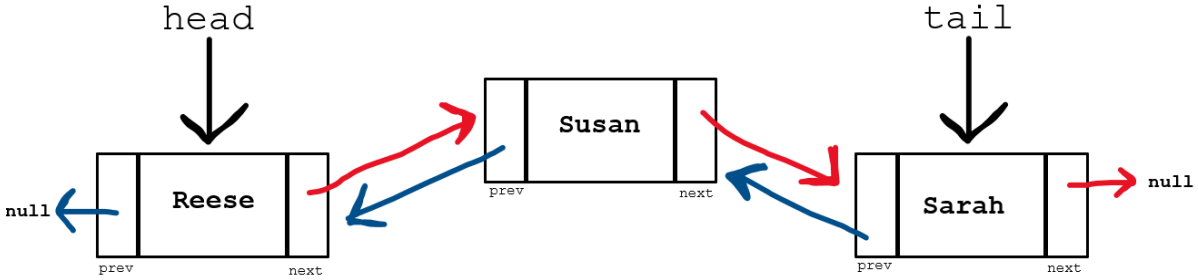
To find the total running time of an algorithm, we calculate the running-time of each operation in the algorithm and then add everything together

- 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:

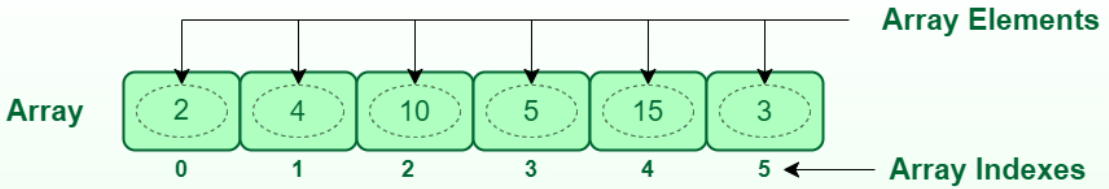


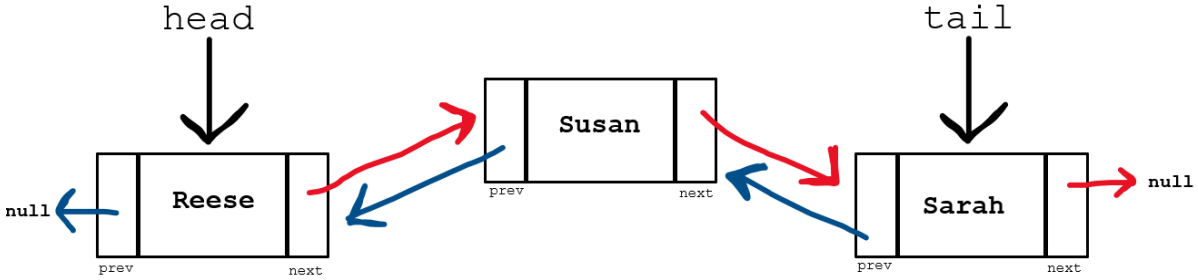


ArrayLists (Arrays)

Linked Lists

Data Structures so far:





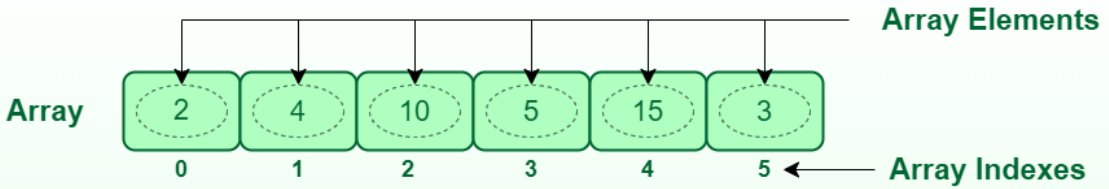
ArrayLists (Arrays)

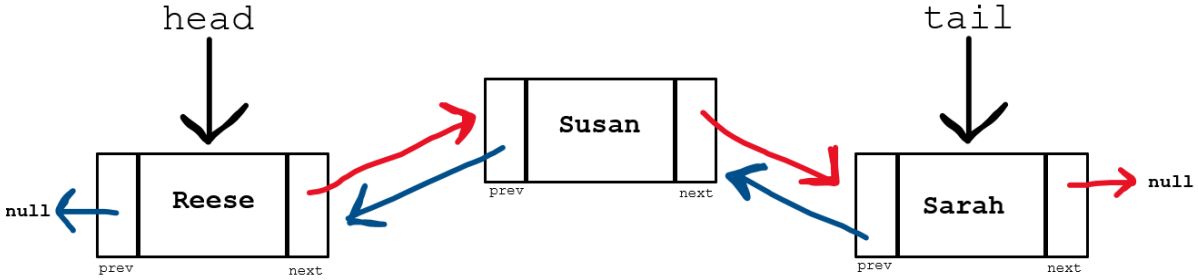
Can hold one data type

Linked Lists

~~Can hold multiple data types~~

Data Structures so far:





