

FUNDAMENTALS

STACKS, LISTS, AND GENERICS

Ruben Acuña

Fall 2021

MODULE OVERVIEW

- The goal of this module is to review concepts in stacks, lists, and generics.
- Learning Objectives:
 - Understand the key concepts of a stack.
 - Understand the key concepts of a linked list.
 - Apply the Java programming language to implement a stack.

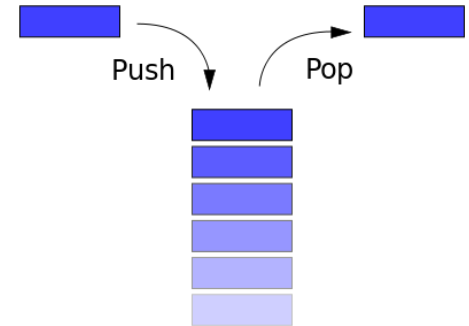


THE STACK CONCEPT

OVERVIEW

Stack

- Ordered, allows duplicates.
- Element access: Last-In First Out (LIFO)
- docs.oracle.com/javase/9/docs/api/java/util/Stack.html



Sedgewick gives a specification:

Pushdown (LIFO) stack

```
public class Stack<Item> implements Iterable<Item>
```

Stack()	<i>create an empty stack</i>
void push(Item item)	<i>add an item</i>
Item pop()	<i>remove the most recently added item</i>
boolean isEmpty()	<i>is the stack empty?</i>
int size()	<i>number of items in the stack</i>

Note that this API is incomplete:
• peek()?

And don't forget the rest of API bits that make this stuff actually usable (Serializable, Cloneable, Iterable, etc.)...

STACK EXAMPLE 1

Stack s = new Stack();



s.push(5);



s.pop();



s.push(1);



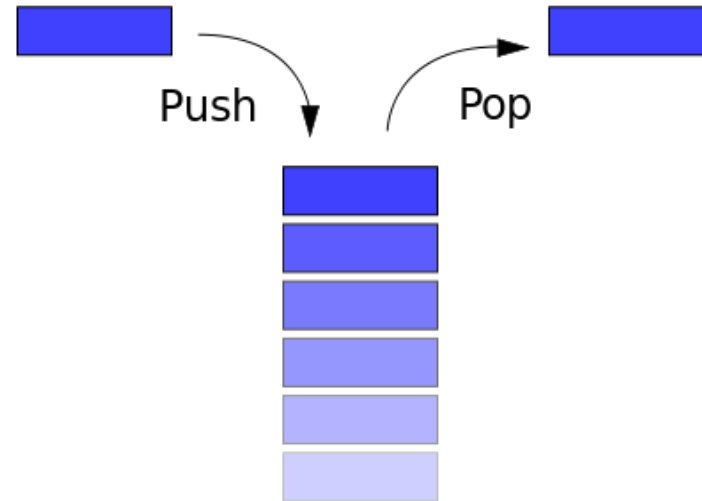
s.push(4);



s.push(0);



s.pop();



Applications:

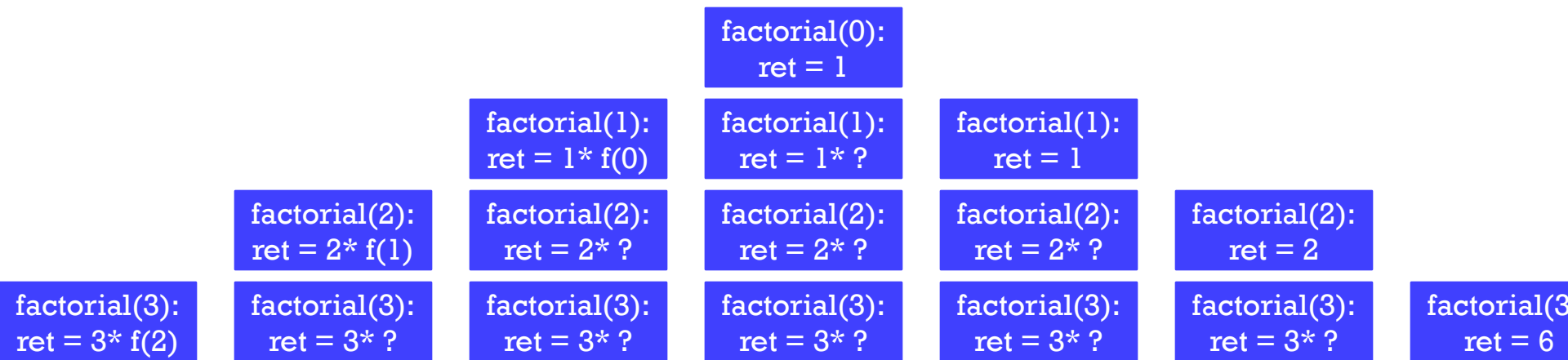
- Undo in a text editor.
- Tab history in a browser.
- Recording and maintaining function calls.

