Dynamic Memory Management II

COSC1076
Semester 1 2019
Week 08

RMIT UNIVERSITY

Admin

- Assignment 2 Preparation
  - Register your group ***this week***
  - Group size 4
  - Groups formed in Lab classes
    - There will be updates in labs
- Week 8 Online Quiz
  - More info at end of lecture
- Friday
  - Public Holiday

Review: Linked Lists

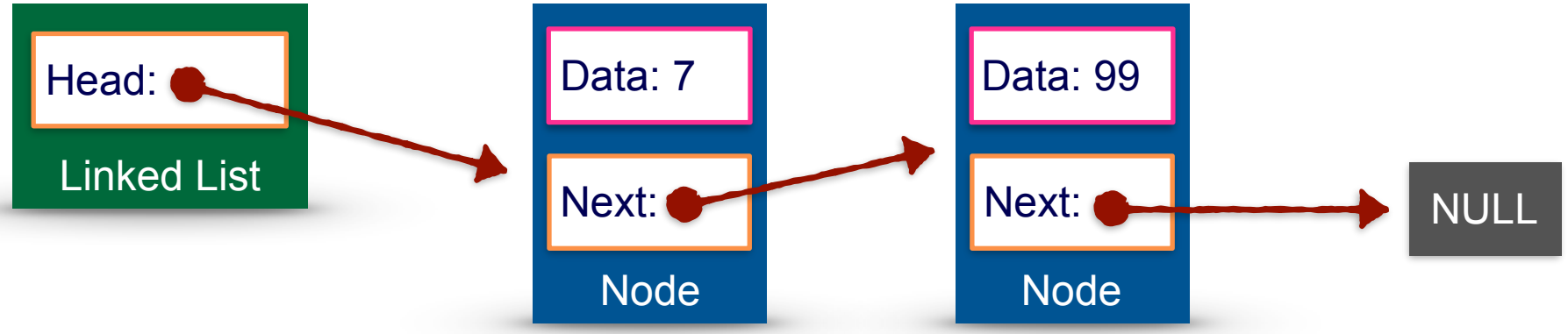
RMIT UNIVERSITY

Abstract Data Types (ADTs)

- An ***Abstract Data Type (ADT)*** consists of:
  1. An interface
     - A set of operations that can be performed.
     - Usually contained in a header file
  2. The allowable behaviours
     - The way we expect instances of the ADT to respond to operations.
- The implementation of an ADT consists of:
  1. An internal representation
     - Data stored inside the object's instance variables/members.
  2. A set of methods implementing the interface.
     - Usually contained in a code file
  3. A set of representation invariants, true initially and preserved by all methods

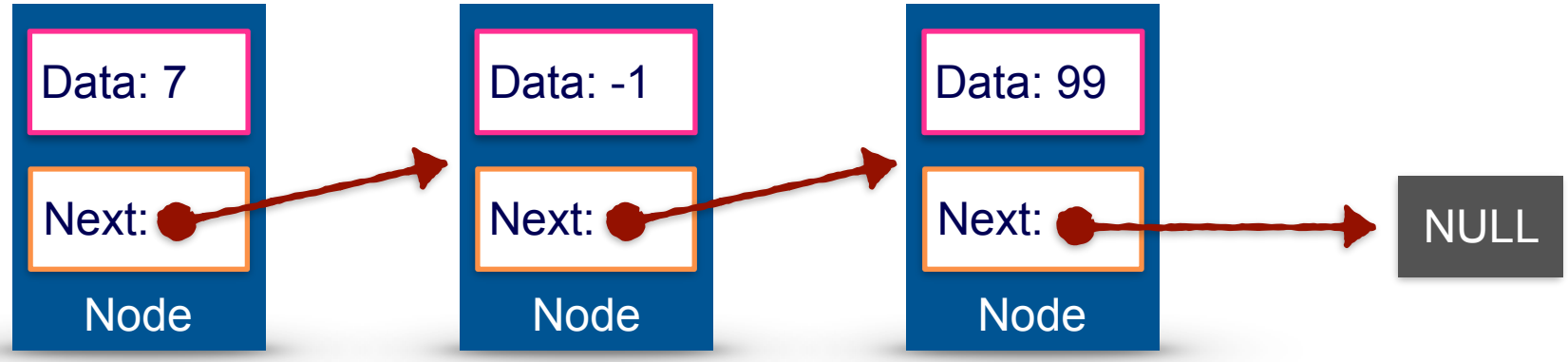
Linked List ADT

- To use the list of nodes, a ***Linked List ADT*** is created which:
  - Internally contains a pointer to the ***head*** (first node) of the list
  - Defines methods to interact with the linked list
    - These methods are similar to those used on the ParticleList!