ArrayLists (Arrays)

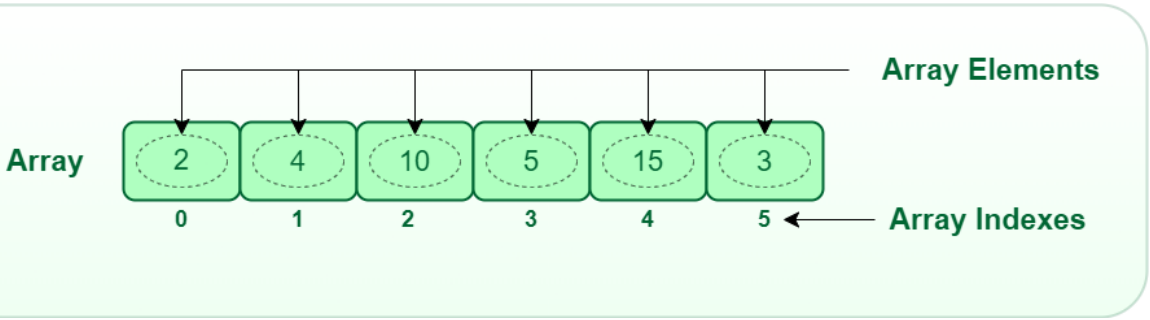
Can hold one data type

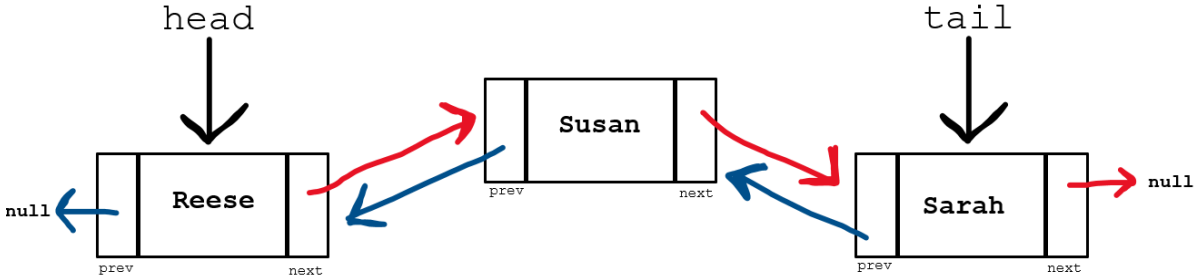
- Can also store objects, which allow for multiple data types

Linked Lists

Nodes in the linked list can hold multiple data types

Data Structures so far:





ArrayLists (Arrays)

Can hold one data type

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

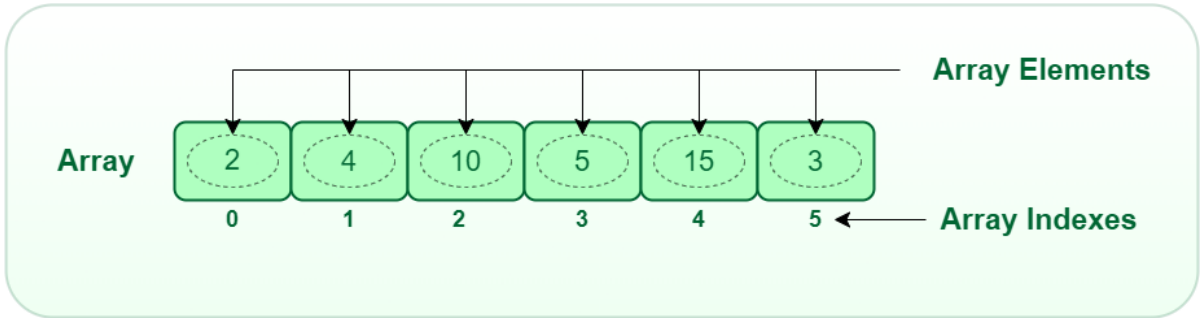
Entire array is stored at a **contiguous** spot in memory

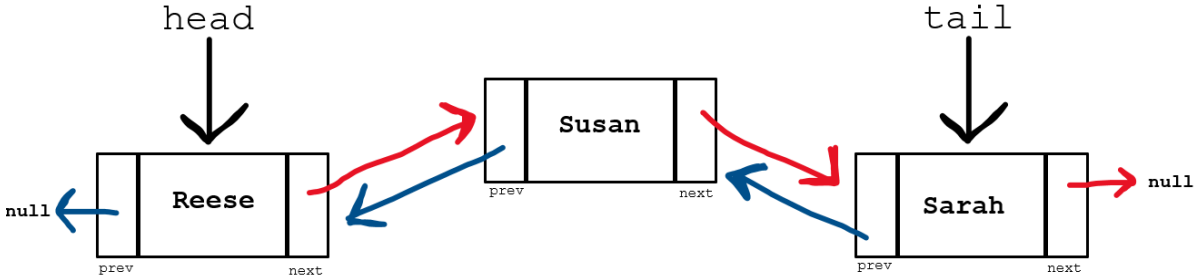
Linked Lists

Nodes in the linked list can hold multiple data types

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

Data Structures so far:





ArrayLists (Arrays)

Can hold one data type

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

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

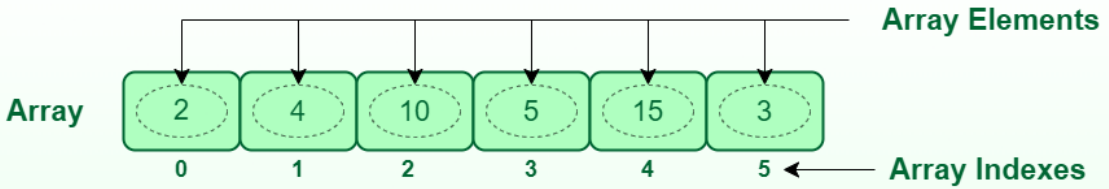
Linked Lists

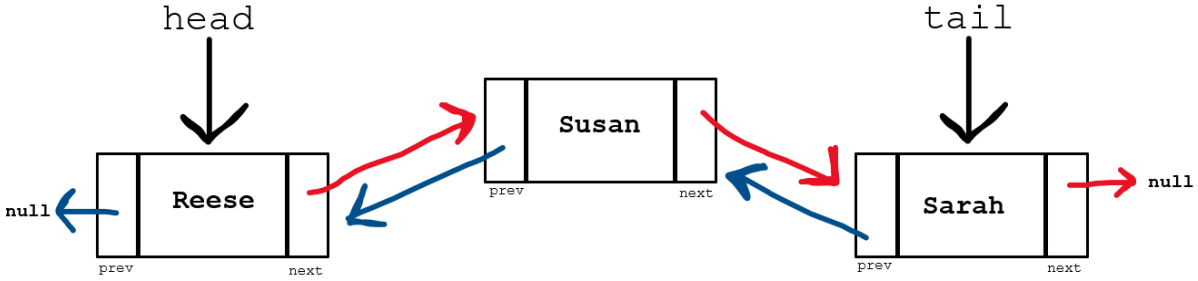
Nodes in the linked list can hold multiple data types

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)

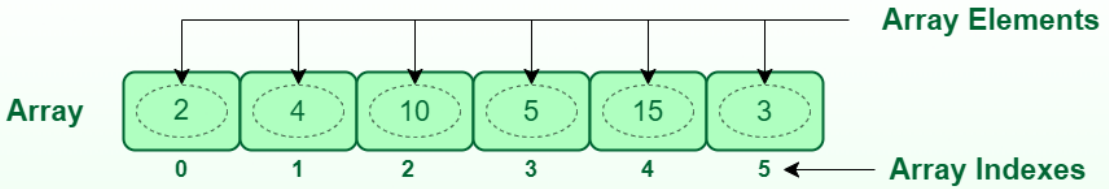
Can add new elements to data structure (resizable)

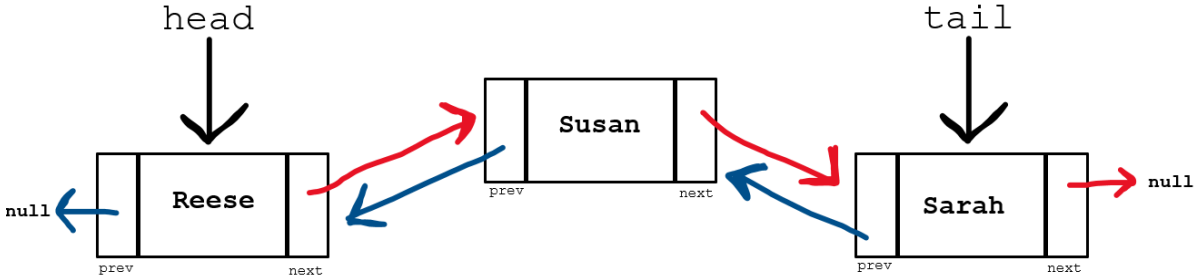
Linked Lists

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)

```
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;
```

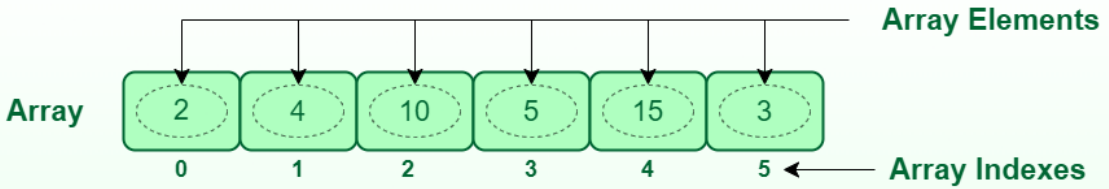
Create a brand-new array, copy everything over from old array

