Data structures & algorithms

Tutorial 3

Lesson overview

- Recap of important topics from last week
- Introduction to templates and templated functions
- Improving MyVector
- Linked Lists
  - Singly Linked Lists
  - Types of linked lists
- Templating MyVector

A quick recap of important things from last week

std::vector

```
std::vector<type> myVector{};
```

- A `vector` is essentially C++'s version of an ArrayList from Java. Once again it is found in the standard library and can be included within your cpp file with `#include <vector>`
- For those unfamiliar with ArrayList's, they are a resizable array that have their elements stored in contiguous locations. Which allows element access via an index.
- Replacing `type` with a type of your choosing will only allow storage of that type only. I.e., `std::vector<int> myVector{};` will only ever be able to store `int`'s
- We also implemented our own vector in our MyVector class that can resize, which used a C-style array as the underlying structure

std::vector

Main operations for `std::vector`

```
std::vector<int> myVector{4, 5, 9, 10}; // Initialize with values
```

- `myVector[n]`: accesses element at index `n`
  - `myVector[0] // returns 4 (element at the 0th index)`
- `myVector.at(n)`: accesses element at index `n`
  - `myVector.at(2) // returns 9 (element at the 2nd index)`
- `myVector.front()`: accesses element at the front of the vector
  - `myVector.front() // returns 4`
- `myVector.back()`: accesses element at the back of the vector
  - `myVector.back() // returns 10`
- `myVector.push_back(value)`: pushes `value` to the back of the vector
  - `myVector.push_back(13) // vector now contains {4, 5, 9, 10, 13}`
- `myVector.pop_back()`: removes element at the end of the vector
  - `myVector.pop_back() // vector now contains {4, 5, 9}`
- `myVector.size()`: returns the size of the vector
  - `myVector.size() // returns 9 (element at the 2nd index)`
- `myVector.empty()`: empties the vector
  - `myVector.empty() // vector now contains {}`

std::vector

Advanced operations for `std::vector`

```
std::vector<int> myVector{4, 5, 9, 10}; // Initialize with values
```

- `myVector.begin()`: access the element at the beginning of the vector as an iterator
  - `*(myVector.begin()) // returns 4`
- `myVector.end()`: access the element one past the end of the vector as an iterator
  - `*(myVector.end()) // returns 0 since it is one past the last element, we can get the last element with *(myVector.end() - 1)`
- `myVector.rbegin()`: accesses reverse iterator to reverse beginning
  - Is essentially `myVector.end()` but makes iterating through the vector in reverse easier
- `myVector.rend()`: accesses reverse iterator to reverse end
  - Is essentially `myVector.begin()` but makes iterating through the vector in reverse easier

std::vector

Looping through an `std::vector`

For loop

```
for(int i = 0; i < myVector.size(); i++) {
    std::cout << myVector.at(i) << std::endl
}
```

Range based for loop

```
for(int i : myVector) {
    std::cout << i << std::endl
}
```

For loop with iterators

```
for(auto i = myVector.begin(); i != myVector.end(); i++) {
    std::cout << *i << std::endl
}
```

Lab time 😎

Templates

```
int doubleValue(int a) {
    return a * 2;
}
```

- Templates are similar to Java’s generics except better, they allow you to define a blueprint for a class or function in which the user can use any type
- Consider this `doubleValue` function, currently it only accepts `int`’s … But what if we want to double the value of a `double`?

Templates

```
int doubleValue(int a) {
    return a * 2;
}

double doubleValue(double a) {
    return a * 2;
}
```

- Templates are similar to Java’s generics except better, they allow you to define a blueprint for a class or function in which the user can use any type
- Consider this `doubleValue` function, currently it only accepts `int`’s … But what if we want to double the value of a `double`? Then we’d need to create a completely new function to do so. And if we wanted to double a more types, we’d need even more code!

Templates

```
int doubleValue(int a) {
    return a * 2;
}
double doubleValue(double a) {
    return a * 2;
}
```

- Templates are similar to Java's generics except better, they allow you to define a blueprint for a class or function in which the user can use any type
- Consider this `doubleValue` function, currently it only accepts `int`'s … But what if we want to double the value of a `double`? Then we'd need to create a completely new function to do so. And if we wanted to double a more types, we'd need even more code!
- We can instead make `doubleValue` a `template` function, and allow it to accept a generic type as a parameter.

Templates

```
template <typename T>
T doubleValue(T a) {
   return a * 2;
}
```

- Templates are similar to Java’s generics except better, they allow you to define a blueprint for a class or function in which the user can use any type
- Consider this `doubleValue` function, currently it only accepts `int`’s … But what if we want to double the value of a `double`? Then we’d need to create a completely new function to do so. And if we wanted to double a more types, we’d need even more code!
- We can instead make `doubleValue` a `template` function, and allow it to accept a generic type as a parameter.

Templates

```cpp
template <typename T>
T doubleValue(T a) {
   return a * 2;
}
```

- The T can be anything you want it to be…

Templates

```
template <typename cat>
cat doubleValue(cat a) {
  return a * 2;
}
```

- The T is the generic type, and it can be called anything you want it to be … cat

Templates

```
template <typename guitar>
guitar doubleValue(guitar a) {
   return a * 2;
}
```

- The T is the generic type, and it can be called anything you want it to be… cat, guitar

Templates

```
template <typename dsa>
dsa doubleValue(dsa a) {
   return a * 2;
}
```

- The T is the generic type, and it can be called anything you want it to be… cat, guitar, dsa…etc.

Templates

```
template <typename T1, typename T2...>
T1 doubleValue(T1 a, T2 b) {
    std::cout << b << std::endl;
    return a * 2;
}
```

- The T is the generic type, and it can be called anything you want it to be… cat, guitar, dsa…etc.
- You are also allowed to include as many generic types as you want! Which can become extremely useful when creating a templated class.

Templates

```
template <typename T>
T doubleValue(T a) {
   return a * 2;
}
```

- The `T` is the generic type, and it can be called anything you want it to be… `cat`, `guitar`, `dsa`…etc.
- You are also allowed to include as many generic types as you want! Which can become extremely useful when creating a templated class
- When calling the `doubleValue` function, you can explicitly specify you type you would like… `int myValueDoubled = doubleValue<int>(10)`

Templates

```
template <typename T>
T doubleValue(T a) {
   return a * 2;
}
```

- The T is the generic type, and it can be called anything you want it to be… cat, guitar, dsa…etc.
- You are also allowed to include as many generic types as you want! Which can become extremely useful when creating a templated class
- When calling the doubleValue function, you can explicitly specify you type you would like…
  `int myValueDoubled = doubleValue<int>(10)`
- You've actually been using template everytime you call `std::vector<int> myVector{}`

Give “Templated functions” a go to see how they work!

Copy constructor

```
int main() {
    MyVector myVector{1, 2, 3, 4};
    MyVector shallowCopy{myVector};




    return 0;
}
```

- We will be revisiting MyVector from last week and improving it. What we will be implementing is a copy constructor and a copy assignment operator to make a deep copy of another MyVector.

Copy constructor

```
int main() {
    MyVector myVector{1, 2, 3, 4};
    MyVector shallowCopy{myVector};
    shallowCopy[0] = 10;



    return 0;
}
```

- We will be revisiting MyVector from last week and improving it. What we will be implementing is a copy constructor and a copy assignment operator to make a deep copy of another MyVector.
- There is a primary difference between a shallow copy and a deep copy. A shallow copy is very similar to a reference, in the fact that the if you alter the shallow copy…

Copy constructor

```cpp
int main() {
   MyVector myVector{1, 2, 3, 4};
   MyVector shallowCopy{myVector};
   shallowCopy[0] = 10;
   myVector[0]; // Returns 10
   shallowCopy[0]; // Returns 10
   return 0;
}
```

- We will be revisiting MyVector from last week and improving it. What we will be implementing is a copy constructor and a copy assignment operator to make a deep copy of another MyVector.
- There is a primary difference between a shallow copy and a deep copy. A shallow copy is very similar to a reference, in the fact that the if you alter the shallow copy… the original gets altered as well. The only difference is a shallow copy creates a completely new object, whilst a reference creates an alias to an existing one.

Copy constructor

```cpp
int main() {
    MyVector myVector{1, 2, 3, 4};
    MyVector deepCopy{myVector};
    deepCopy[0] = 10;
    myVector[0]; // Returns 1
    deepCopy[0]; // Returns 10
    return 0;
}
```

- Whilst a deep copy also creates a new object but copies all the data over into a new location that won’t affect the original variable.
- The goal for the next exercise is the implement your copy constructor and operator assignment function to create a deep copy of another MyVector
- Hint: the copy constructor will require you to use a loop, whilst the operator function will require you to use `std::swap()`

Give “Improving MyVector” a go to implement those functions!

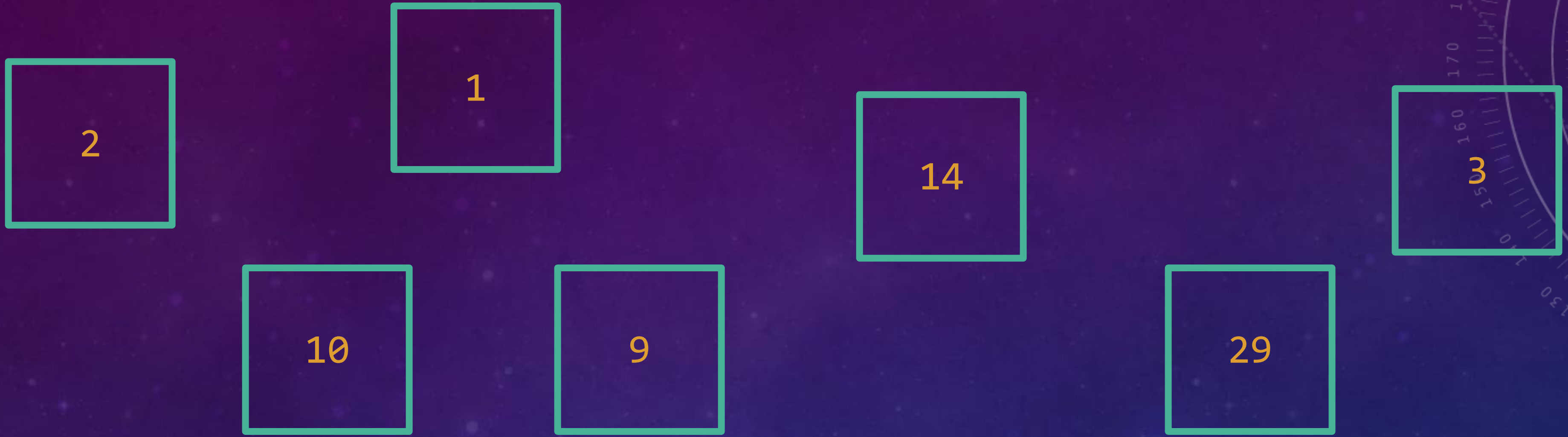
Singly linked list

- A singly linked list is a data structure that closely resembles a vector (ArrayList).

