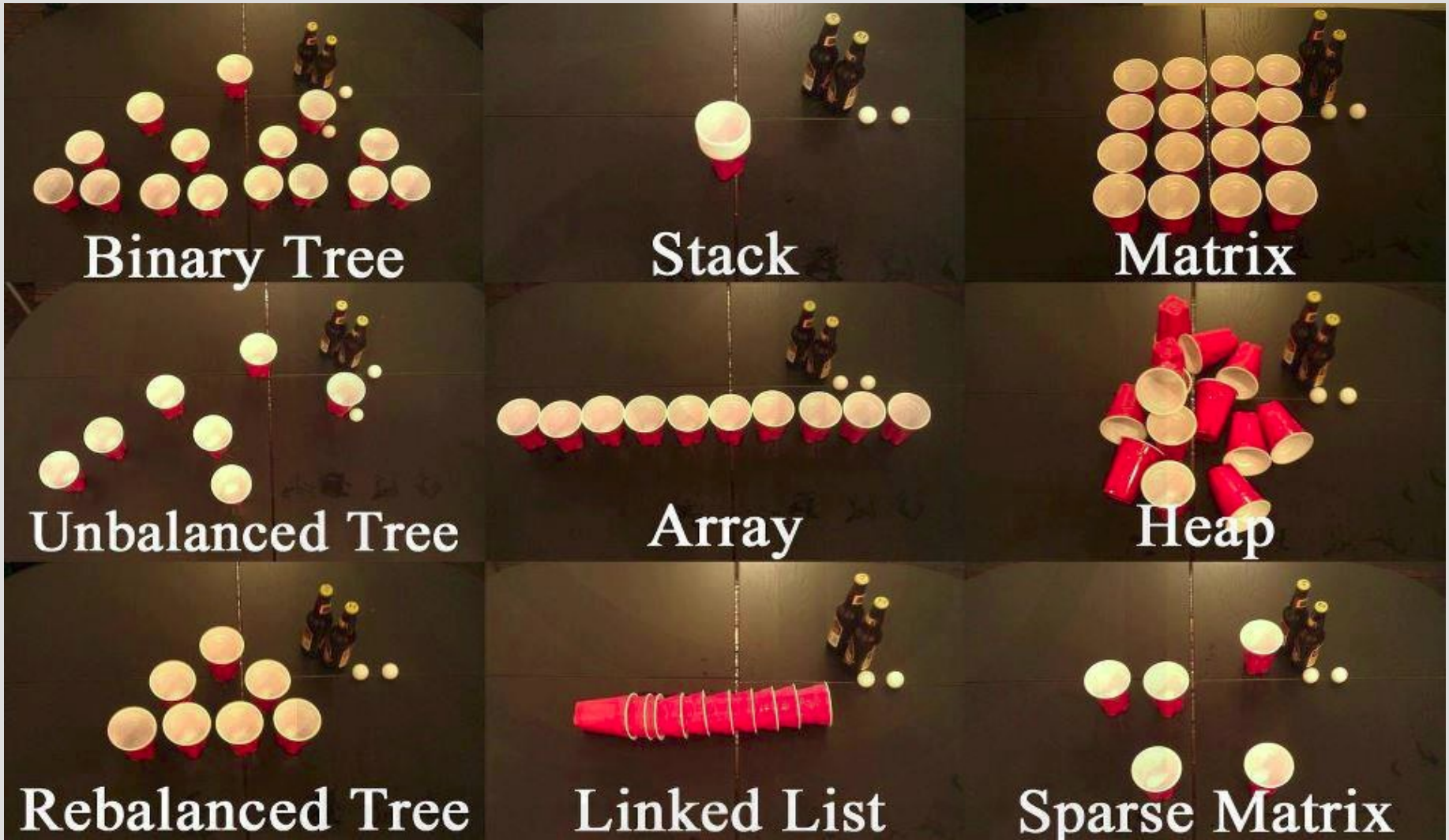