STACK EXAMPLE 2: CALLSTACK

During a program's execution, methods call methods, and those methods call even more methods.

The calls (i.e., who called what) as well as the local data for each call (i.e., variables), are tracked in the *callstack*. A function call is a *push* while a function return is a *pop*.

Remember the recursive factorial?





A SIMPLE STACK

REQUIREMENTS

Assumptions:

- Stores strings (no reason for choice!)
- Can use array to store data (justification: fast access)
- Methods don't need error handling (**NOT A GOOD ASSUMPTION**)

Need to implement a constructor and four methods (push, pop, isEmpty, and size).

Will the design be immutable or mutable?

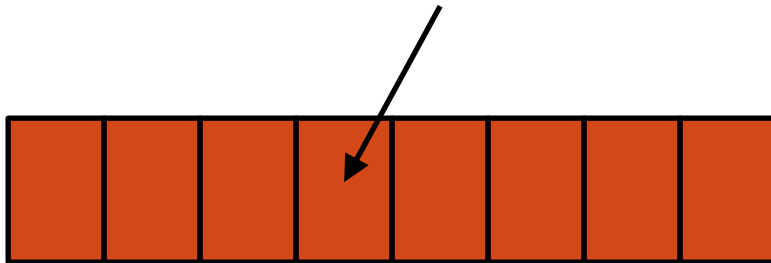
First problem: how to represent data...

REPRESENTING STACK DATA

- Already said we would use an array:



- Now what? All activity (push/pop) happens at one place – some index. Let's call it n .



- n will need to change as push or pop is called – since those methods change the contents of the stack.
- If n starts as 0, then push can increase it and pop can decrease it.

REPRESENTING STACK DATA

- Initially, $n = 0$:



- What should happen if we push? Array is updated and n increases.



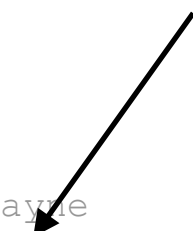
- n is now 1. Will its value correspond with the contents? Push again:



- n is now 2. What about pop?

IMPLEMENTATION

verbose name but at least we are honest...



Variables:

- data: array containing elements.
- n: number of elements in stack. also, the index in the array where the next should be placed.



```
//from Sedgewick and Wayne
public class FixedCapacityStringStack {
    private String[] data;
    private int n = 0;

    public FixedCapacityStringStack (int cap) {
        data = new String[cap];
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public void push(String item) {
        data[n++] = item;
    }

    public String pop() {
        return data[--n];
    }
}
```



But what of size()?

?



A BETTER STACK

POTENTIAL IMPROVEMENTS

- What if we want to store another datatype?
- What happens if we add an item when the array is full?
- What if we want to display the contents of the stack?
- Any potential memory use issues?

```
//from Sedgewick and Wayne
public class FixedCapacityStringStack {
    private String[] data;
    private int n = 0;

    public FixedCapacityStringStack (int cap) {
        data = new String[cap];
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public void push(String item) {
        data[n++] = item;
    }

    public String pop() {
        return data[--n];
    }
}
```

GENERIC

- Right now, our stack is limited to strings. How to make this more generic?
- Copy/pasting the class is not a good solution... Why?
- Java provides the idea of generics: `SomeClass<Type>`. Lets the programmer use `Type` as a datatype in the definition of `SomeClass`:

```
public class SomeClass<Element> {  
    public Element value;  
    public Node next;  
}
```

- Note1: Limited to Object types (i.e., *nullable*). Can still use with `int`, (etc), though.
- Note2: Java is annoying – see next slide.