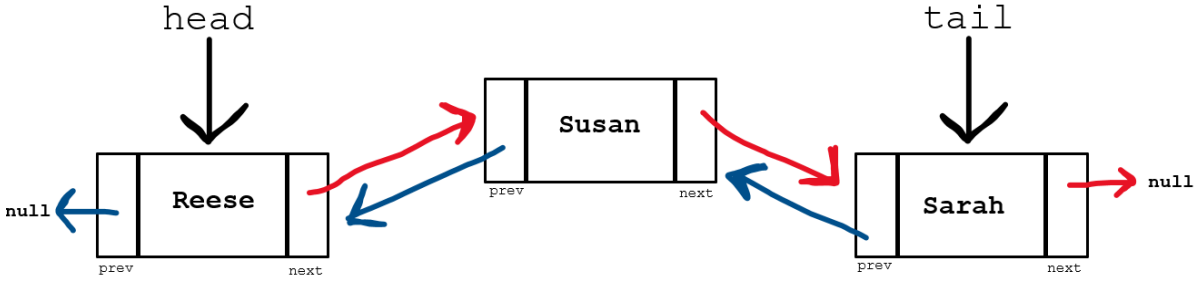
Linked Lists

```
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

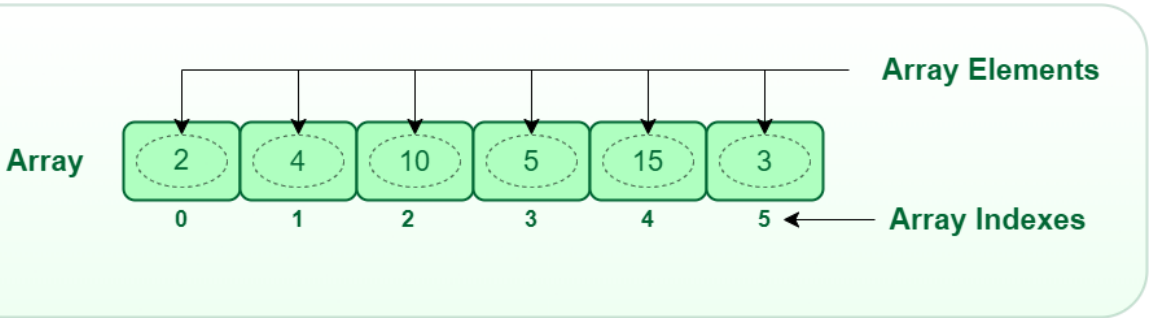
```
int[] newArray = new int[myArray.length + 1];
for(int i = 0; i < myArray.length; i++) {
            newArray[i] = myArray[i];
}

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

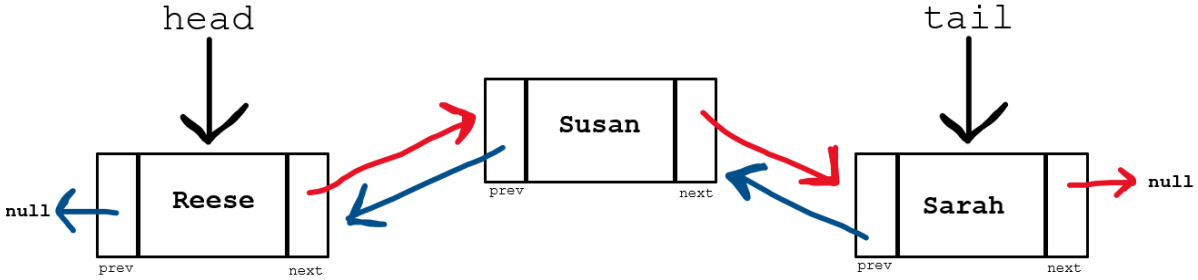
Create a brand-new array, copy everything over from old array O(n)

```
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)

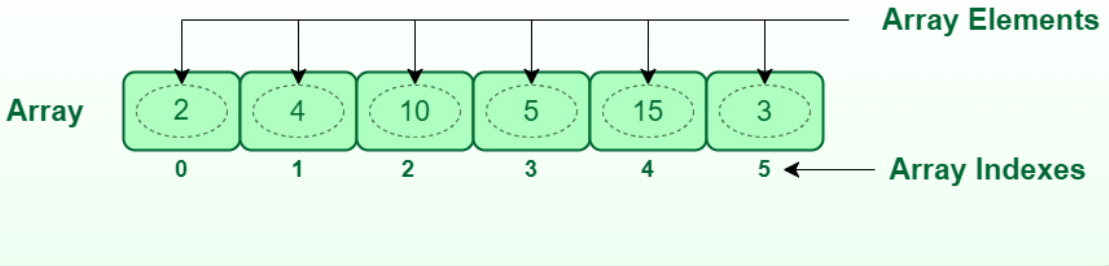
```
int[] newArray = new int[myArray.length + 1];
for(int i = 0; i < myArray.length; i++) {
            newArray[i] = myArray[i];
}