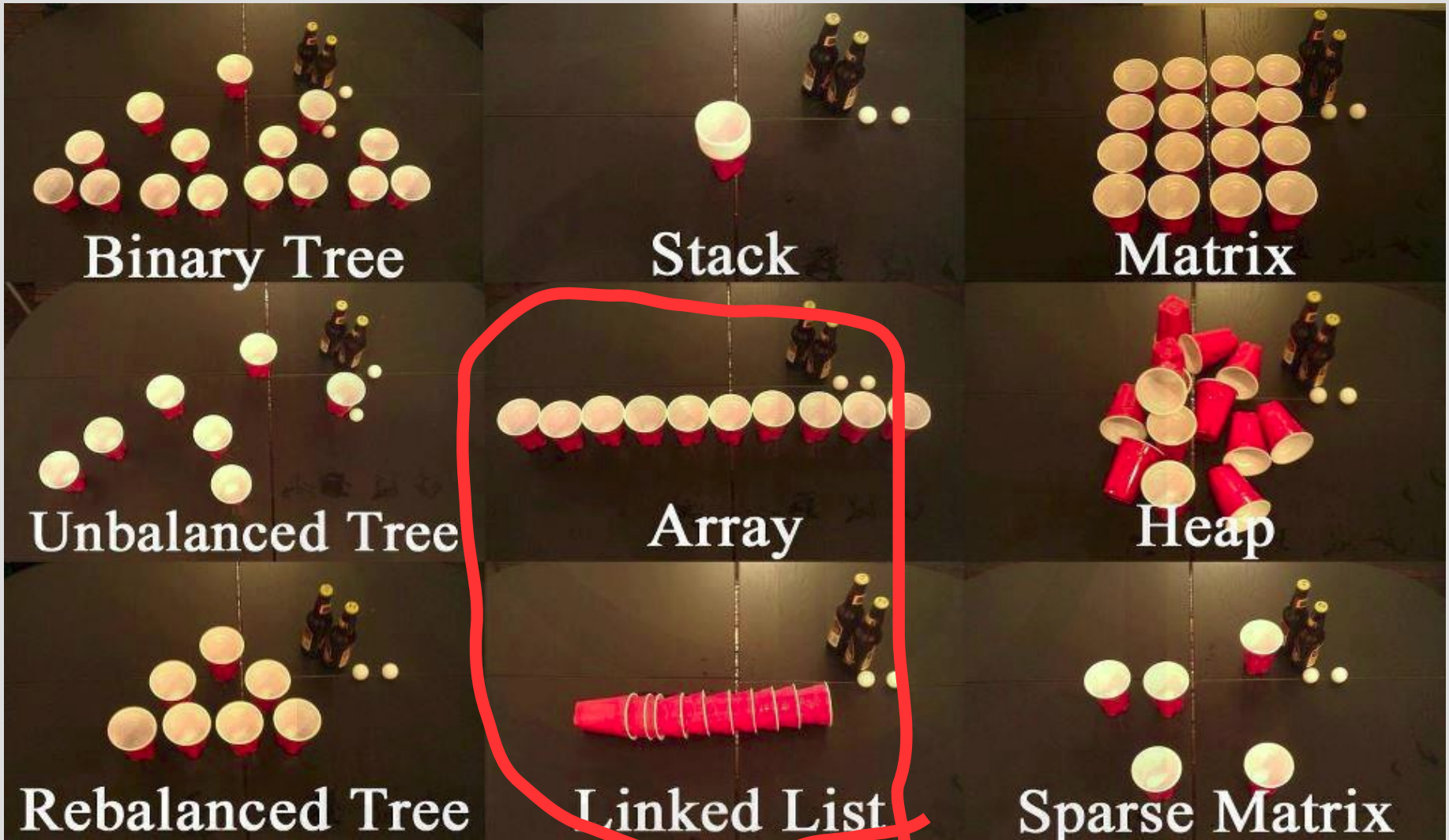
# Data structures