Singly linked list

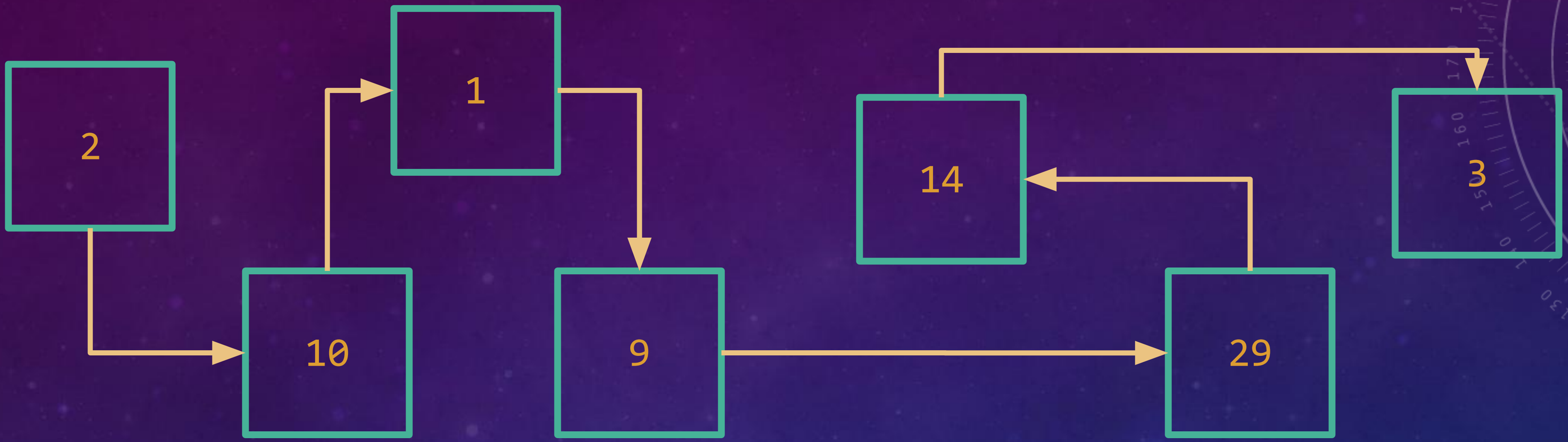
- A singly linked list is a data structure that closely resembles a vector (ArrayList). Rather than a contiguous block of memory storing each piece of data like a vector.

Singly linked list



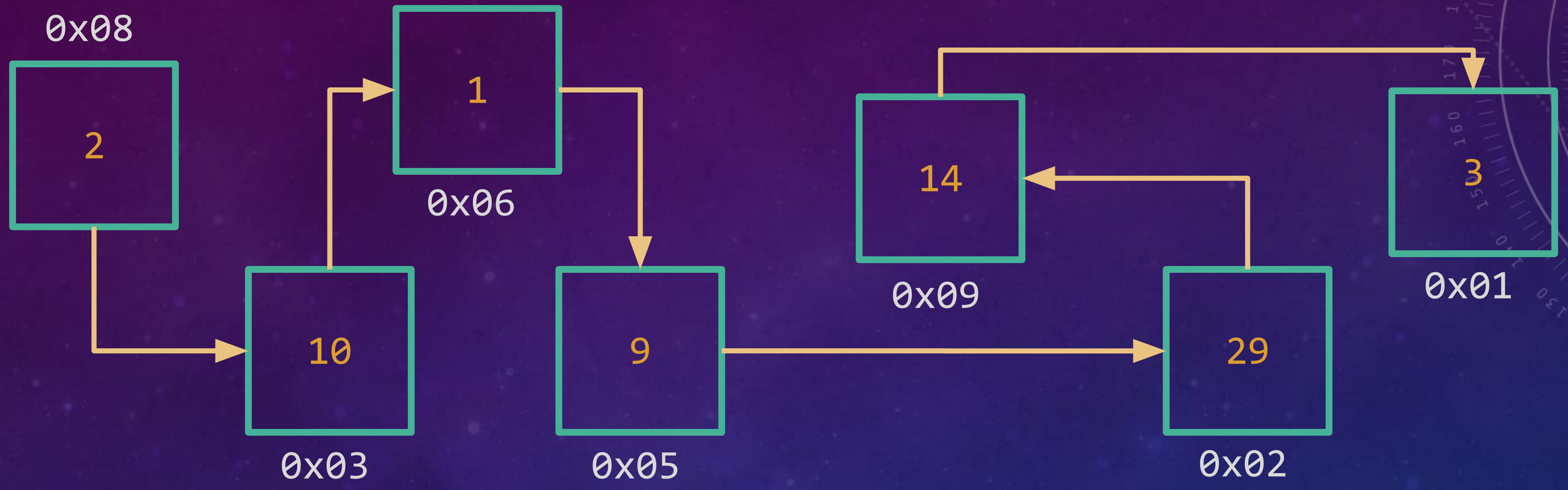
- A singly linked list is a data structure that closely resembles a vector (ArrayList). Rather than a contiguous block of memory storing each piece of data like a vector. A singly linked list stores nodes for each piece of data…

Singly linked list



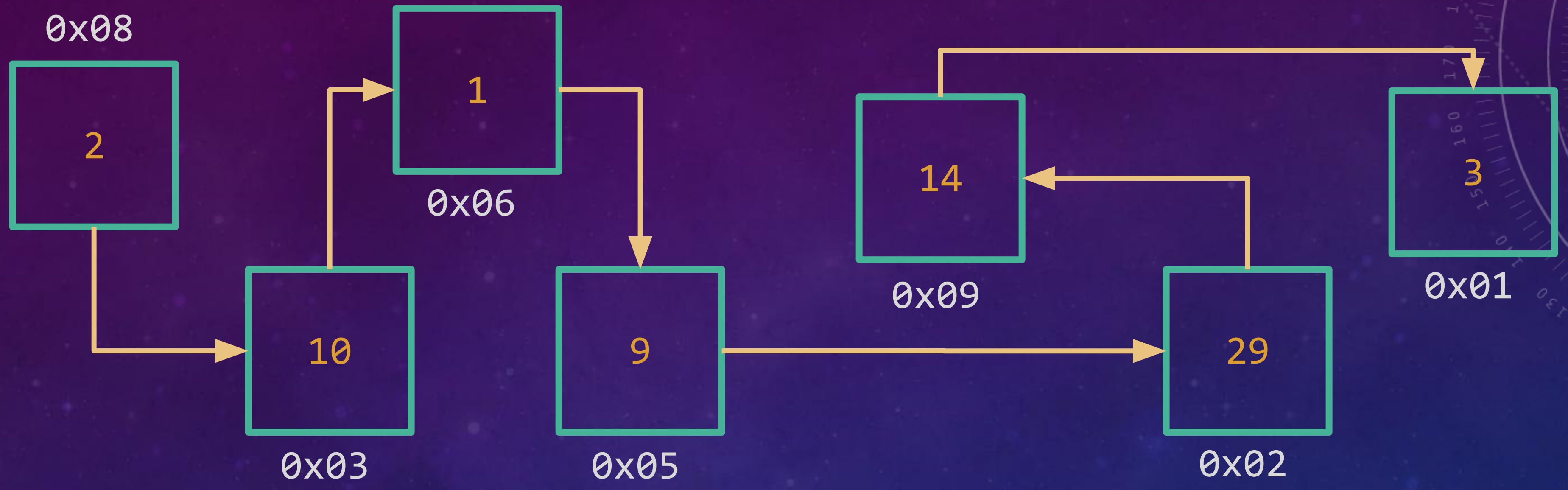
- A singly linked list is a data structure that closely resembles a vector (ArrayList). Rather than a contiguous block of memory storing each piece of data like a vector. A singly linked list stores nodes for each piece of data… each node is then linked together, with the previous node pointing at the next one.

Singly linked list



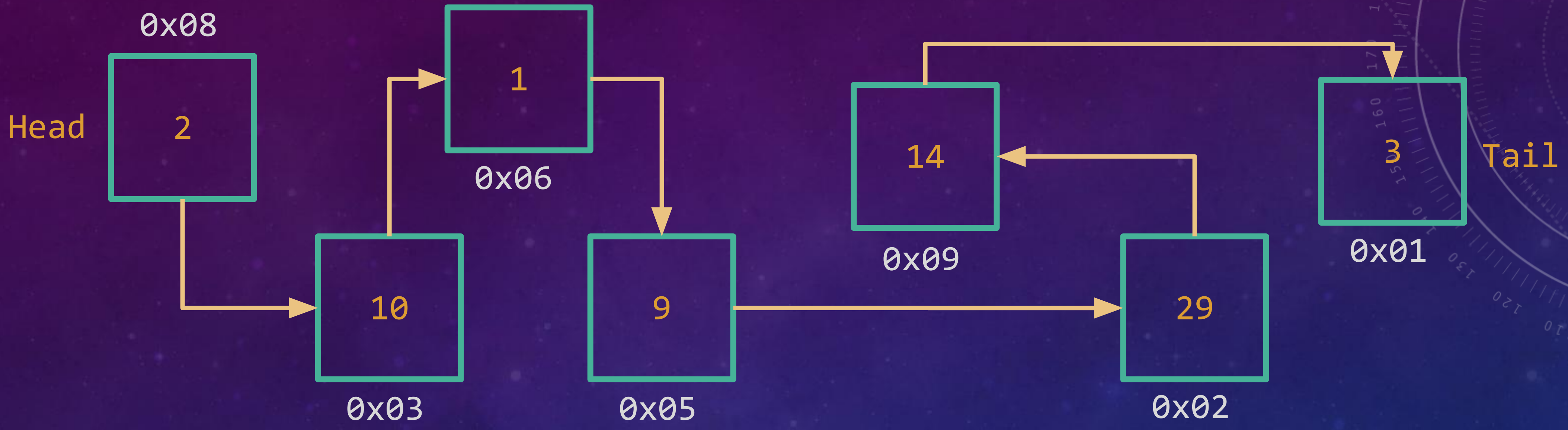
- A singly linked list is a data structure that closely resembles a vector (ArrayList). Rather than a contiguous block of memory storing each piece of data like a vector. A singly linked list stores nodes for each piece of data… each node is then linked together, with the previous node pointing at the next one.
- Since each piece of data is stored in its own node, they aren't stored in a contiguous line. Rather in random areas on the RAM.