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

Linked Lists

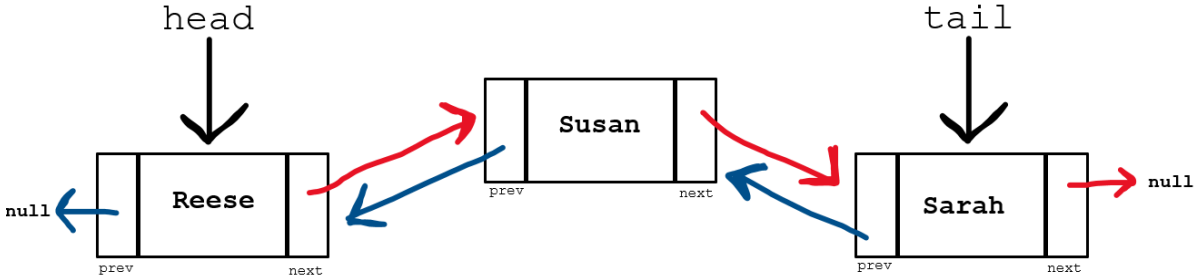
```
public void addToFront(Node newNode) {
    if(head == null) {
        head = newNode;
    }
    else {
        newNode.setNext(head);
        head = newNode;
    }
}
```

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)



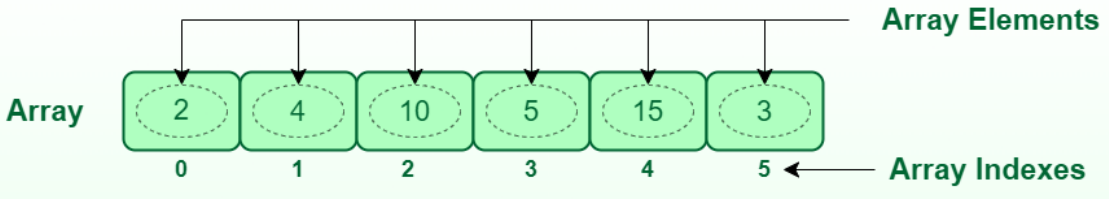
Linked Lists

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

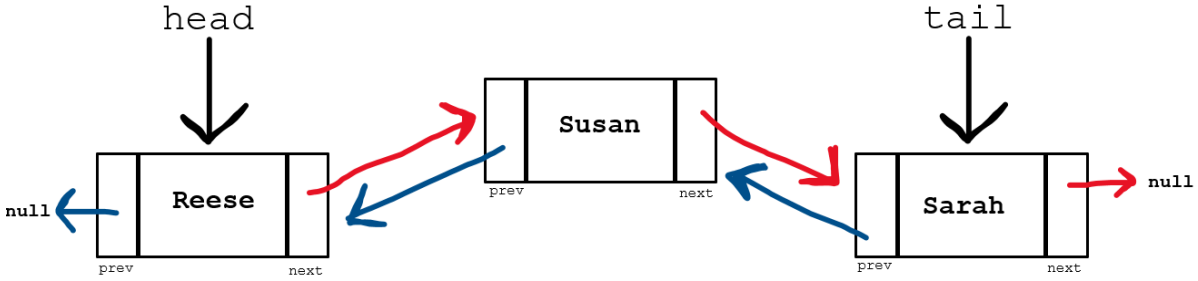
If you are constantly needing to add new elements to the data structure, using a Linked List requires much less work in the long run

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

Data Structures so far:



ArrayLists (Arrays)



Linked Lists

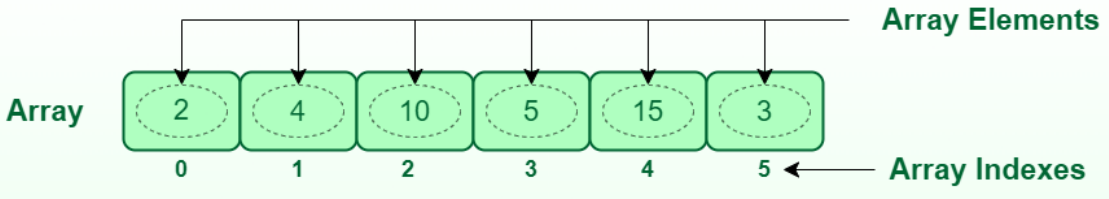
When to use each data structure?

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

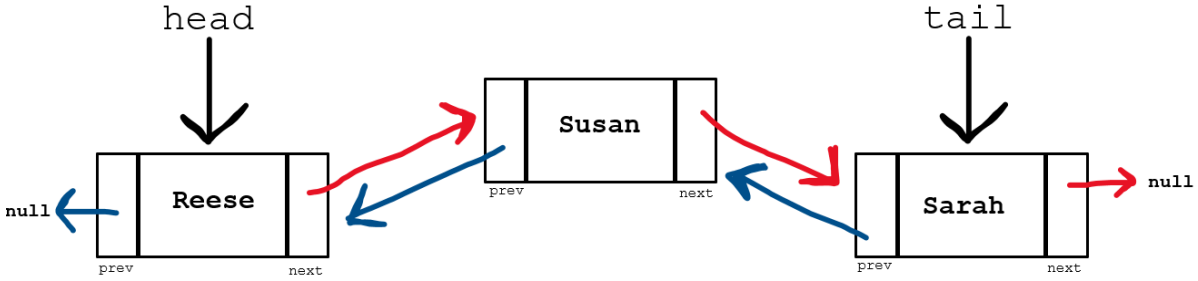
If you don't know how much data you need to store, or if you are constantly needing to add new elements to the data structure → **Linked Lists**