## Ch ???



# Highlights


Binary Tree
Stack
Matrix
Unbalanced Tree
Array
Heap
Rebalanced Tree
Linked List
Sparse Matrix

# Arrays

What are some properties of arrays?

Pros:

Cons:

# Arrays

What are some properties of arrays?

Pros:

1. Instantly access any spot
2. Built into many languages (C++ too!)

Cons:

1. Hard to add to end (sorta can)
2. Fixed size/length
3. Inserting in the middle is very annoying

# Arrays

Partially filled arrays can get around inserting at end issue

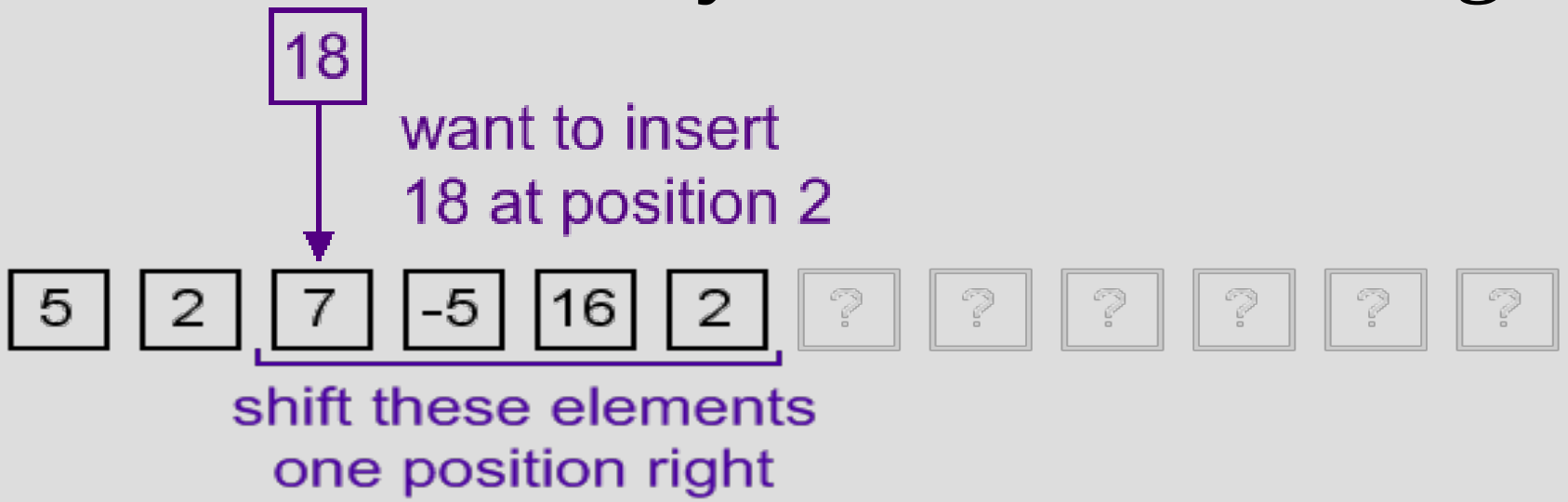
Dynamically created arrays can get around fixed size issue in conjunction with above

(if out of space, ask for new array twice as big and copy over old stuff)

(see: arrays.cpp)

# Arrays

We can never really fix insertion though...




The biggest problem is inserting depends on how many other things are already in the array

So in big arrays inserting is much harder!

# Useful templates?

Last time we talked about how you can pass types into function/classes much like how we used to pass variables

One of the downsides of this is that general types are harder to work with

For example, multiplication works for ints and double but not for string

# Useful templates?

There are a couple very general places where templates are useful:

1. swapping (did already)
2. converting (static_cast)
3. storing

Much like arrays, we want them to be able to store any type of data

# vector class

Normal arrays have multiple issues:
(1) cannot grow (have to do partially filled)
(2) cannot insert (have to shift)

However, there is a class that does these things for you automatically called “vector”

```
#include <vector>
```

It also uses templates so you can store any type (much like normal arrays)
(see: vector.cpp)

# vector class

Useful vector functions:

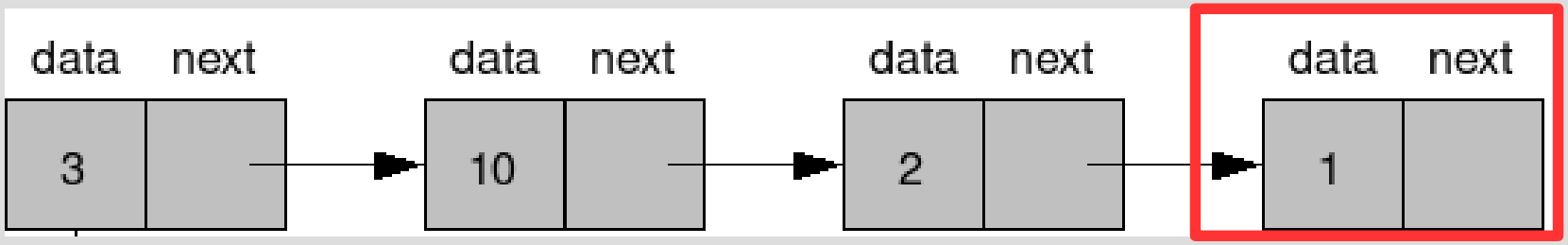
- push_back(array_type) - adds this element to back of array
- at(int) or [int] - index into the array at this index
- size() - how many elements there are now
- insert(iterator, array_type) - inserts an element at iterator's spot (shifts current element and all later down one)
- erase(iterator) - removes an element

# Linked List

We get around this problem with pointers

Instead of using an array directly, we will make a mini-class with a single int and a pointer to another type as itself