Linked List

- A ***Linked List*** is a data types that:
  - Stores elements of the same type
  - Stores the elements as a sequential “chain”
  - The last element of the chain is “null"



Linked Lists vs Arrays/Vectors

	Array/Vector	Linked List
Element Types	All elements are same type	All elements are same type
Resizeable	Yes Requires Copy	Yes
Storage Layout	Contiguous	Non-Contiguous
Cell lookup	Constant Time Random Access	Linear Time Random Access

Linked Lists vs Arrays/Vectors

	Array/Vector	Linked List
Insert at Front	Linear Time Requires Copy	Constant Time
Insert at Back	Constant Time	Linear Time (if no back ptr) Constant Time (back ptr)
Delete at Front	Linear Time Requires Copy	Constant Time
Delete at Back	Constant Time	Linear Time (if no back ptr) Constant Time (back ptr)

Linked Lists Continued

RMIT UNIVERSITY

More operations

- Today we will look at:
  - delete at index
  - Double Ended Linked Lists
- Remember, before starting on any method, draw the picture first!

Double Ended Linked Lists

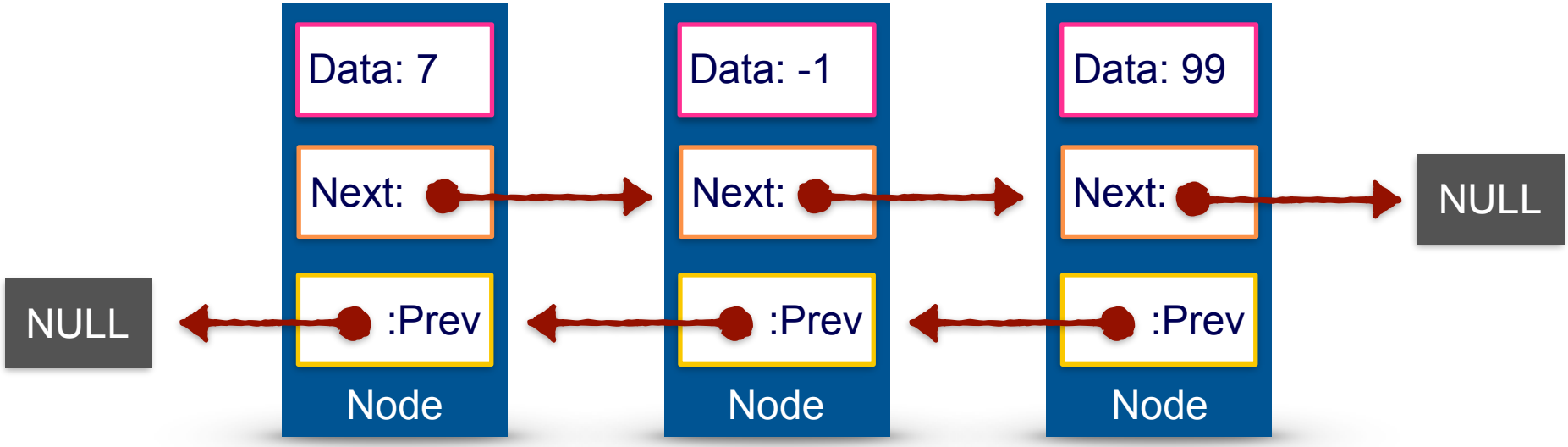
RMIT UNIVERSITY

More operations

- Today we will look at:
  - delete at index
  - Double Ended Linked Lists
- Remember, before starting on any method, draw the picture first!