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

```
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

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

```
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

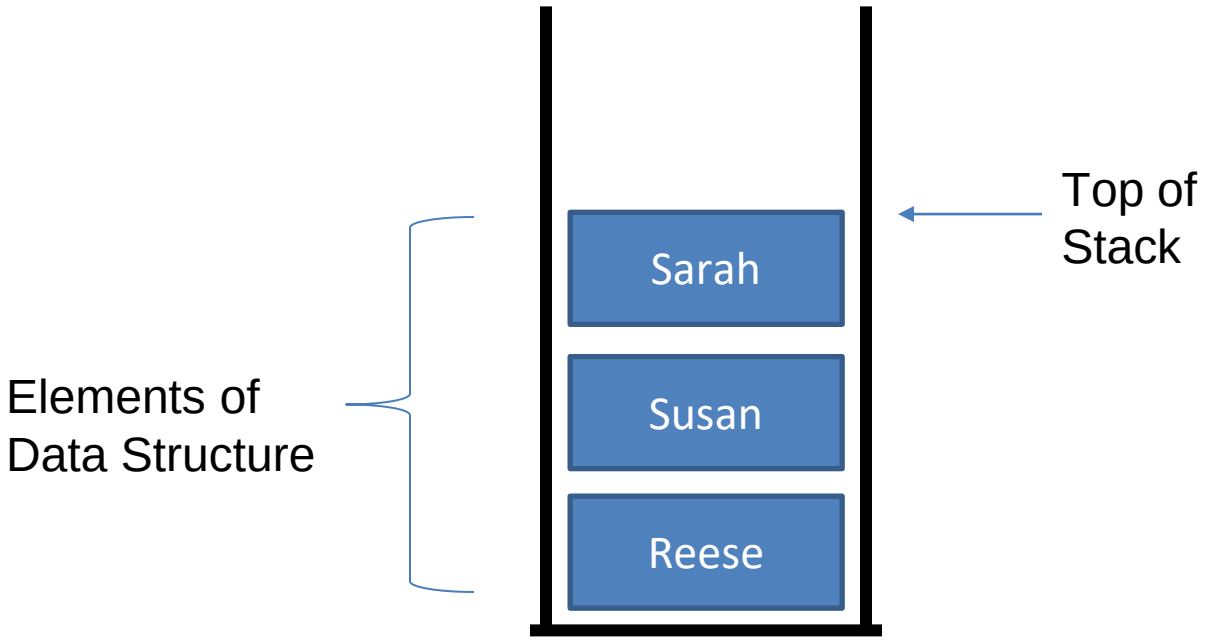
Methods

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

