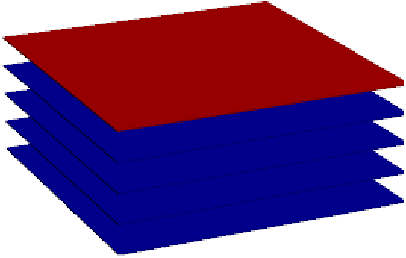
CI583: Data Structures and Operating Systems
The stack



More simple data structures

So far we have used two basic data structures: the array and the list.

Both of these are general-purpose collection types, the main difference being that lists are more suitable when you don't know in advance the exact size of a collection. Arrays are more suitable when random access to elements is required, since this is $O(1)$ for arrays and $O(n)$ for lists.

More simple data structures

In this lecture we will examine some more *specialised* data structures, designed for particular tasks: the stack, the queue and the priority queue.

In each case, the underlying storage mechanism might be an array or a list – it doesn't matter to us as users of, say, the stack. All that matters is that the stack provides the methods and capabilities we expect from a stack.

The stack

A stack is a collection type that allows us to push elements onto the front of it, pop (remove) elements from the front of it and, usually, to peek at the front element without removing it.

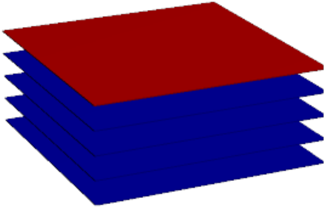
We cannot access anything other than the first element. If we want to get access to the third element, we need to call pop three times.

This method of access is called LIFO – last in, first out.

The stack

Stacks are closely linked to low-level ways of interacting with computers and are used extensively in systems programming.

Code that we write in a high-level language is compiled down to code that spends most of its time pushing and popping data from stacks. There are even stack-based programming languages such as FORTH.



The stack

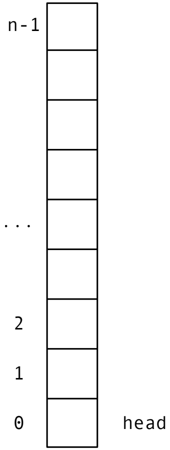
Another important application for stacks is in writing parsers, which convert raw text input (such as a source file written using a programming language) into data structures with some particular meaning.

One of the steps involved in this task is often to push each *lexical token* (in our source file these might include `if`, `else`, variable names etc) onto a stack.

The stack

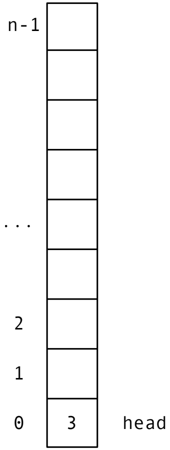
If we are implementing a stack using an array then the head of the stack isn't necessarily the first element in the array (otherwise we would have to move elements every time we pushed or popped).

Here is an empty stack with n elements.



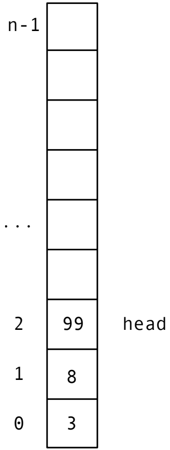
The stack

After calling `stack.push(3)`:



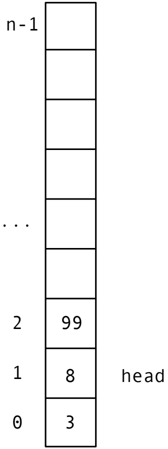
The stack

```
stack.push(8),
stack.push(99):
```



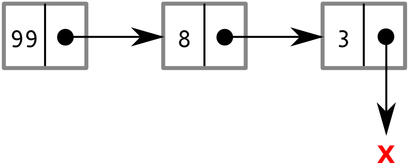
The stack

At this point, `pop()` returns 99 and moves the position of the head.



The stack

Alternatively, we could implement the stack using a list. This is simpler, since we can make push and pop just operate on the head of the list so we don't need to keep track of where the head is.



Abstract data types

A stack of ints that uses an array to store the data:

```
 class Stack {
   private int[] data;
   private int head;

   public Stack(int n) {
    data = new Object[n];
    head = -1;
  }
  public void push(int e) {
    data[++head] = e; //increment head then use its value
  }
  public int pop() {
    return data[head--]; //use head's value then decrement it
  }
  public int peek() {
    return data[head];
  }
}
```

Abstract data types

There are various things missing from this implementation – what are they?

```
  class Stack {
 private int[] data;
 private int head;

 public Stack(int n) {
  data = new Object[n];
  head = -1;
}
public void push(int e) {
  data[++head] = e; //increment head then use its value
}
public int pop() {
  return data[head--]; //use head's value then decrement it
}
public int peek() {
  return data[head];
}
}
```

Abstract data types

Or using a list...

```
class Stack {
  private LinkedList data;
  public Stack2() {
    data = new LinkedList();
  }
  public void push(int e) {
    data.cons(e);
  }
  public T pop() {
    int h = data.head();
    data = data.tail();
    return h;
  }
  public int peek() {
    return data.head();
  }
}
```

Abstract data types

This brings us to the idea of abstract data types (ADTs). The stack is defined by the ability to push, pop and peek and there are many ways we might implement one.