STACK V2

//from Sedgewick and Wayne

```
public class FixedCapacityStack<Item> {
    private Item[] data;
    private int n = 0;

    public FixedCapacityStack(int cap) {
        data = (Item[]) new Object[cap];
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public void push(Item item) {
        data[n++] = item;
    }

    public Item pop() {
        return data[--n];
    }
}
```

//example usage

```
FixedCapacityStack st = new
    FixedCapacityStack<Integer>();
```

//Java (roughly) rewrites as:

```
FixedCapacityStack st = new
    FixedCapacityStack$0();
```

//class specialization generated from
parameterization.

```
public class FixedCapacityStack$0 {
    private Integer[] data;
    private int n = 0;

    public FixedCapacityStack(int cap) {
        data = (Integer[]) new Object[cap];
    }

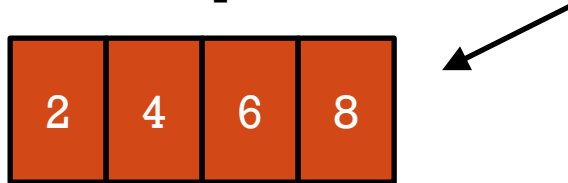
    public boolean isEmpty() {
        return n == 0;
    }

    public void push(Integer item) {
        data[n++] = item;
    }

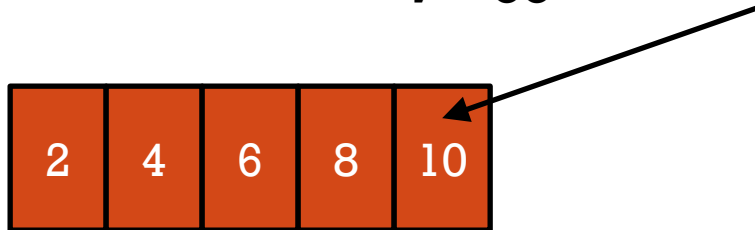
    public Integer pop() {
        return data[--n];
    }
}
```

RESIZING ARRAY DATA

- There is a problem when $n = \text{data.length}$:



- What will happen if we push?
- Solution: make array bigger. Then n will be okay.



- Here we made the array one element larger. Does this seem reasonable?

STACK V3

Looks good?

- What happens if `cap=0`?
- If we changed `push`, does that mean we need something in `pop`?

In general, how do we feel about `Stack(int)` and `pop()`?

```
//(mostly) from Sedgewick and Wayne
public class Stack<Item> {
    private Item[] data;
    private int n = 0;

    public Stack(int cap) {
        data = (Item[]) new Object[cap];
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public void push(Item item) {
        if(data.length == n)
            resize();
        data[n++] = item;
    }

    public Item pop() {
        return data[--n];
    }

    private void resize() {
        data = Arrays.copyOf(data,
            data.length * 2);
    }
}
```

MORE POLISH

Would be nice if the user did not have to worry about specifying an initial capacity.

(100 is arbitrary here...?)

```
public Stack() {  
    data = (Item[]) new Object[100];  
}
```

Pop fails if the stack is empty.

Will return an array exception but we should do better.

```
public Item pop() {  
    if (isEmpty())  
        throw new EmptyStackException();  
    return data[--n];  
}
```

IMPLEMENTING ITERABLE

Suppose we wanted to support the following:

```
for(Integer i : someStack)
    System.out.println("element: " + i);
```

What would we need to do? We need to implement the `Iterable` interface provided by Java:

```
public interface Iterable<Item> {
    Iterator iterator();
}
```

Not much here. The point is that the object returns an *Iterator* object from a method with a standard name.

ITERATOR

An iterator's job is to present elements from a collection one at a time while hiding how those elements are concretely stored.

What would we need to do? This time we need to create a new object that implements the Iterator interface with respect to how Stack stores data.

```
public interface Iterator<Item> {  
    boolean hasNext(); //true if there is another element  
    Item next();       //returns next object. advances.  
    void remove();    //"safe" way to remove elements.  
}
```

DONE!!!*

```
//(mostly) from Sedgewick and Wayne
public class Stack<Item> implements Iterable<Item>{
    ...

    private Iterable<Item> iterator() {
        return new ReverseArrayIterator();
    }

    private class ReverseArrayIterator implements Iterator<Item> {
        private int i = n;
        public boolean hasNext() { return i > 0; }
        public Item next() {return data[--i];} //is this okay?
        public void remove() {} //why blank?
    }
}
```

*Not really. This code can fail in a horrible way.... What is it?