Singly linked list



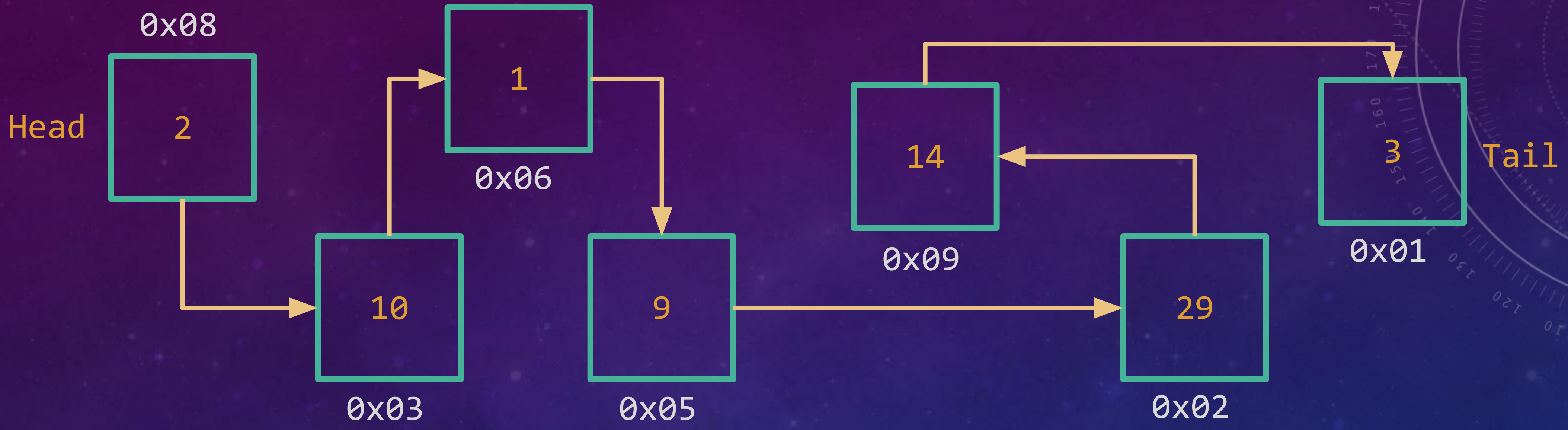
- A singly linked list is a data structure that closely resembles a vector (ArrayList). Rather than a contiguous block of memory storing each piece of data like a vector. A singly linked list stores nodes for each piece of data… each node is then linked together, with the previous node pointing at the next one.
- Since each piece of data is stored in its own node, they aren't stored in a contiguous line. Rather in random areas on the RAM. Because of this, this means singly linked lists are not accessible via an index like vectors. Which makes element access O(n) time complexity, rather than O(1)

Singly linked list



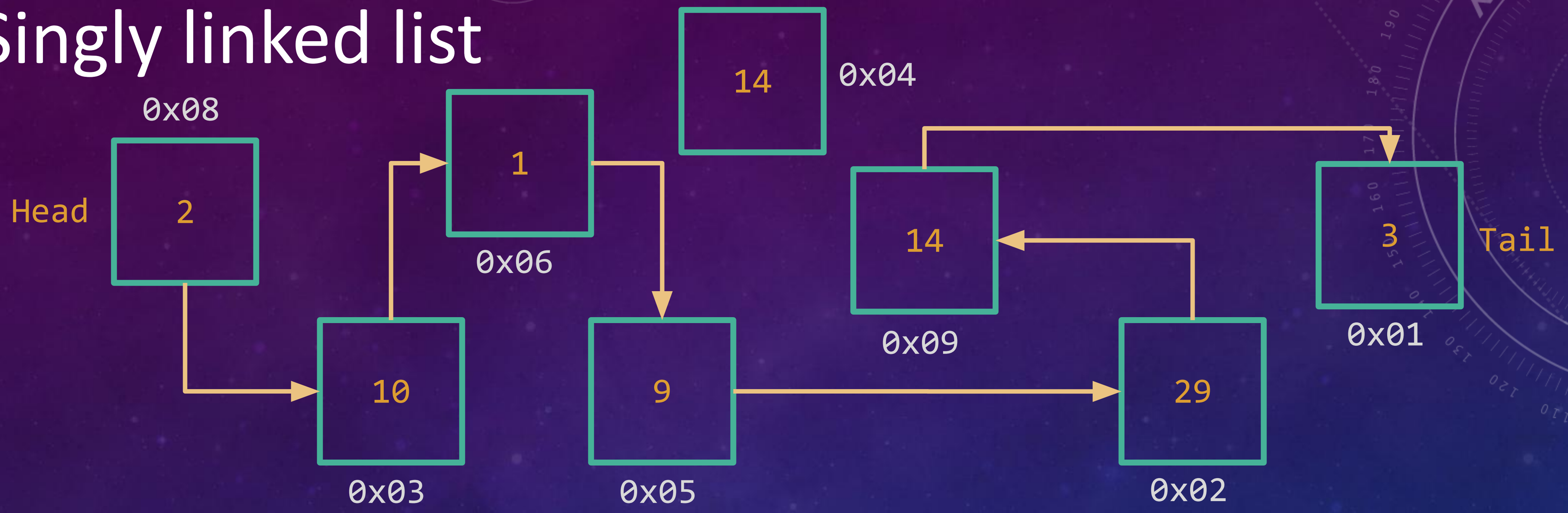
- A singly linked list is a data structure that closely resembles a vector (ArrayList). Rather than a contiguous block of memory storing each piece of data like a vector. A singly linked list stores nodes for each piece of data… each node is then linked together, with the previous node pointing at the next one.
- Since each piece of data is stored in its own node, they aren't stored in a contiguous line. Rather in random areas on the RAM. Because of this, this means singly linked lists are not accessible via an index like vectors. Which makes element access O(n) time complexity, rather than O(1)
- Sometimes linked lists will also have a variable for the head, and tail of the linked list. But not always! Which will allow O(1) time complexity to retrieve either value and insert or delete at either end.

Singly linked list



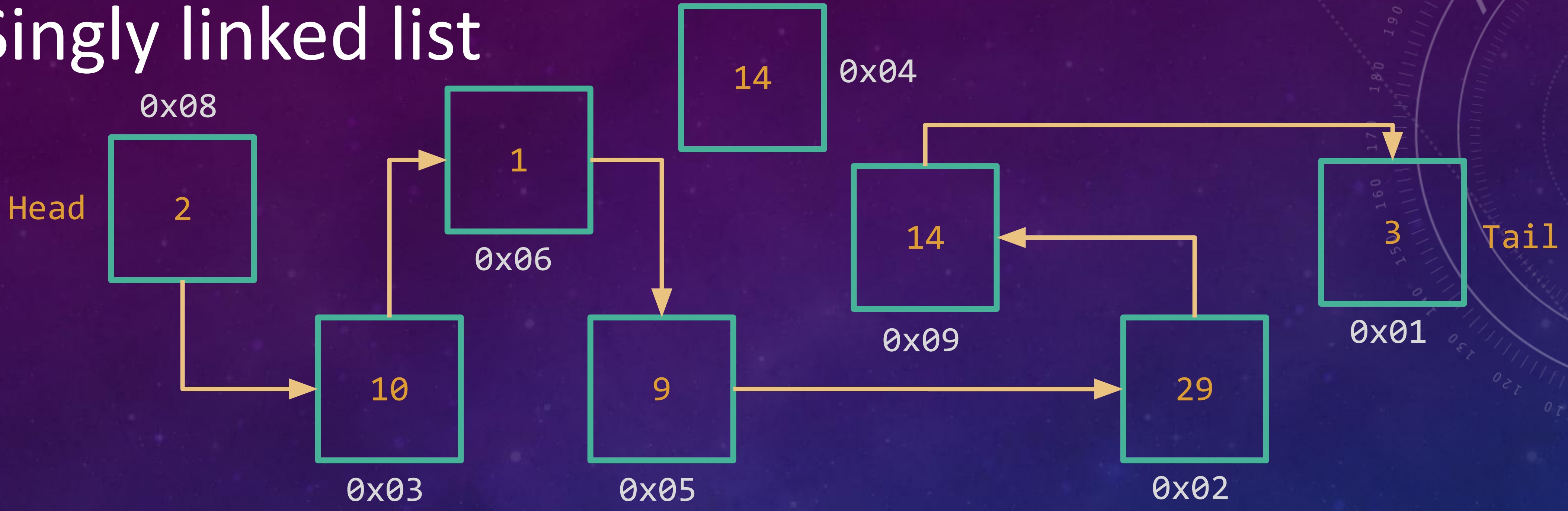
- A common reason why people would choose a linked list over an ArrayList is its ability to insert and remove elements. Which allows you to remove or insert an element into the linked list, without having to worry about resizing, unlike an ArrayList.

Singly linked list



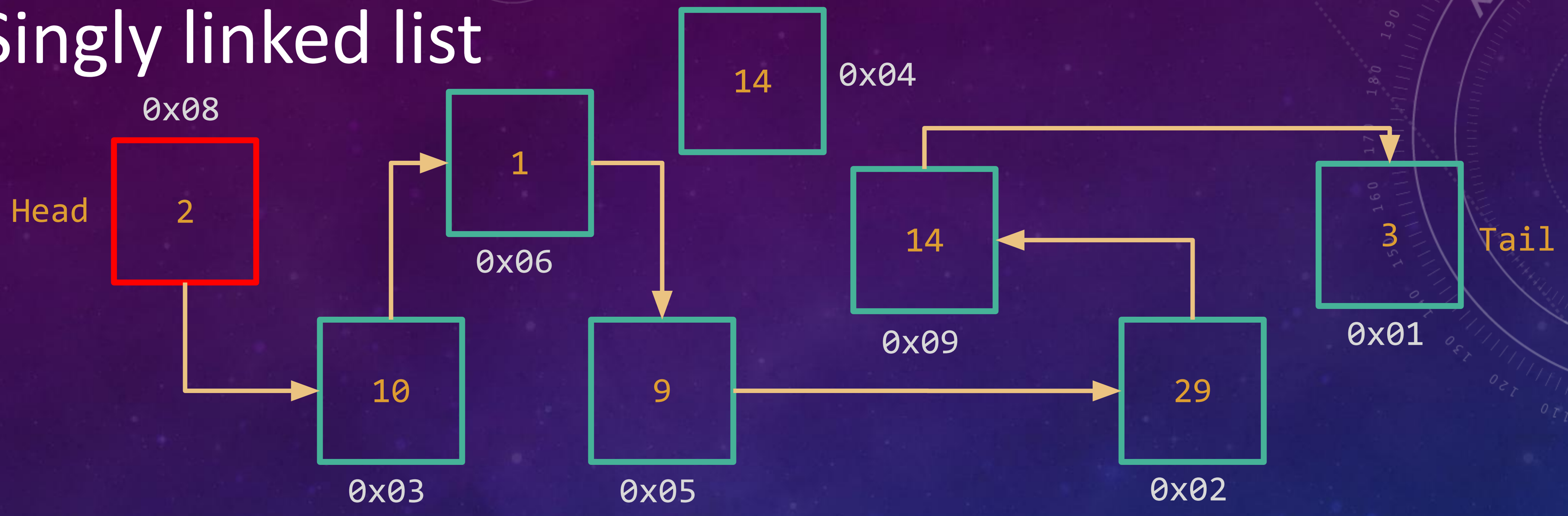
- A common reason why people would choose a linked list over an ArrayList is its ability to insert and remove elements. Which allows you to remove or insert an element into the linked list, without having to worry about resizing, unlike an ArrayList.
- Let's say, I want to insert element 14 between nodes 1 and 9.