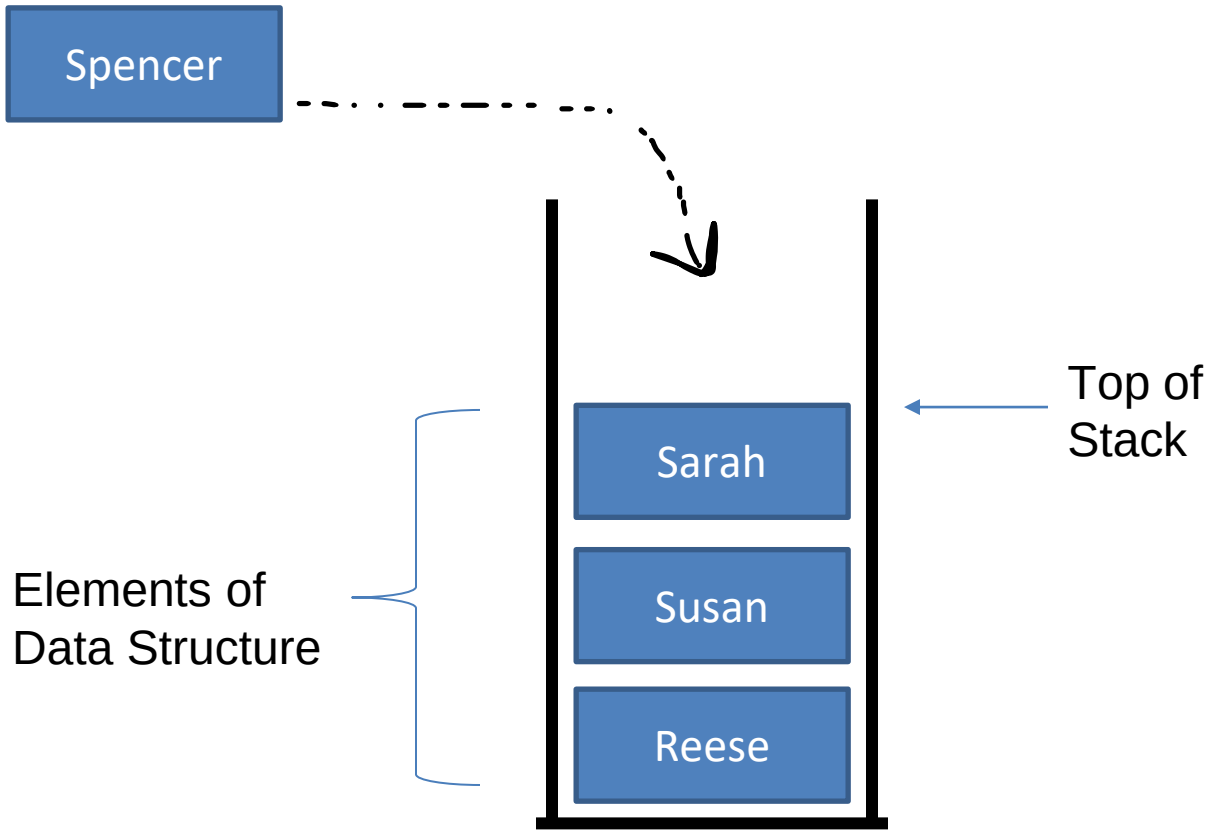
The Linked List Class

```
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



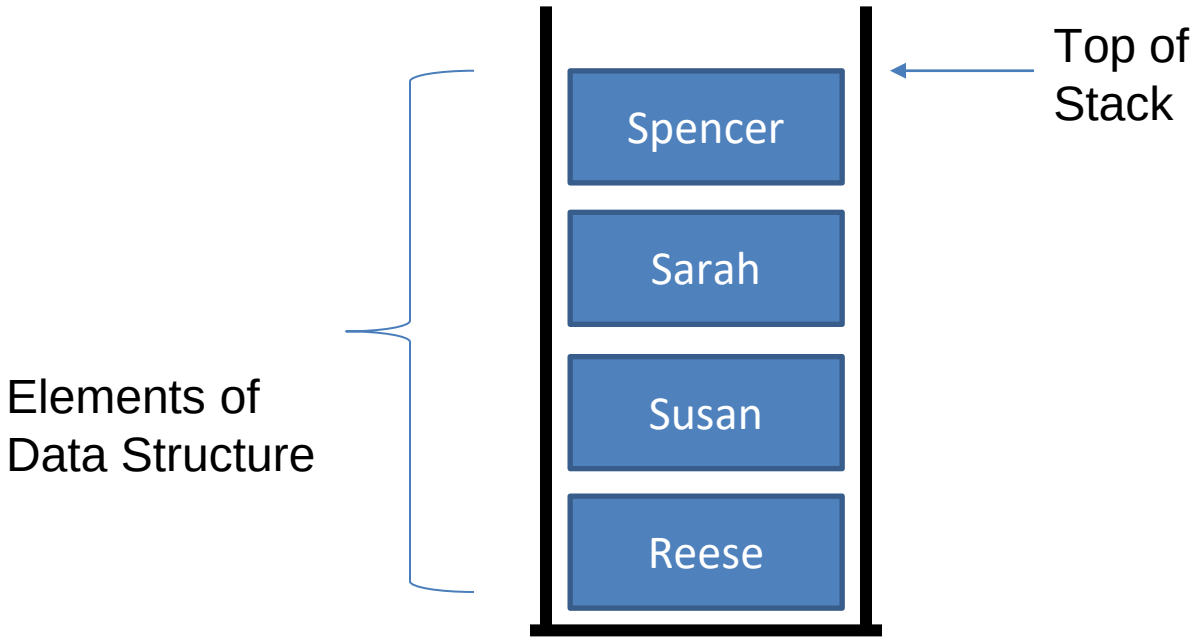
A **stack** is a data structure that can hold data, however the way we interact with a stack is a little bit different



When only interact with the top of the stack.