ARRAY STACK

```
//(mostly) from Sedgewick and Wayne
public class Stack<Item> {
    private Item[] data;
    private int n = 0;

    public Stack(int cap) {
        data = (Item[]) new Object[cap];
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public void push(Item item) {
        if(data.length == n)
            resize();
        data[n++] = item;
    }

    public Item pop() {
        return data[--n];
    }

    private void resize() {
        data = Arrays.copyOf(data,
            data.length * 2);
    }
}
```

PERFORMANCE EVALUATION

Thus far we have managed to implement the Stack ADT functionality just fine. However, we still need to evaluate its performance (i.e. complexity) before calling it a day.

- What is the Big-Oh of isEmpty()?
- What is the Big-Oh of pop()?
- What is the Big-Oh of push()?

Now what?

```
public boolean isEmpty() {
    return n == 0;
}
public Item pop() {
    if(isEmpty())
        throw new EmptyStackExcep...
    return data[--n];
}
public void push(Item item) {
    if(data.length == n)
        resize();
    data[n++] = item;
}
private void resize() {
    data = Arrays.copyOf(data,
        data.length * 2);
}
```



LINKED LISTS

MOTIVATION

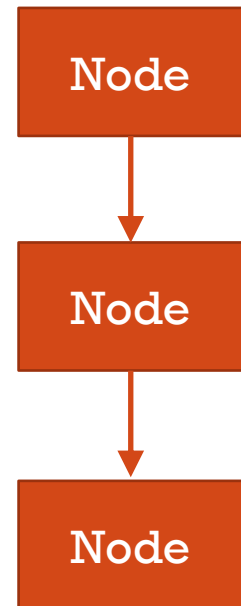
- Our existing stack implementation seems fast enough.
- You're not going to get anything faster for `isEmpty()` or `size()`!
- However, when it comes to `push()`, there is some hand waving. Most of the time it's $O(1)$, but there is a chance the array will need resizing. In that case, it's $O(n)$. Some people are fine with this, but should be careful to say that “`push()` is $O(1)$ *amortized*”.
 - (This problem would apply to `pop()` as well, if we supported reducing the size of the array.)
- What would be nice is if we could implement a `push()` and `pop()` that are truly $O(1)$... so let's do it.
- In order to do this, we'll need to use a new type of basic data structure: *linked lists*.

THE LINKED LIST CONCEPT: RECURSIVE DATA

- Here's a thought: if we were able to make a method call itself, why can't we make a class refer to itself? For example:

```
class Node {  
    Node next;  
}
```

- Node is the general term given to an object that is part of a linked list.
- This class is *self-referential*... lists are recursive!
- Notice use of the word “referential” (i.e., reference), which is referring to how objects are stored in Java.
- It would not be “technically correct” to say that the Node contains another Node, it's more correct to say that Node contains a reference to another Node.
 - What's the difference?



Three Node objects
that been created
and linked.

THE LINKED LIST CONCEPT: RECURSIVE DATA

- But how do we use that to store something? Simple: we add other pieces of data to the nodes

```
class Node {  
    int data;  
    Node next;  
}
```

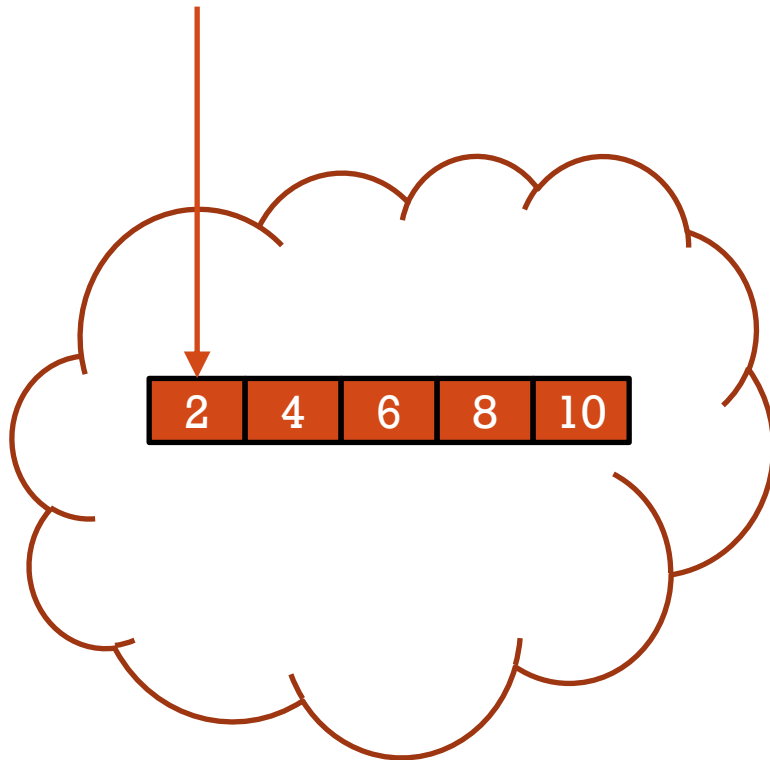
- This new data is “along for the ride” with the overall linked structure that is being formed by the Node’s next variable.