An ADT is a template that defines data and behaviour at an abstract level. This can be achieved using *Java generics*.

Java generics

Java generics allow us to write code that works for many (or any) types.

It is what is happening when you see Java code that uses “angle brackets”, such as `strs = new ArrayList<String>()`.

In the docs for the `ArrayList` class, you will see it described as `ArrayList<T>`. That means `ArrayList` is a container for objects of *any type*, which we call `T`.

When we create an `ArrayList` we have to say what type of thing we want to store in it, i.e. what is the type `T`.

Abstract data types

Using generics we can create a Stack that works for any type:

```
public class Stack<T> {
  //...
public void push(T e) { ... }
public T pop() { ... }
public T peek() { ... }
}

//

Stack<Integer> myIntStack = new Stack<Integer>();
myIntStack.push(42); //OK
myIntStack.push(‘‘Hi!’’); //compile-time error
```

Balancing parens

As a demonstration of the usefulness of stacks, consider the task of ensuring that all parentheses are nicely balanced in a piece of text. So “{([()])}” is balanced but “{()” is not, because parens are not all closed, and neither is “([(]))”, because the nesting is wrong.

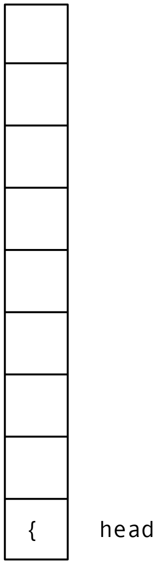
We can model this problem with a stack. Every time we encounter an opening paren character, push it onto a stack.

Every time we encounter a closing paren, pop the stack and check that the types match.

Balancing parens

{([])}

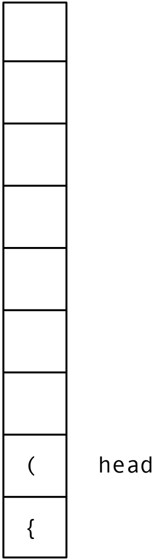
`push('{')`



Balancing parens

{([])}

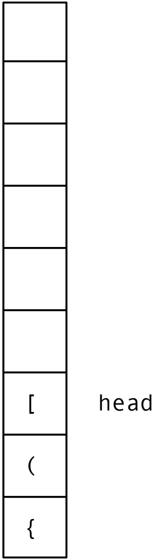
```
push('(')
```



Balancing parens

{([])}

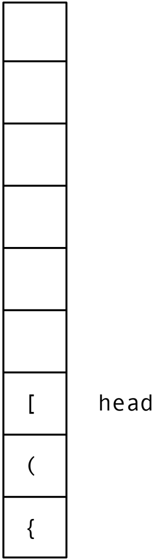
```
push('[')
```



Balancing parens

{([])}

`pop()=='['`



[head

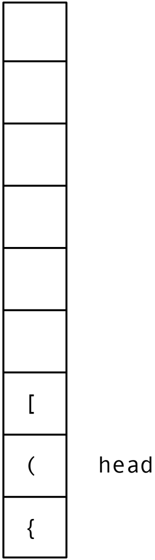
(

{

Balancing parens

{([])}

```
pop()==’(’
```



[

(

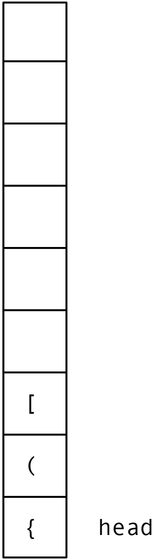
{

head

Balancing parens

{([])}

```
pop()=='{'
```

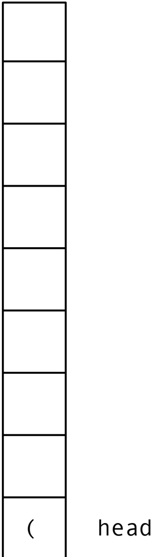


Balancing parens

An unbalanced example.

([(]))

```
push(’(’)
```



(

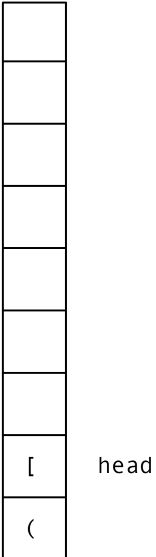
head

Balancing parens

An unbalanced example.

([(]))

```
push('[')
```

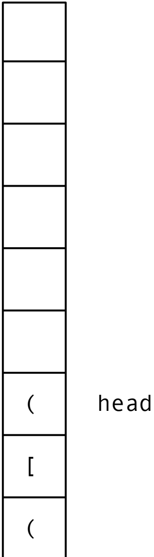


Balancing parens

An unbalanced example.

([(]))

```
push('(')
```



(head

[

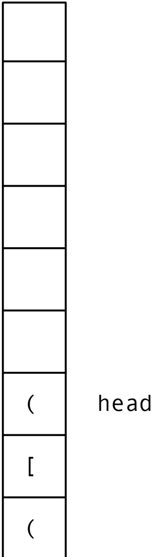
(

Balancing parens

An unbalanced example.

([(]))

```
pop()!=’[’
```



(

head

[

(