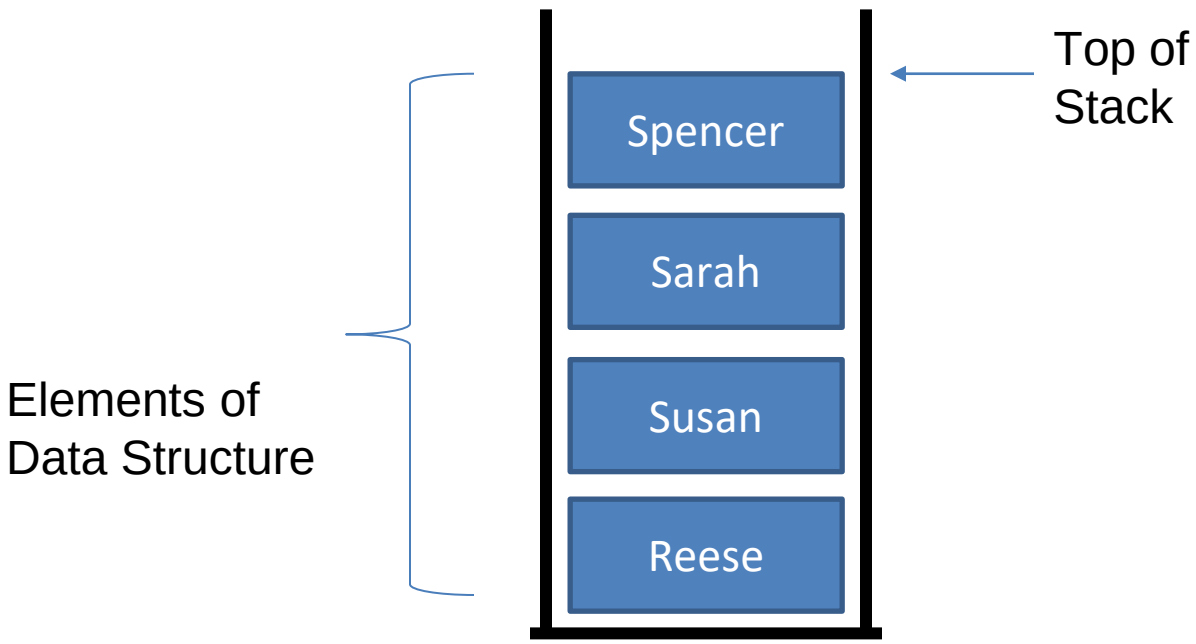
If we want to add a new element, we must put it on the top of the stack

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



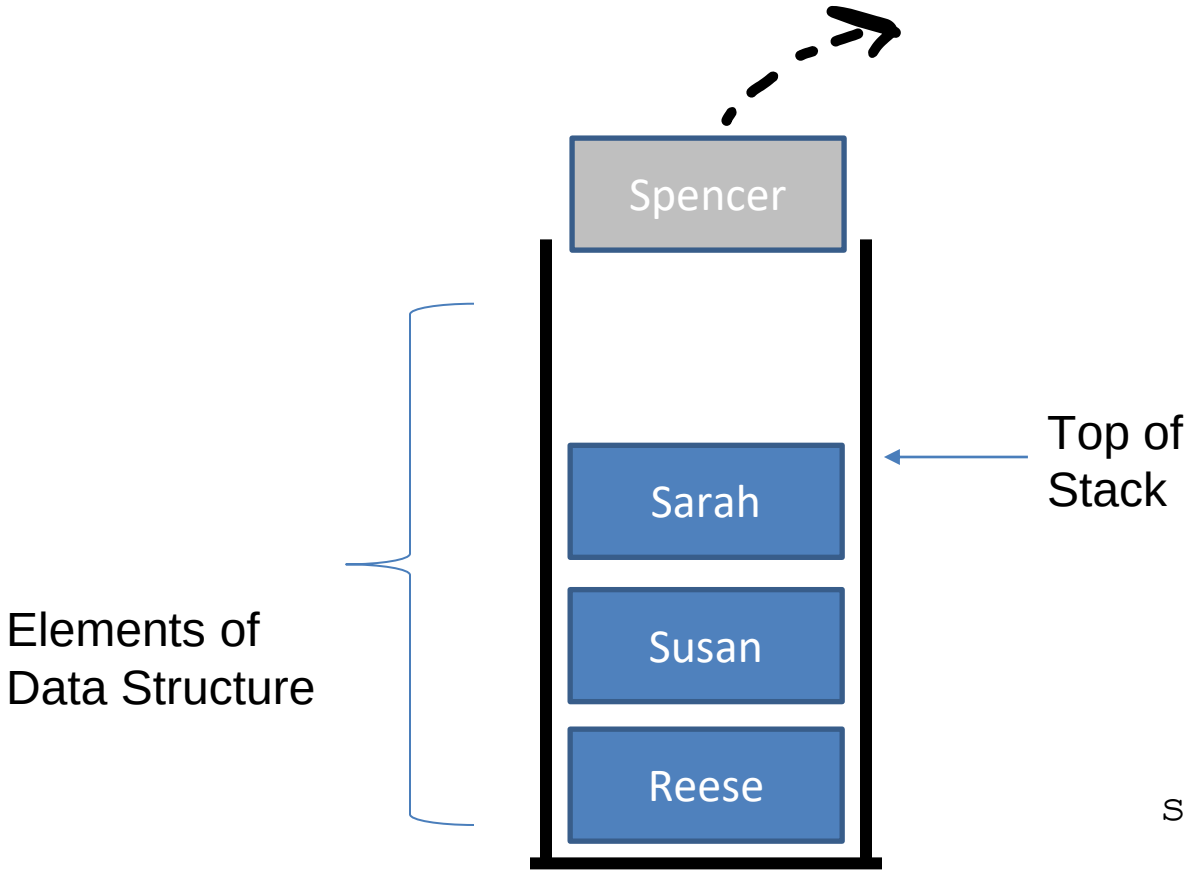
Adding something to a stack is known as the **push** operation

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)