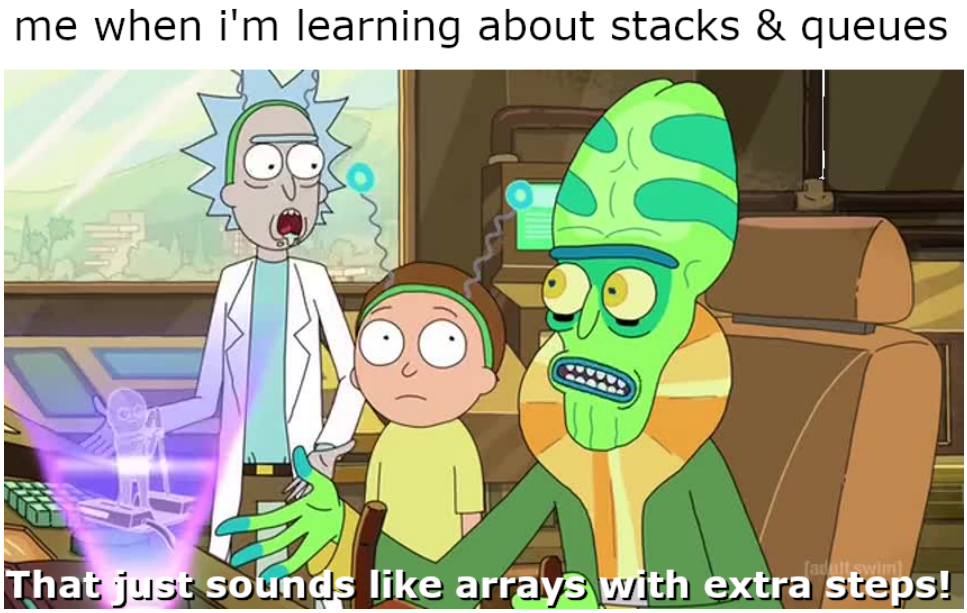
Week 4



Announcements

- Good job on the midterm!
- HW 2 is up
  - due this **Tuesday**, 4/27

Questions?

- anything?
- note: you guys might not have seen breadth/depth first search yet, but will be very familiar with them by the end of HW 2

Stacks

```
// last in, first out aka LIFO
// type of search? - depth first search
```

```
// some stl api functions
#include <stack>

std::stack<int> myStack

myStack.size();    // 0
myStack.empty();   // True

myStack.push(1);  //   top |      1  | bottom
myStack.push(2);  //   top |    2 1  | bottom
myStack.push(3);  //   top |  3 2 1  | bottom

myStack.empty();   // False

myStack.top();     // 3

// pop() returns void, so you need to use top() first to get the value
// could be different in other libraries/languages

myStack.pop();     //   top |    2 1  | bottom
myStack.pop();     //   top |      1  | bottom

myStack.top();     // 1
```

```
// fyi, this isn't a complete Stack class implementation
class Stack {
  public:
    //...
    int  top();
    void pop();
    bool push(int v);
  private:

    int arr[100];
    int size = 0;
    // ...
}

// todo: where is the bottom and top in the array below?
// making the Top the back of the array is best so we don't have to keep
// shifting elements every time we pop or push

//     Bottom  [ 1, 2, 3, 4 ] Top

// todo - implement the functions below
int Stack::top() {
  if (size > 0)
    return arr[size-1];
```

```
  // return err if empty Stack (or its up to the caller to make sure the
  // (stack isn't empty
}

void Stack::pop() {
  size--;
}

// we can simply insert to position 'size' because it points to the
// next open spot
bool Stack::push(int val) {

  if (size < 100) {
    arr[size] = val;
    size++;
    return true;
  }
  return false;
}
```

Infix and Postfix

```
Infix   - operator is between operands
Postfix - operator is after   operands

// todo: which is which?
// ((5 + 4) * 3) / 2)    infix
// 5 4 + 3 * 2 /         postfix

// why use postfix at all? - unambiguous, don't need parentheses
// convert infix to postfix? - in Carey's notes

// todo - eval this (#6 on worksheet)
9 5 * 8 - 6 7 * 5 3 - / *
45 8 - 6 7 * 5 3 - / *
37 6 7 * 5 3 - / *
37 42 5 3 - / *
37 42 2 / *
37 21 *
777

// todo - evaluate a postfix expression (write pseudocode)
char arr = ['5', '4', '+', '3', '*', '2', '/' ]

// (hint) what data structure should we use?
use a stack
for element in array
  if operand
```

```
    push to stack

  else if operator
    //top() then pop() is how you'd access the top element and continute iterating
    int num1 = top()
    pop()
    int num2 = top()
    pop()

    do operation, push result to stack

result = top()
```

Queues

```
// first in, first out - FIFO
// type of search? Breadth First Search

#include <queue>

std::queue<int> myQueue;

myQueue.empty();
myQueue.size();

myQueue.push(1);
myQueue.push(2);

myQueue.front();
myQueue.back();

myQueue.push(3);

myQueue.pop();
```

```
// can implement a queue with an array or linked list
// [1, 2, 3]
// 1 -> 2 -> 3

// Worksheet #1
Given a string of '(', ')', '[', and ']',
write a function to check if the input string is valid.

Validity is determined by each '(‘ having a corresponding ')',
```

```
and each '[' having a corresponding ']',
with parentheses being properly nested and brackets being properly nested.

Examples:
            “[()[[([][])]]]” → Valid
            “(((((])))))“ → Invalid
            “((()))” → Invalid
            “()[]” → Valid
            "([)]" -> Invalid

// which datastructure? - stack
// complete the code on your own
bool isValid(string parens) { // Fill in code here
  std::stack<> s; // what type does the stack hold?

  for () {      // iterate through string
    // if open, push

    // if closed, pop
      // if stack is empty, not valid
        return false;
      // check if the popped bracket is of the same type as current one
  }
}
```

Inheritance

- we'll do inheritance and polymorphism next week, but here's a basic introduction to the syntax and why we use them

```
// re-use code, make things simpler, fewer bugs

// "is a"  - superclass/baseclass subclass
// "has a" - member variable

Class Rick  { /* ... */ };
Class Morty { /* ... */ };

Rick  ricks[100]
Morty mortys[100]

/*
there's no easy way for me to loop through all of my characters
I need to write a different loop for each class type, which can get annoying

loop through ricks
  rick.talk
```

```
loop through mortys
  morty.talk
*/

// instead, we put the common functionality in a superclass, 'Character'
class Character: {
  public:
    void talk() { std::cout << myPhrase << std::endl; }
  private:
    string myPhrase;
}

// now our Rick and Morty classes inherit from class Character
class Rick: public Character {
  // ...
}

class Morty: public Character {
  // ...
}

// and now we can do this, which is cleaner
Character arr[2]
Morty m;
Rick r;
arr[0] = r;
arr[1] = m;

for (i = 0; i < 2; i++)
  arr[i].talk();
```