Singly linked list



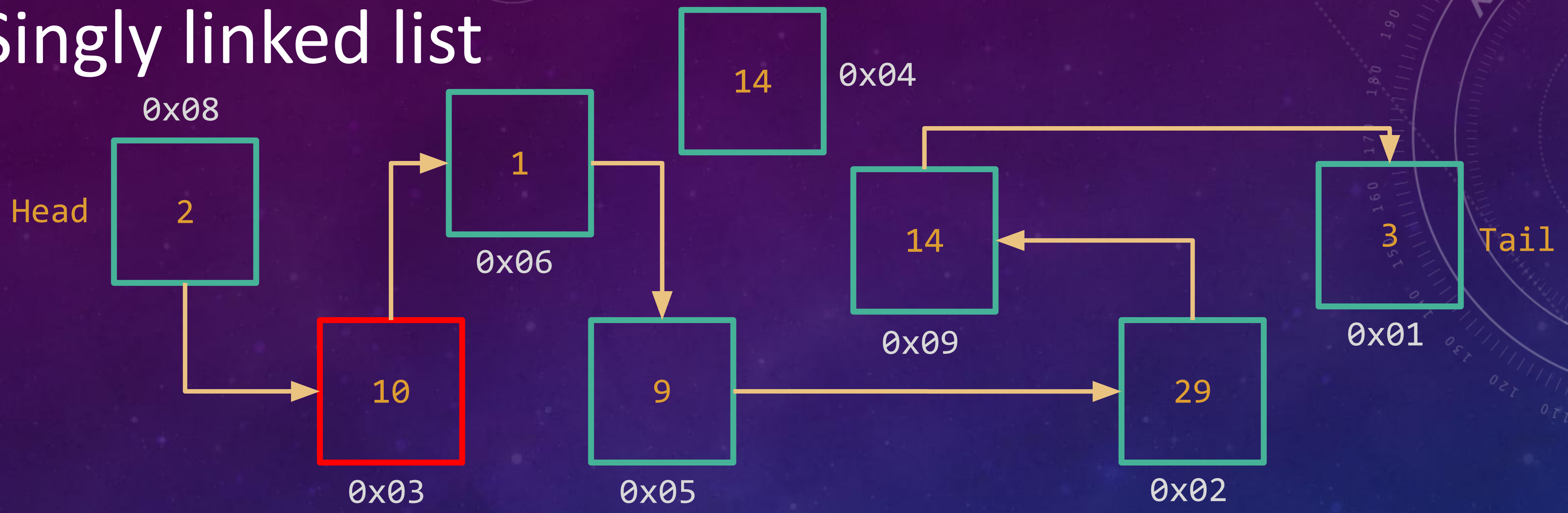
- A common reason why people would choose a linked list over an ArrayList is its ability to insert and remove elements. Which allows you to remove or insert an element into the linked list, without having to worry about resizing, unlike an ArrayList.
- Let's say, I want to insert element 14 between nodes 1 and 9. All I would need to do is iterate through the linked lists…

Singly linked list



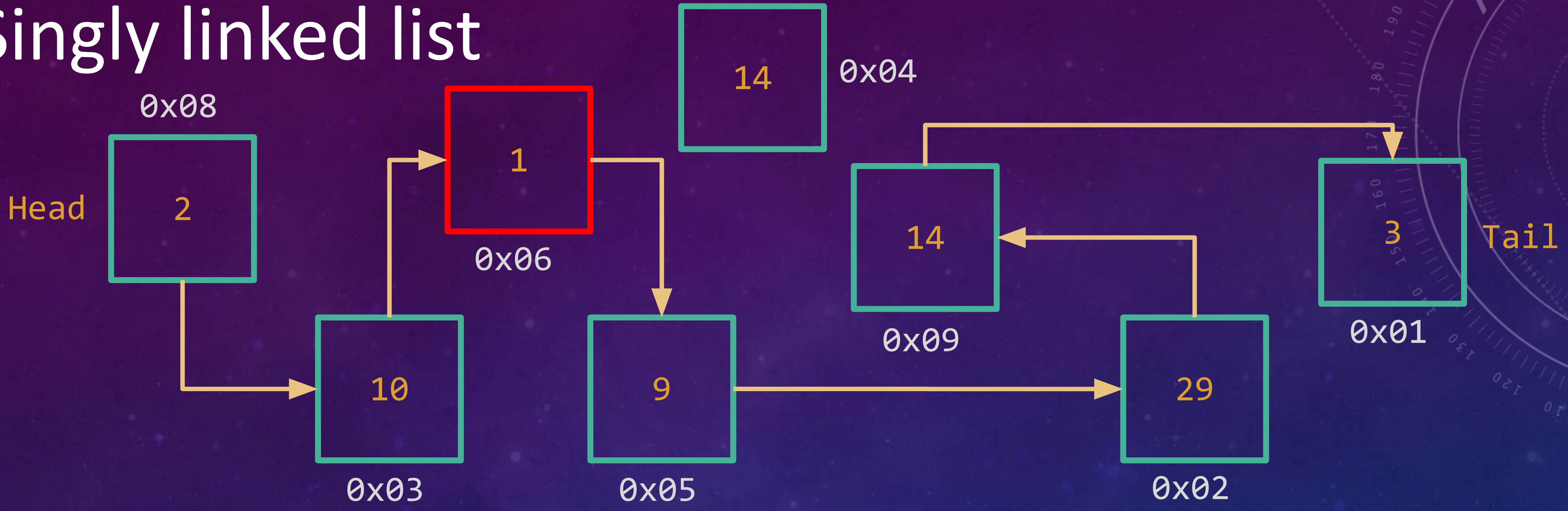
- A common reason why people would choose a linked list over an ArrayList is its ability to insert and remove elements. Which allows you to remove or insert an element into the linked list, without having to worry about resizing, unlike an ArrayList.
- Let's say, I want to insert element 14 between nodes 1 and 9. All I would need to do is iterate through the linked lists…

Singly linked list



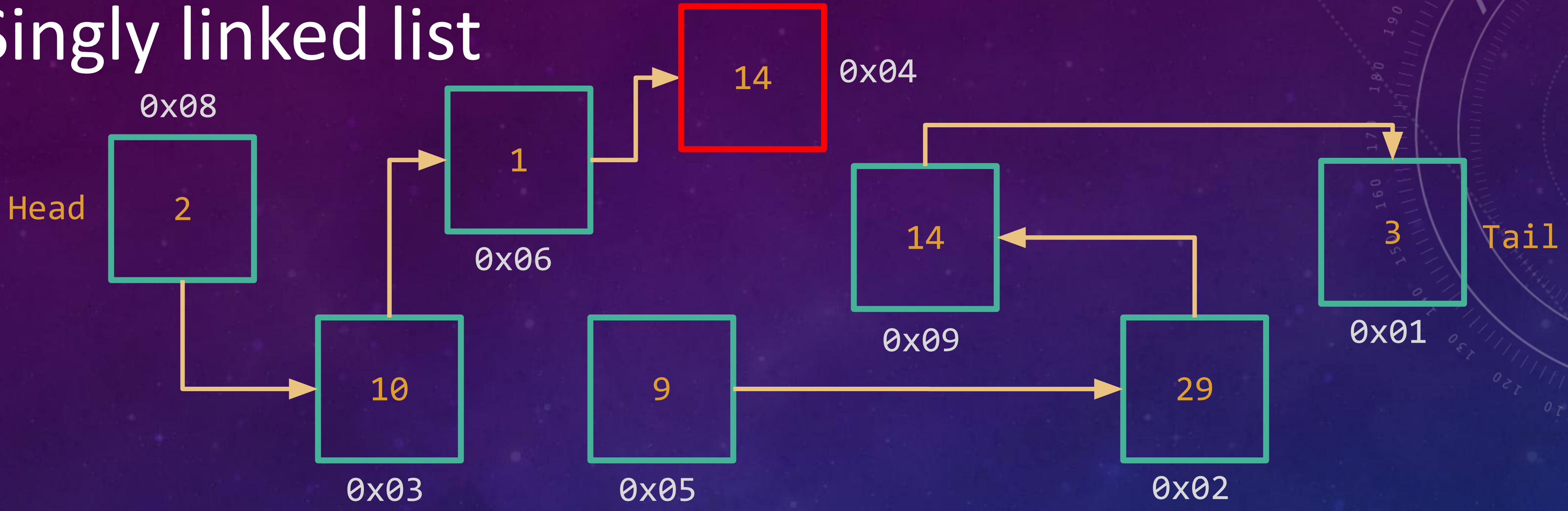
- A common reason why people would choose a linked list over an ArrayList is its ability to insert and remove elements. Which allows you to remove or insert an element into the linked list, without having to worry about resizing, unlike an ArrayList.
- Let's say, I want to insert element 14 between nodes 1 and 9. All I would need to do is iterate through the linked lists…