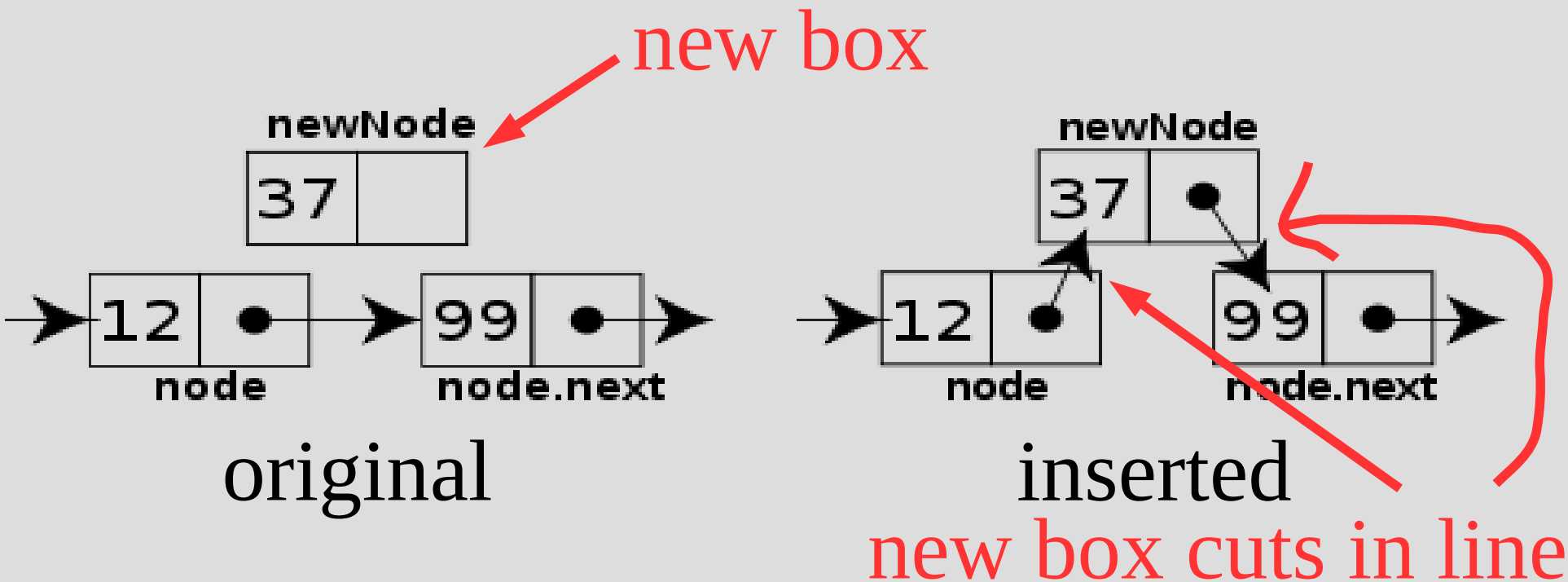
class “item”




(see: linkedList.cpp)

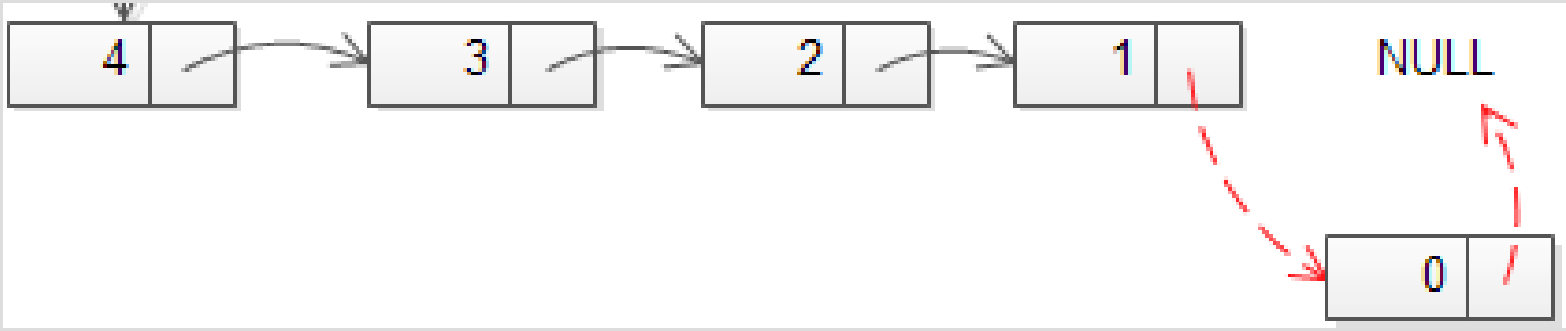
# Linked List

To insert an item into a linked list, we simply need to make a new box, then change where the previous and new box are pointing



# Linked List

To add to the end of a linked list, we first need to go to the end, then simply add a new box here and change the last link




To find the last item, we keep checking “next” then moving to it, until “next” is nullptr (a pointer to nowhere/the abyss)

# Linked List vs Arrays

Linked lists have very easy inserts (you don't need to shift anything, thanks to pointers!)

However, access time is not fast...

If you want the last item, you would need to go through all others (one at a time)

**Normal array: slow insert, fast access**
**Linked list: fast insert, slow access**