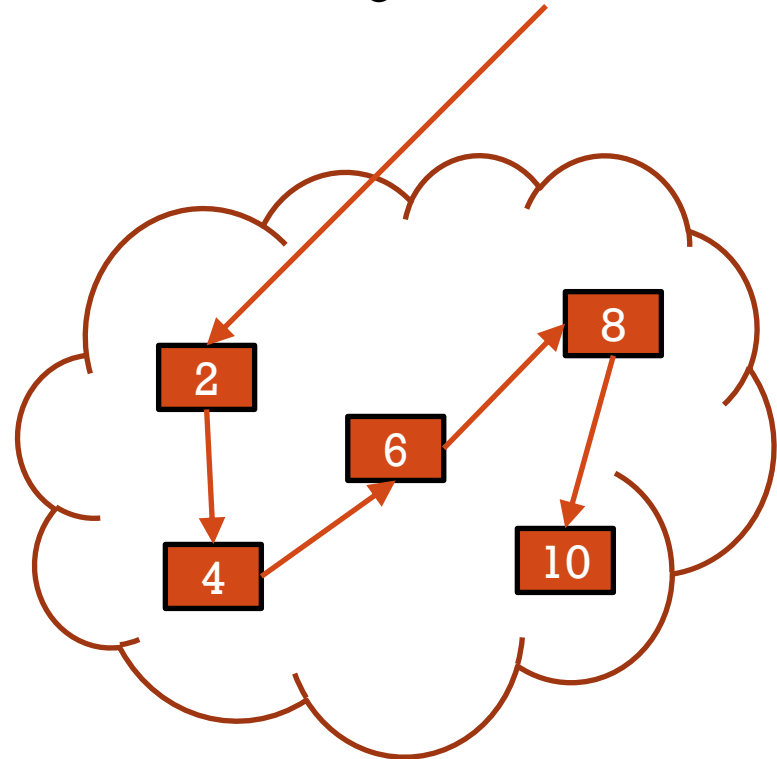
ARRAYS VS LISTS

Let's say we want to store five pieces of data in memory: 2, 4, 6, 8, 10. Let's draw, roughly how that would be represented in system memory.