Singly linked list



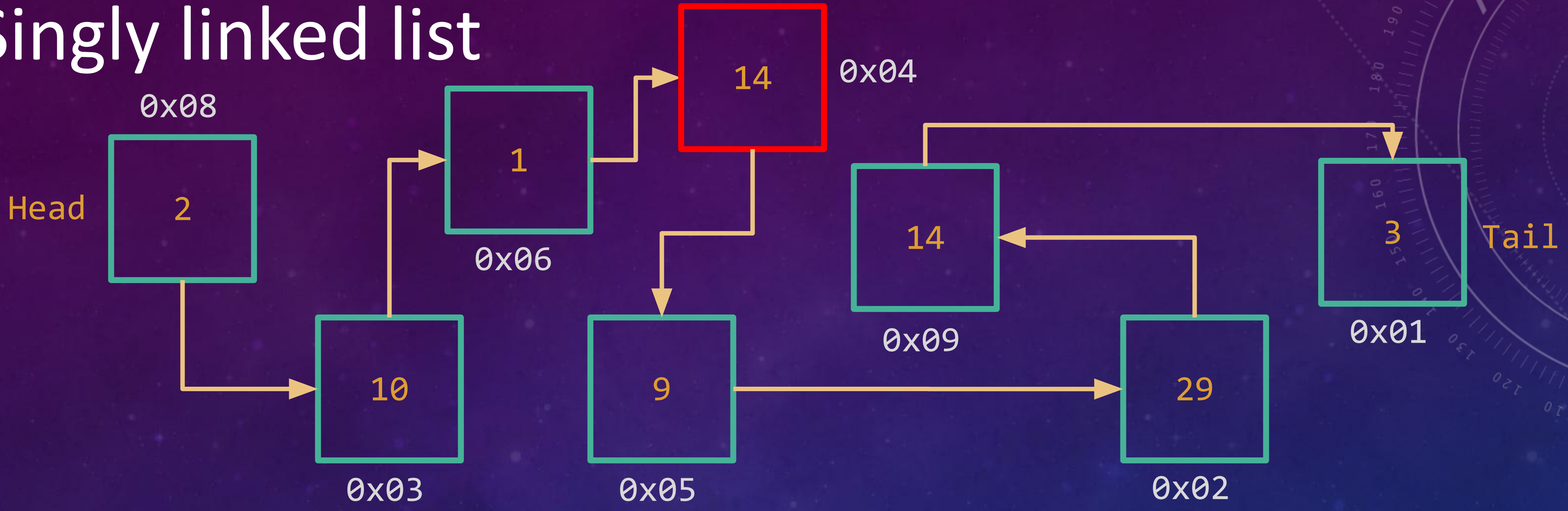
- A common reason why people would choose a linked list over an ArrayList is its ability to insert and remove elements. Which allows you to remove or insert an element into the linked list, without having to worry about resizing, unlike an ArrayList.
- Let's say, I want to insert element 14 between nodes 1 and 9. All I would need to do is iterate through the linked lists… And tell the node containing 1 to now point to the node containing 14

Singly linked list



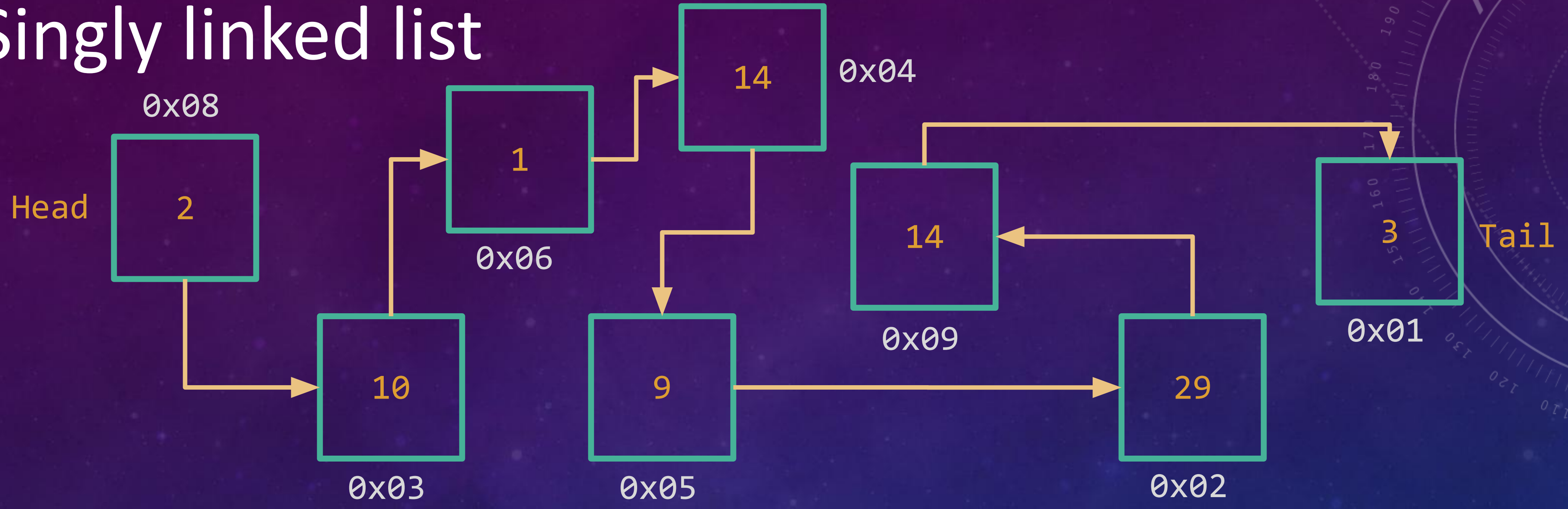
- A common reason why people would choose a linked list over an ArrayList is its ability to insert and remove elements. Which allows you to remove or insert an element into the linked list, without having to worry about resizing, unlike an ArrayList.
- Let's say, I want to insert element 14 between nodes 1 and 9. All I would need to do is iterate through the linked lists… And tell the node containing 1 to now point to the node containing 14… then tell the node containing 14 to point to the node containing 9.

Singly linked list



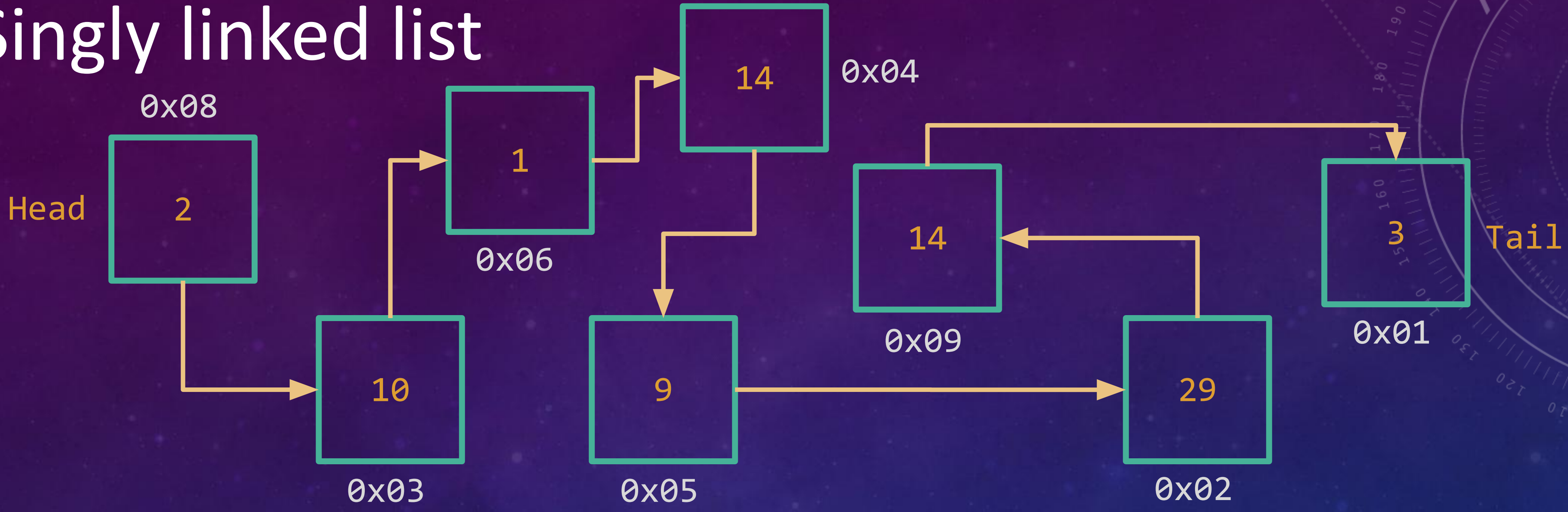
- A common reason why people would choose a linked list over an ArrayList is its ability to insert and remove elements. Which allows you to remove or insert an element into the linked list, without having to worry about resizing, unlike an ArrayList.
- Let's say, I want to insert element 14 between nodes 1 and 9. All I would need to do is iterate through the linked lists… And tell the node containing 1 to now point to the node containing 14… then tell the node containing 14 to point to the node containing 9.

Singly linked list



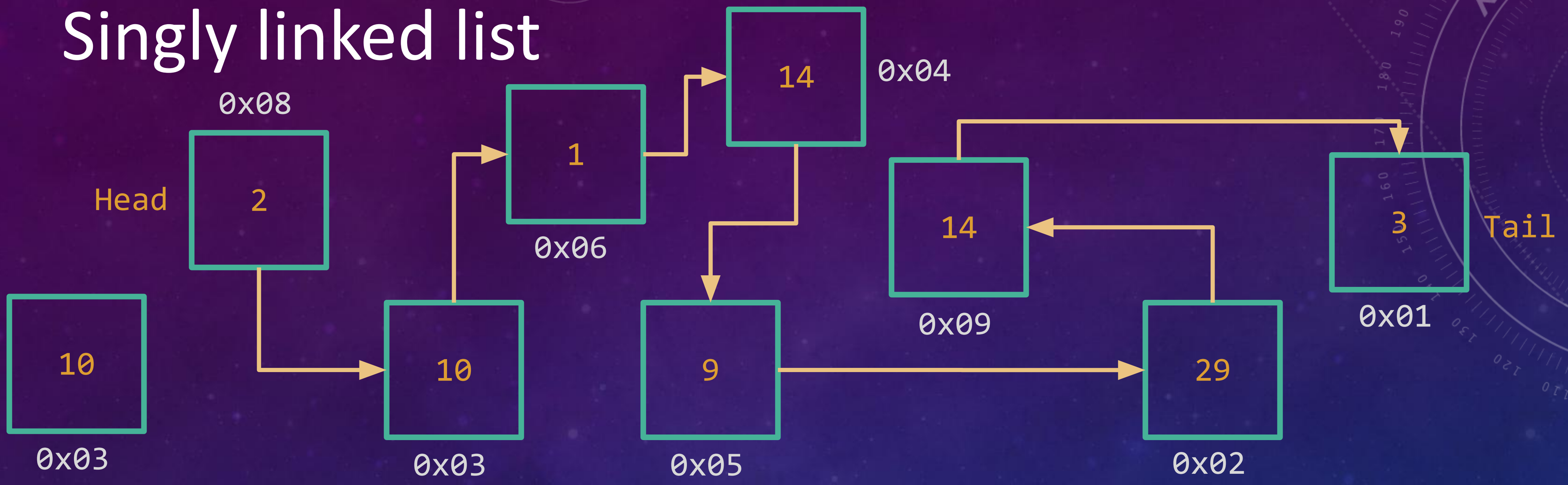
- A common reason why people would choose a linked list over an ArrayList is its ability to insert and remove elements. Which allows you to remove or insert an element into the linked list, without having to worry about resizing, unlike an ArrayList.
- Let's say, I want to insert element 14 between nodes 1 and 9. All I would need to do is iterate through the linked lists… And tell the node containing 1 to now point to the node containing 14… then tell the node containing 14 to point to the node containing 9.
- No need to resize a linked list!