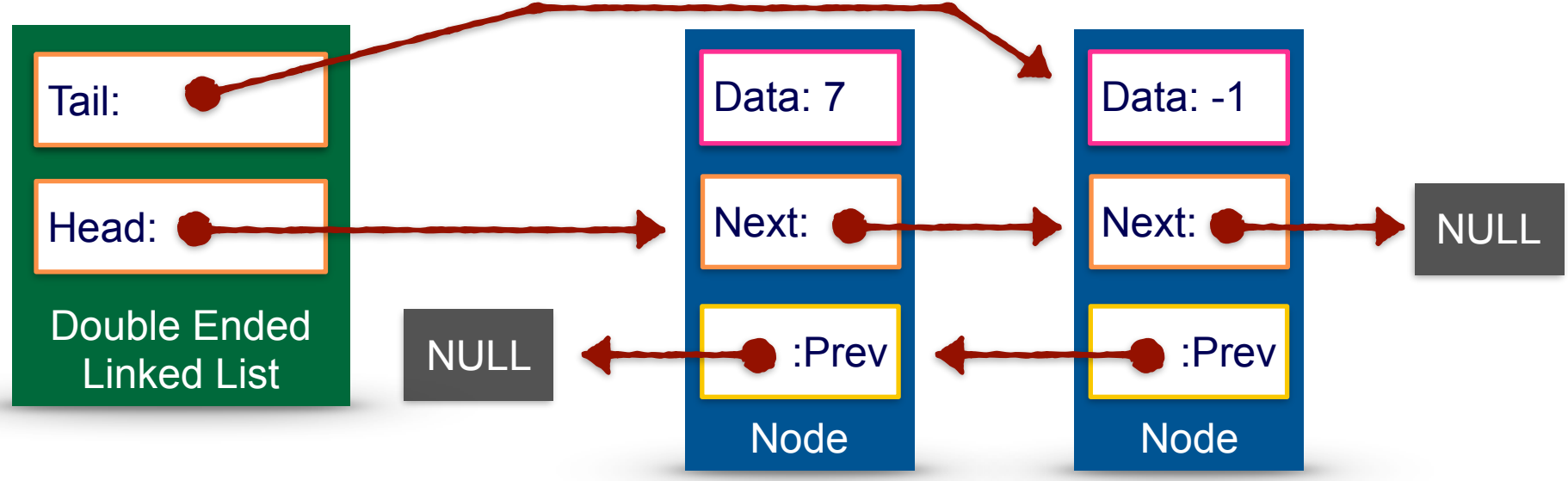
Linked List

- A ***Double Ended Linked List*** is a data types that:
  - Is a linked list
  - Has pointers for the chain of nodes in both directions



Linked List

- The ***Double Ended Linked List ADT***:
  - Has a pointer to the ***head*** and ***tail*** of the linked list
  - Defines methods to interact with the linked list



Linked Lists vs Arrays/Vectors

	Linked List	Double Ended Linked List
Element Types	All elements are same type	All elements are same type
Resizeable	Yes	Yes
Storage Layout	Non-Contiguous	Non-Contiguous
Cell lookup	Linear Time Random Access	Linear Time Random Access

Linked Lists vs Arrays/Vectors

	Linked List	Double Ended Linked List
Insert at Front	Constant Time	Constant Time
Insert at Back	Linear Time	Constant Time
Delete at Front	Constant Time	Constant Time
Delete at Back	Linear Time	Constant Time

Scope

RMIT UNIVERSITY

Scope

- ***Scope*** describes where a name/label/entity exists and is usable.
- Names include:
  - Variables
  - Constants and `#define`'s
  - Functions
  - Classes
  - Class Methods & Fields
    - Scope also includes the visibility of the class methods and fields
- Scoping rules are programming language dependent
  - While many of the following notes apply across languages there are important differences, especially with languages such as python!