`int[] data..`



`LinearNode<Integer> head...`

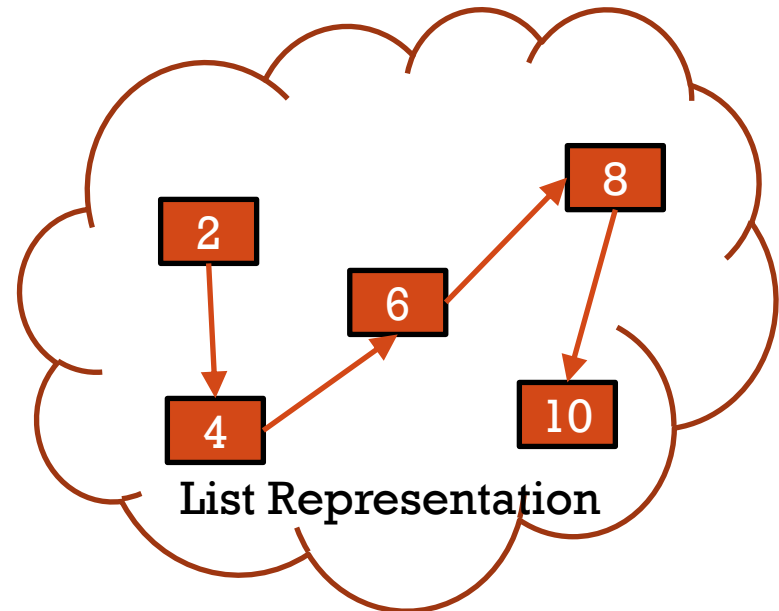
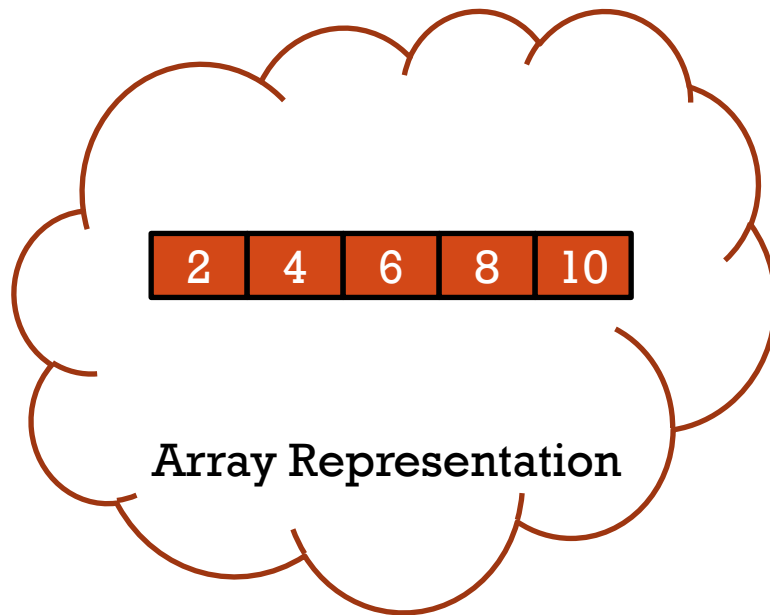


PERFORMANCE COMPARISON

Consider the following operations:

- Looping over each value and printing it out (traverse)
- Printing out the 3rd element (select)
- Using binary search to check if 8 is in the data (search)
- Adding the number 7 between 6 and 7 (insert)
- Removing the number 4 (delete)

...which data structure should be faster?



A LINKED STACK

30

TOWARDS A FASTER STACK

- At this point, we have the basic idea for a new data structure that we can use to implement a Stack. Will a list help though?
- Oftentimes, when you're implementing an ADT, you'll be faced with the question: should I use an array or a list?
- So how do we know how what to use? It depends on the operations that your ADT will frequently use... pick the data structure that supports them the fastest.
- In general, whenever you need to add or remove elements from a data structure you should consider lists. Arrays will always be $O(n)$ (worse case for resize), while lists will be better.
 - How much better?
- Our plan will be to implement a Node class, and then recreate the stack.

TYPES OF NODE IMPLEMENTATIONS

Minimalistic:

```
class Node <Item> {  
    Item element;  
    Node next;  
}
```

Monolithic:

```
class ContactNode {  
    String name;  
    int phone[30];  
    String email;  
    ContactNode next;  
}
```

OOP:

```
class Node {  
    Node next;  
}
```

```
class PersonNode extends Node  
{  
    String name;  
    int phone[30];  
    String email;  
}
```

Best: Probably whatever is used by
STDLIB's list implementation.

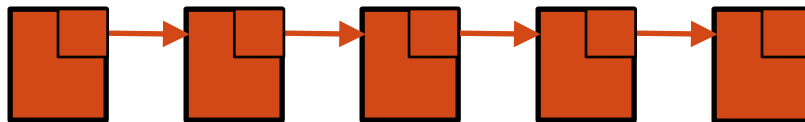
A PROPERLY ENGINEERED MINIMALISTIC NODE

```
public class LinearNode<T> {  
    private LinearNode<T> next;  
    private T element;  
  
    public LinearNode() {  
        next = null;  
        element = null;  
    }  
  
    public LinearNode(T elem) {  
        next = null;  
        element = elem;  
    }  
  
    public LinearNode<T> getNext() {  
        return next;  
    }  
  
    public void setNext(LinearNode<T> node) {  
        next = node;  
    }  
}
```

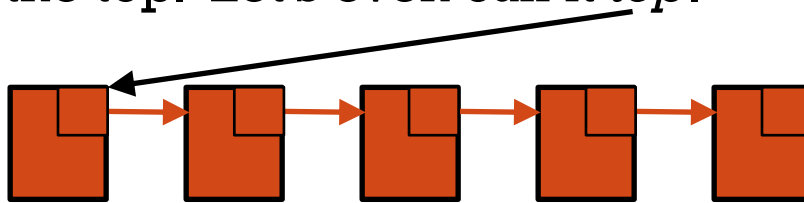
```
    public T getElement() {  
        return element;  
    }  
  
    public void setElement(T elem) {  
        element = elem;  
    }  
}
```

REPRESENTING STACK DATA

- As planned, we'll use a (singly) linked list:



- Same as last time, all activity (push/pop) happens at one place – the top. Let's even call it *top*.