Singly linked list



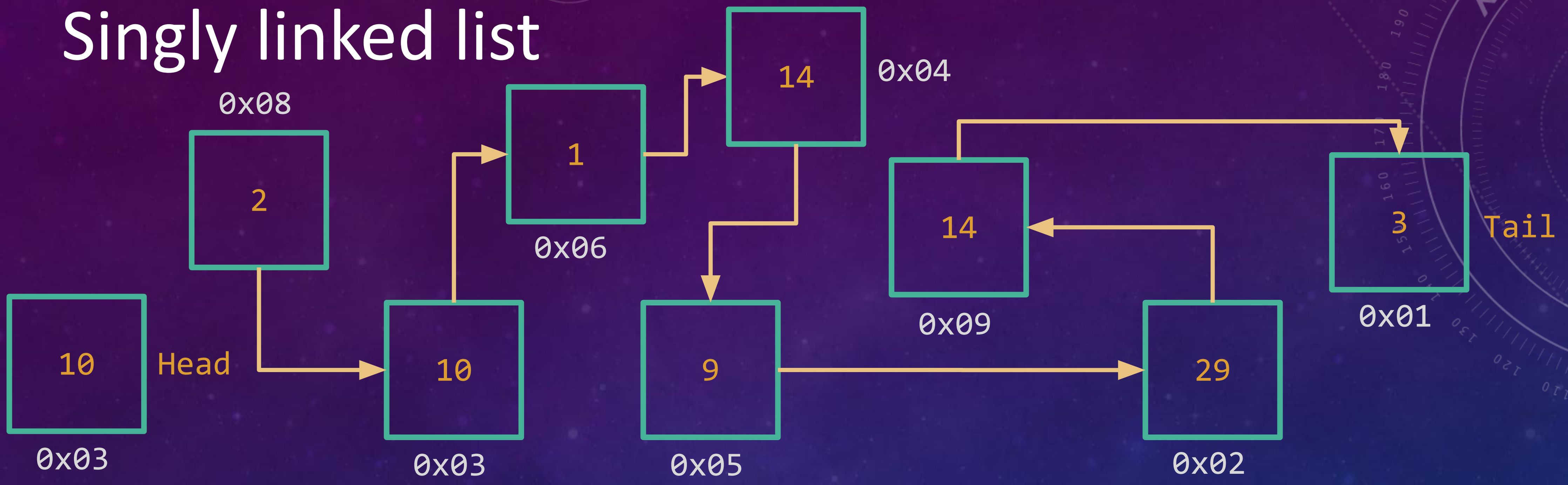
- Similarly, to push an element to the front of a linked list. All that is required is a couple of simple steps…

Singly linked list



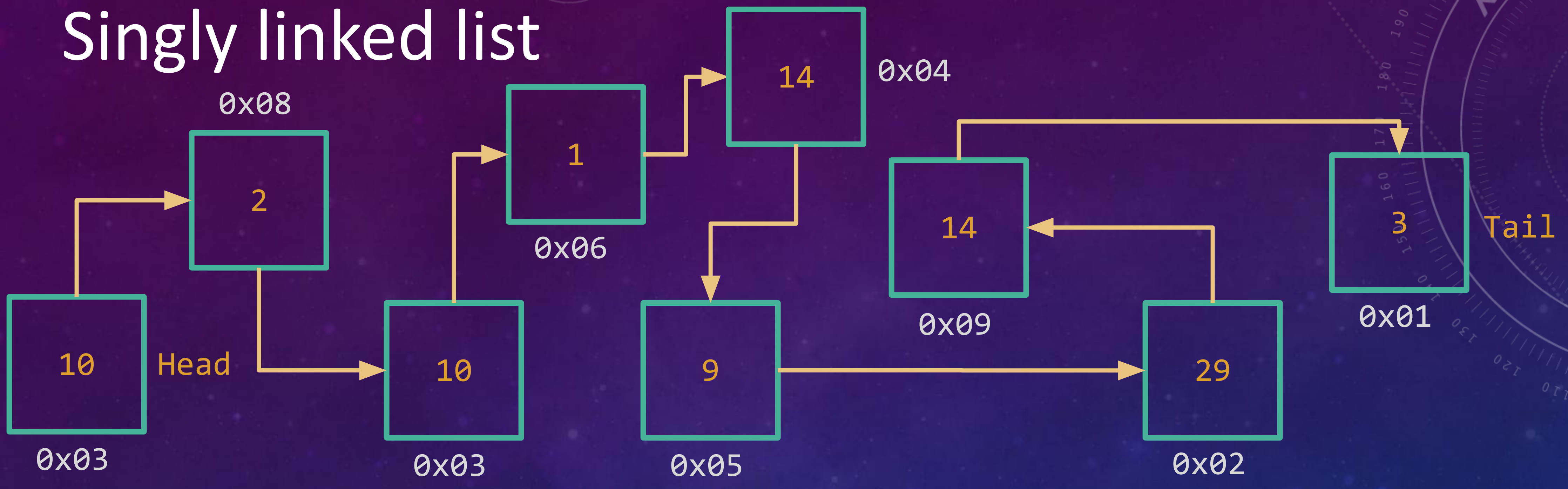
- Similarly, to push an element to the front of a linked list. All that is required is a couple of simple steps… define the new node you want to be at the head…

Singly linked list



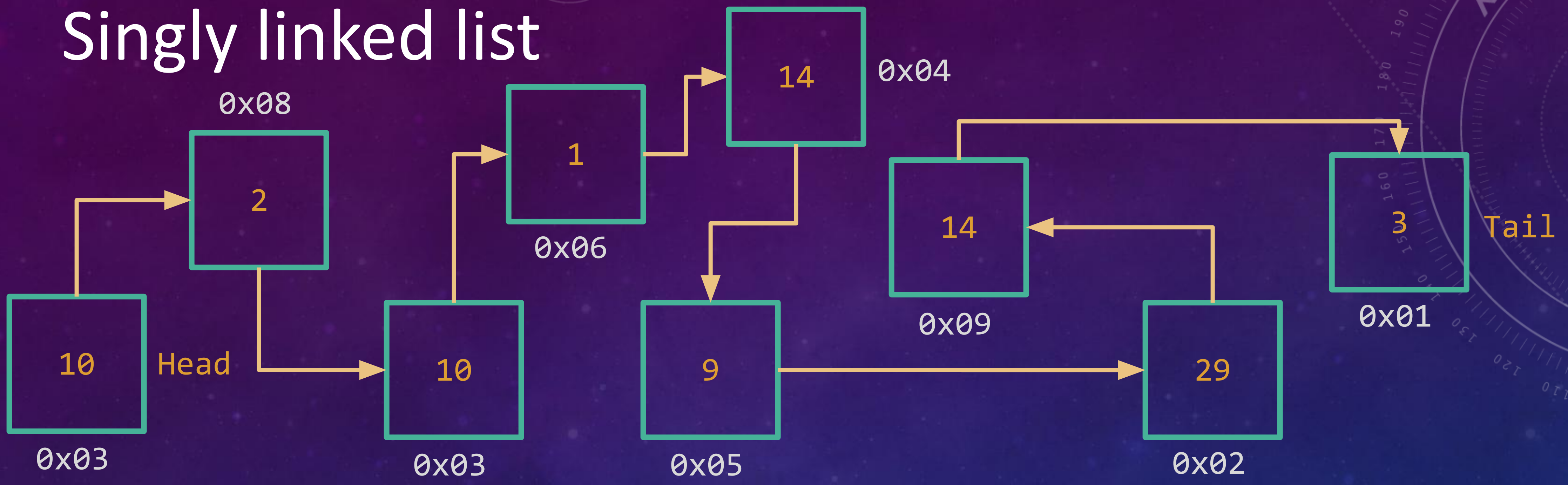
- Similarly, to push an element to the front of a linked list. All that is required is a couple of simple steps… define the new node you want to be at the head… assign that new node as the linked lists new head…

Singly linked list



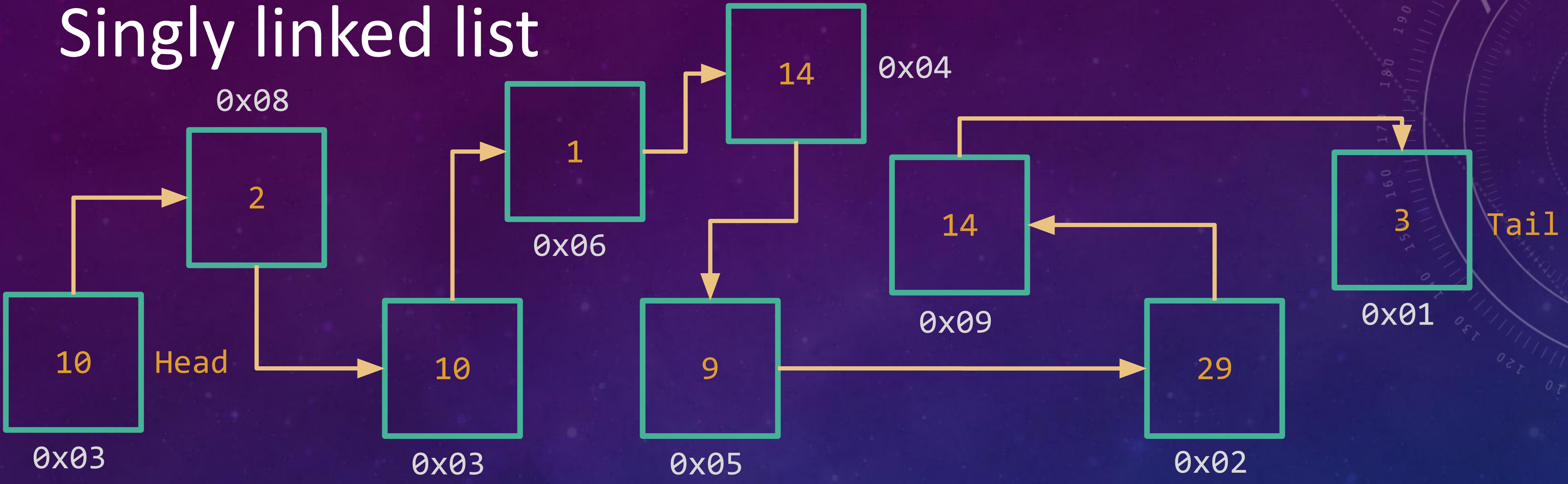
- Similarly, to push an element to the front of a linked list. All that is required is a couple of simple steps… define the new node you want to be at the head… assign that new node as the linked lists new head… and point the new head at the previous head. It’s that simple

Singly linked list



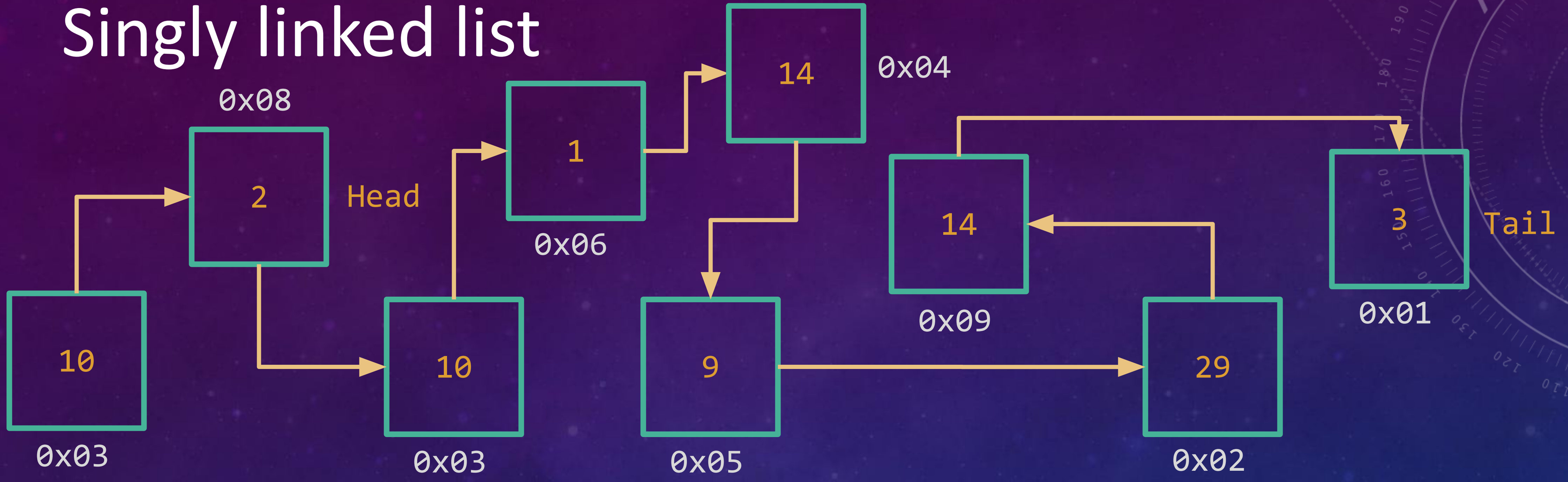
- Similarly, to push an element to the front of a linked list. All that is required is a couple of simple steps… define the new node you want to be at the head… assign that new node as the linked lists new head… and point the new head at the previous head. It's that simple
- There are always many different types of linked lists.

Singly linked list



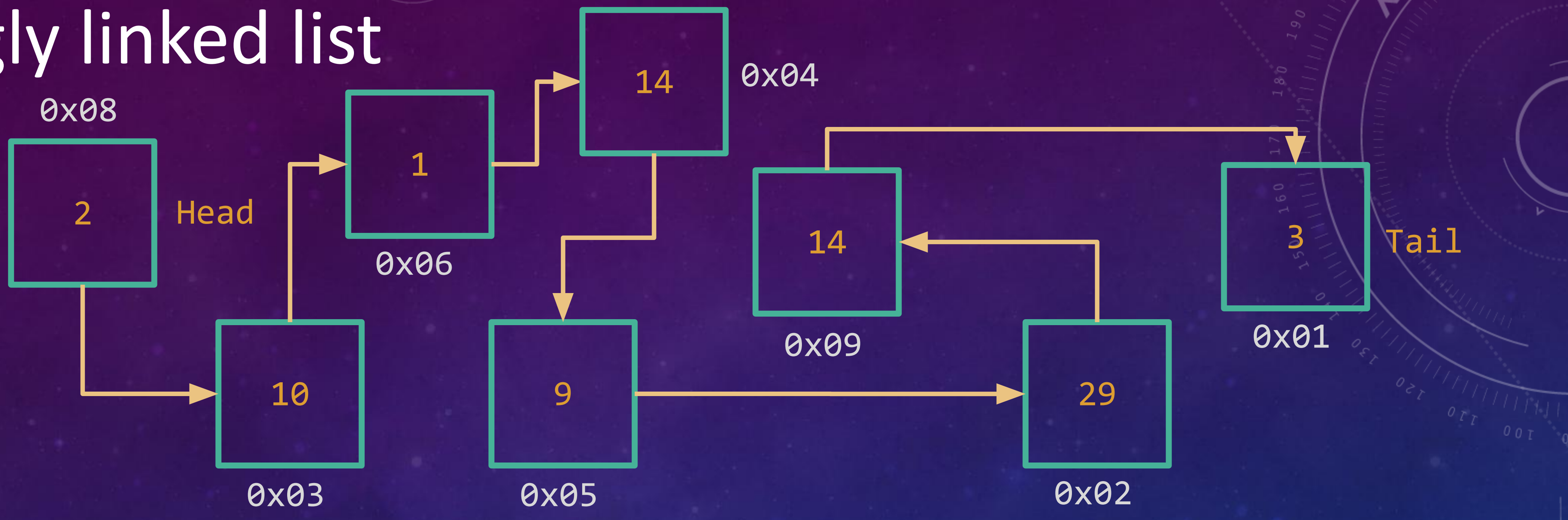
- Similarly, to push an element to the front of a linked list. All that is required is a couple of simple steps… define the new node you want to be at the head… assign that new node as the linked lists new head… and point the new head at the previous head. It’s that simple
- There are always many different types of linked lists.
- Whilst to pop an element from the front, we would need to…

Singly linked list



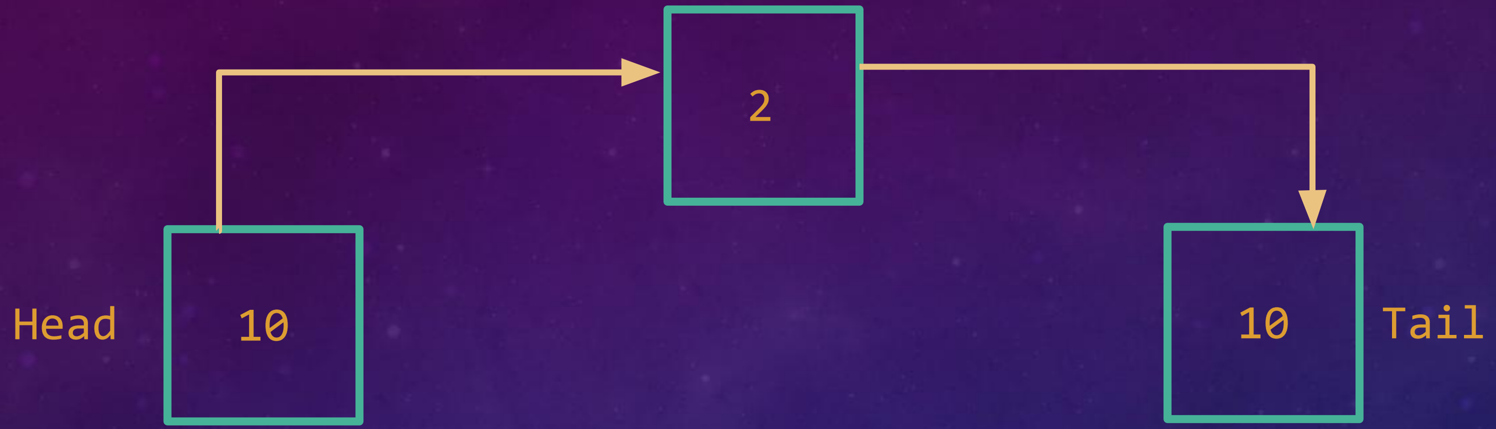
- Similarly, to push an element to the front of a linked list. All that is required is a couple of simple steps… define the new node you want to be at the head… assign that new node as the linked lists new head… and point the new head at the previous head. It's that simple
- There are always many different types of linked lists.
- Whilst to pop an element from the front, we would need to… assign the node the current head is pointing to as the new head, and…