- *top* will need to change as push or pop is called.
- Initially there won't be anything in the stack, so let's say it starts as *null* to indicate that.
- Later on, a push can add a list element on top of stack and pop can remove it.

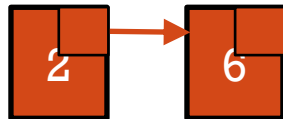
REPRESENTING STACK DATA

- Initially, top is null, so nothing exists:

- What should happen if we push?



- top is now non-null. It points at the number we added. Push again:



- top is now pointing a different node – the new top most element.
- What will pop do?

SETTING THE SCENE

- The first thing we need to do is initialize a list for our stack.
- In order for a ADT to contain a list data structure, all we need to do is create a member variable that is typed as a node.

```
private LinearNode<ItemType> head;
```
- A common variable name is *head* (or *front*), but we can use whatever we want. Remember that this variable will be pointing to the first node in the list.
 - Another one is *tail* (or *back*), which represents the last element in a list.
- Typically the variable will be set equal to null, to indicate that the structure does not contain any nodes at the beginning.

APPLYING TO LINKEDSTACK

- The class definition will be basically the same as for the array stack. We've just made a change to the generic type's name.

```
public class LinkedStack<ItemType> implements
Stack<ItemType> {

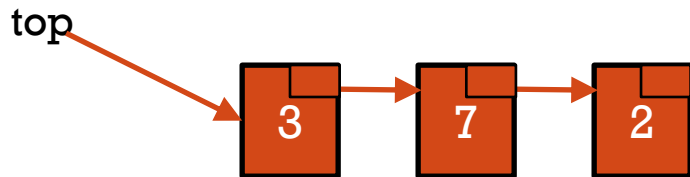
    private LinearNode<ItemType> top;
    private int n = 0;

    public LinkedStack() {
        top = null;
    }
}
```

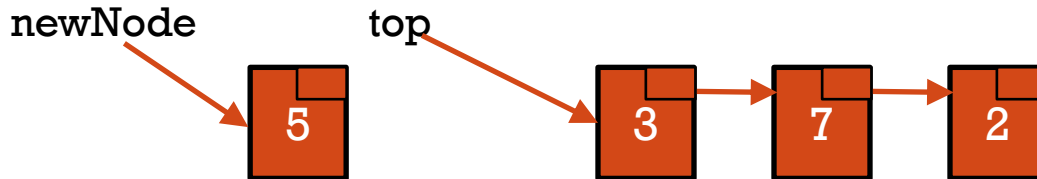
- In addition to what we discussed, there will also be a variable called n. What might this variable be used for?

ADDING A NODE TO THE FRONT

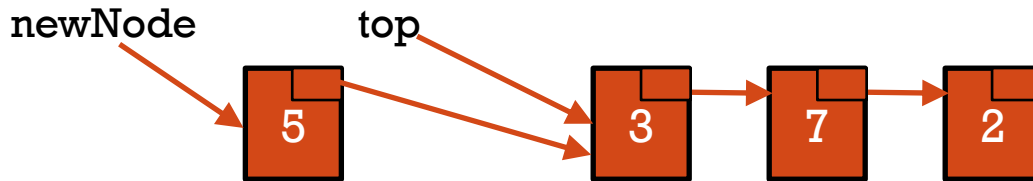
- Let's say that `top` is pointing to the most recently added node.



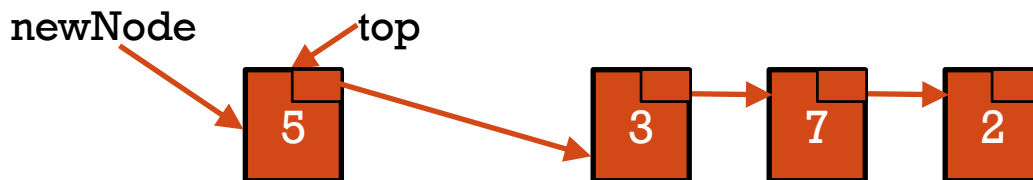
- And we get a number to add: 5. First thing is create a new node.



- With the node, we will connect it's *next* to the node `top` is pointing at.



- Finally, we'll update `top` to point at the new node. Done!



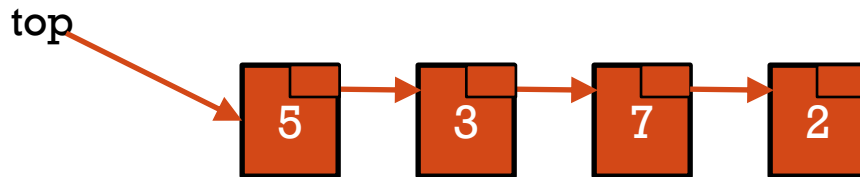
APPLYING TO LINKEDSTACK

- The code isn't so bad.
- What does each line do?

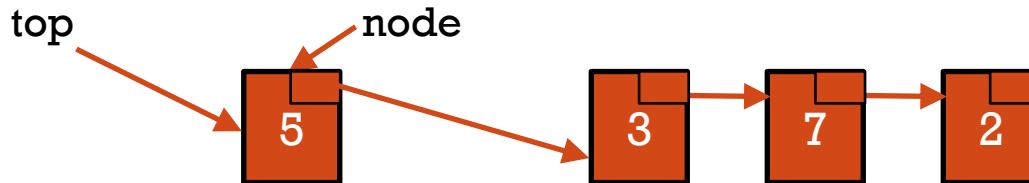
```
public void push(ItemType item) {  
    LinearNode<ItemType> node = new LinearNode<>(item);  
    node.setNext(top);  
    top = node;  
    n++;  
}
```

REMOVING A NODE FROM THE FRONT

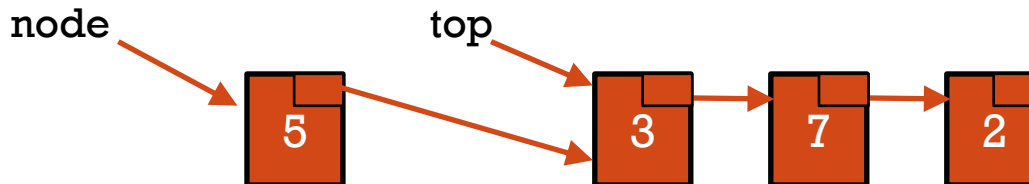
- Let's say that `top` is pointing to the most recently added node.



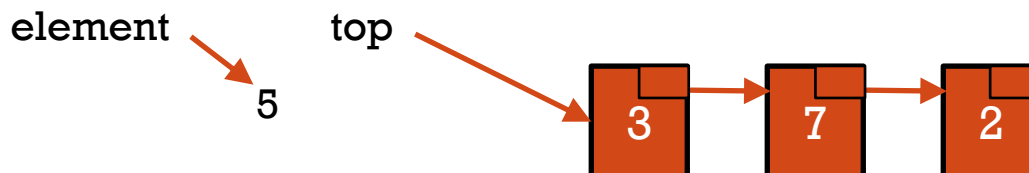
- First, we'll create a new variable to point at the top – we'll need it.



- We'll then set `top` to `top's next`, to effectively skip over a node.



- At this point the node has been removed from the list. In addition, we can use `node` to get the element that was just removed.



APPLYING TO LINKEDSTACK

- The code isn't so bad.
- What does each line do?

```
public ItemType peek() {  
    if(isEmpty())  
        throw new NoSuchElementException();  
  
    return top.getElement();  
}
```

```
public ItemType pop() {  
    if(isEmpty())  
        throw new NoSuchElementException();  
  
    ItemType element = top.getElement();  
    top = top.getNext();  
    n--;  
  
    return element;  
}
```

FINAL TOUCHES

- isEmpty and size are easy to implement.
 - Is there any other way we could implement isEmpty?
- The other feature we should look at is adding iterator support. Maybe later.

```
public boolean isEmpty() {  
    return top == null;  
}  
  
public int size() {  
    return n;  
}
```