Singly linked list



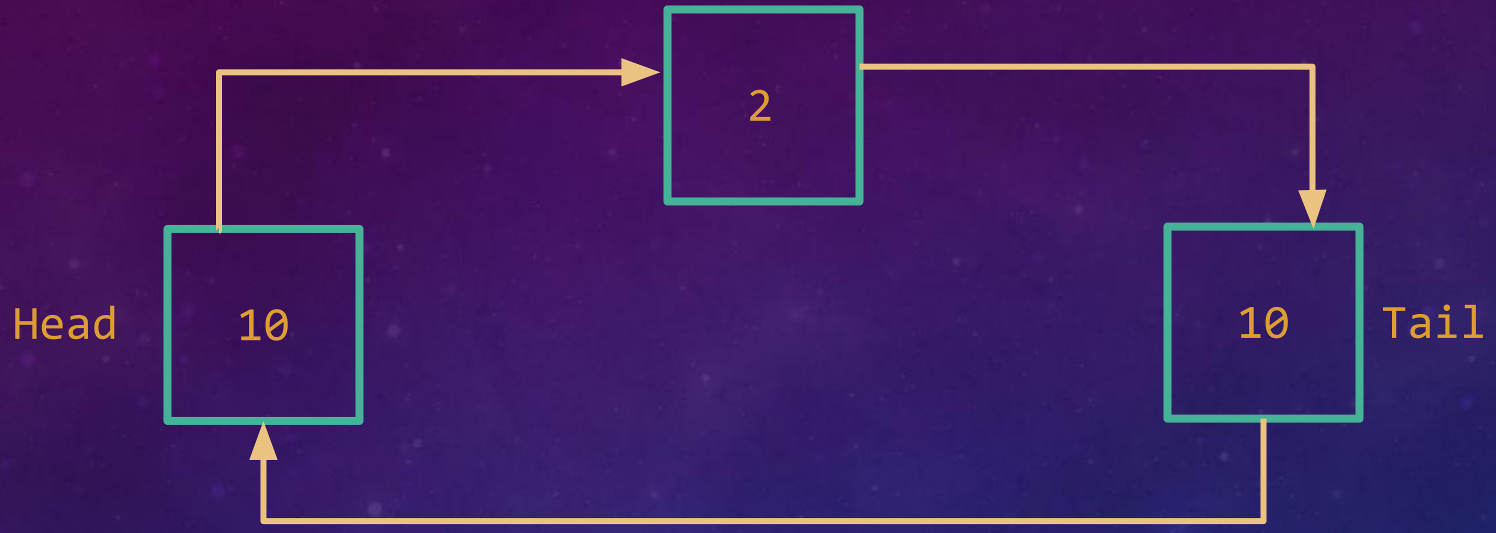
- Similarly, to push an element to the front of a linked list. All that is required is a couple of simple steps… define the new node you want to be at the head… assign that new node as the linked lists new head… and point the new head at the previous head. It's that simple
- Whilst to pop an element from the front, we would need to… assign the node the current head is pointing to as the new head, and… delete the node that was previously the head of the linked lists

Singly linked list



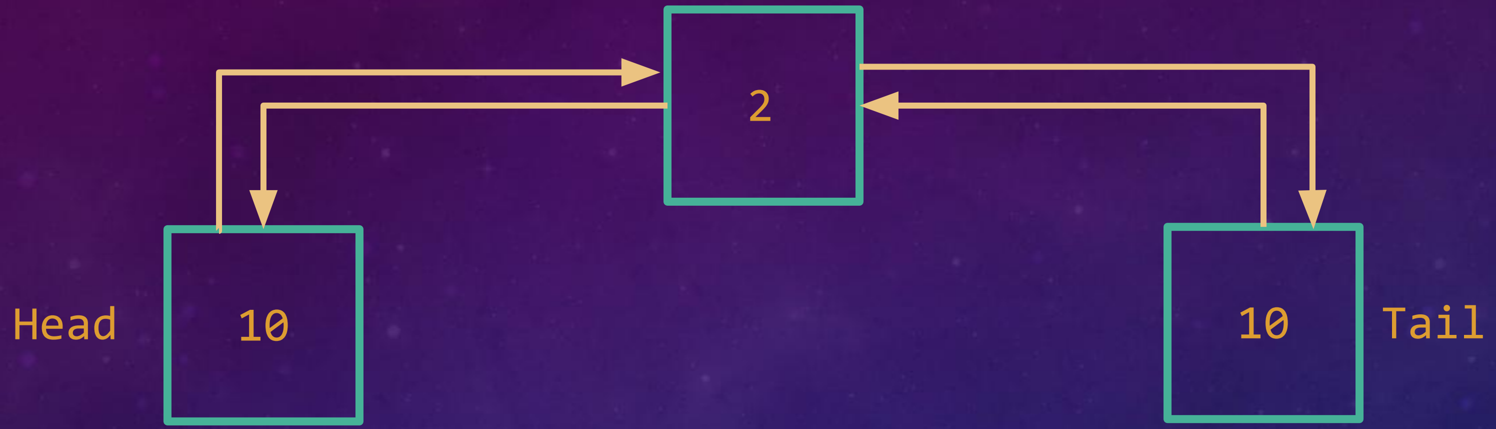
- Similarly, to push an element to the front of a linked list. All that is required is a couple of simple steps… define the new node you want to be at the head… assign that new node as the linked lists new head… and point the new head at the previous head. It's that simple
- There are also many different types of linked lists. Currently, we've only been looking at singly linked lists where the links are only in one direction, but there are also…

Circular singly linked list



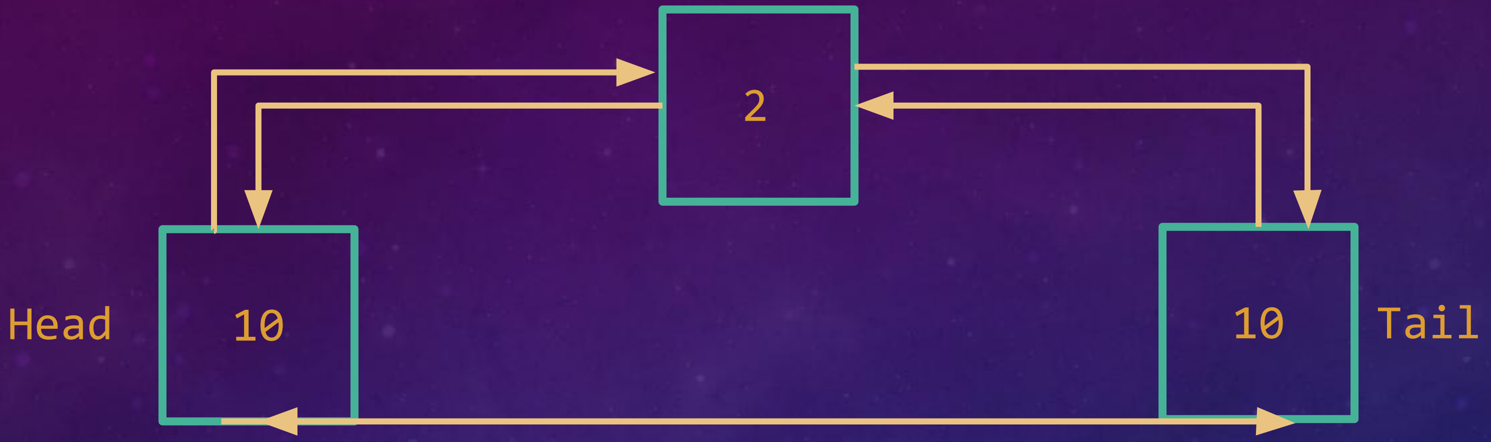
- Similarly, to push an element to the front of a linked list. All that is required is a couple of simple steps… define the new node you want to be at the head… assign that new node as the linked lists new head… and point the new head at the previous head. It's that simple
- There are also many different types of linked lists. Currently, we've only been looking at singly linked lists where the links are only in one direction, but there are also… circular singly linked lists, where the tail connects back to the head…

Doubly linked list



- Similarly, to push an element to the front of a linked list. All that is required is a couple of simple steps… define the new node you want to be at the head… assign that new node as the linked lists new head… and point the new head at the previous head. It's that simple
- There are also many different types of linked lists. Currently, we've only been looking at singly linked lists where the links are only in one direction, but there are also… circular singly linked lists, where the tail connects back to the head… doubly linked lists where there are links in both directions…

Circular doubly linked list



- Similarly, to push an element to the front of a linked list. All that is required is a couple of simple steps… define the new node you want to be at the head… assign that new node as the linked lists new head… and point the new head at the previous head. It's that simple
- There are also many different types of linked lists. Currently, we've only been looking at singly linked lists where the links are only in one direction, but there are also… circular singly linked lists, where the tail connects back to the head… doubly linked lists where there are links in both directions… and finally circular doubly linked lists, which have links on both directions including links to the tail and head

std::forward_list

```
std::forward_list<type> myList{};
```

- Here is how to declare a `forward_list` in your C++ code, don’t forget to `#include <forward_list>`
- You won’t be using `forward_list` a lot in this course, and in the problems you will be solving. But it will give you a good understanding of the data structure as a whole with the main operations it will contain.

std::forward_list

Main operations for `std::forward_list`

```
std::forward_list<int> myList{};
```

- `myList.front()`: accesses element at the front of the list
  - `myList.front() // returns 4`
- `myList.push_front(value)`: pushes `value` to the front of the list
  - `myList.push_front(13)`
  - `myList.push_front(9)`
  - `myList.push_front(10)`
  - `myList.push_front(3): // linked list now contains {3, 10, 9, 13}`
- `myList.pop_front()`: removes the element at the front of the list
  - `myList.pop_front(): // linked list now contains {10, 9, 13}`
- `myList.insert_after()`: inserts an element at a certain position (Check CppReference for more detail)
- `myList.erase_after()`: removes an element at a certain position (Check CppReference for more detail)
- `myList.size()`: returns the size of the list
  - `myList.size() // returns 3 with {10, 9, 13}`
- `myList.empty()`: empties the list
  - `myList.empty() // list now contains {}`

Give “Singly linked list” a go to try and implement one yourself!!!

Templating MyVector

- Templating MyVector is a fairly straightforward exercise. So I put more effort into linked lists this tutorial, since your first assignment will involve them.
- But essentially all that is required is you alter the code we worked on last week for our toy vector, and change `int`’s that are related to the data type stored in MyVector to `T`

Templating MyVector

- Templating MyVector is a fairly straightforward exercise. So I put more effort into linked lists this tutorial, since your first assignment will involve them.
- But essentially all that is required is you alter the code we worked on last week for our toy vector, and change `int`'s that are related to the data type stored in MyVector to `T`
- For example, this constructor from last week, will turn into…

```
MyVector::MyVector(std::initializer_list<int> vals) {
   size_ = capacity_ = vals.size();
   arrayPointer_ = new int[size_];
   int i = 0;
   for(int x : vals) {
    arrayPointer[i++] = x;
   }
}
```

Templating MyVector

- Templating MyVector is a fairly straightforward exercise. So I put more effort into linked lists this tutorial, since your first assignment will involve them.
- But essentially all that is required is you alter the code we worked on last week for our toy vector, and change `int`'s that are related to the data type stored in MyVector to `T`
- For example, this constructor from last week, will turn into… this

```
template <typename T>
MyVector::MyVector(std::initializer_list<T> vals) {
    size_ = capacity_ = vals.size();
    arrayPointer_ = new T[size_];
    int i = 0;
    for(T x : vals) {
     arrayPointer[i++] = x;
    }
}
```

Access to google drive

- I will upload slides to the Google Drive after every class
- https://drive.google.com/drive/folders/1H5psebndM_YVyoJE-BJ_ODNJOfgq9-ul

Contact: Thomas.golding@uts.edu.au