Variable Scope

- Variables are typically scoped by:
  - Code block
    - If/elseif
    - While/for
    - Manual blocks
  - Functions/methods
  - Global scope
- Local Variable scope is controlled by the program call stack
  - A local variable is mapped to the block in which a local variable is created
    - This happens when the variable is pushed onto the stack
  - When the block is exited, the corresponding variables go out of scope
    - The variables are popped off the stack

Function Scope

- Function are typically scoped by:
  - Namespace
  - C++ Source File & Header includes
- Function Parameters
  - Parameter Scope is from the beginning of the function to immediately after the function return point
  - Parameters may be shadowed by local function variables

Class Scope

- Function are typically scoped by:
  - Namespace
  - C++ Source File & Header includes
- Classes define three scope visibilities
  - Public - May be accessed from any entity with a reference to the class
  - Protected - May only be accessed from the class, or any sub-class
  - Private - May only be accessed from the class itself, not any sub-class
- By default, class scope is private

Namespace

- A Namespace defines a scope in to encapsulate
  - Functions
  - Classes
  - Namespace “Global” variables (outside of another scope)
  - Constants
- Classes also define a namespace, of the name as the class

Pre-processor statements

- Pre-processor statements, such as `#define`, are handled *before* outside the syntax-tree is parsed
  - Any `#define` value is processed before namespaces or scope apply
  - Thus, `#define` statements are “outside” of any scope

Shadowing

- ***Shadowing*** is where labels (such as variables, functions, etc.) in an inner-scope ***hide*** an entity with the identical label in an outer-scope
  - For example:
- In general,
  - The entity from the inner-most scope is referenced by the label
  - It is possible to reference entities from a outer-scope, generally through labels that the explicitly call that scope
    - Most typically through namespace and class names

Self Managed Objects

RMIT UNIVERSITY

Review: Object Ownership

- ***Ownership*** is the concept of who (which module/object/code) has responsibility for managing the life-cycle of memory allocated on the heap
  - The owner manages how an object is accessed and modified
  - Importantly, the owner must “clean-up” memory
- Ownership over an object may be transferred

Self-Managed Objects

- A ***self-managed object*** is responsible for managing it's own lifecycle.
- A self-managed objects decides:
  - How other objects access, read and modify its contents
  - When to delete itself
- Generally, an self-managed object tracks all references to itself
  - When there are no more references to the self-managed object, then the object cleans-up its own memory
- In C++, self-managed objects are implemented by so-called ***smart pointers***
  - Also known as "auto" pointers

Smart Pointer

- A ***smart pointer*** is a class that wraps a pointer to an object that has been allocated on the heap
  - The smart pointer controls all access to the object
  - The smart pointer internally tracks references to itself
  - The smart pointer may delete the object if:
    - There are no more references to the smart pointer
    - The smart pointer goes entirely out of scope
    - The smart pointer is told to manage a different object
- C++ Provides two forms of smart pointers
  - unique_ptr
  - shared_ptr
  - weak_ptr (not covered in this course)

Unique Pointer

- A ***unique pointer*** is a smart pointer where:
  - There may be only ONE unique pointer to the same object
  - The unique pointer cannot be copied
    - It can be “moved” via C++ Move Semantics (see later this lecture)
    - A unique pointer can be returned from a function/method
  - The object is deleted when:
    - The unique pointer goes out of scope
    - Another object is given to the unique pointer
- A unique pointer is a generic object, and is given a type
- A unique pointer is created using the special `make_unique` function

Shared Pointer

- A ***shared pointer*** is a smart pointer where:
  - Multiple shared pointers may refer to the same object
  - The shared pointer can be copied
  - The object is deleted when:
    - There are no shared pointers with a reference to the object
- A shared pointer is a generic object, and is given a type
- A unique pointer is created using the special `make_shared` function

Using Unique & Shared Pointers

- In the <memory> header file
- Must be created with special function, not a new statement
- Syntactically, they work just like “raw” pointers
  - Dereference with “*” or “->” syntax
- Work with dynamically allocated arrays as well
  - Arrays values are accessed with “[]” syntax
  - Calls correct version of delete to clean-up the array memory

Why not just use Smart Pointers?

- Actually, the C++ language specification recommends the user of smart pointers over manually managing “raw” pointers
  - The C++ implementation of smart pointers is very lightweight and efficient
  - Smart pointers resolve memory management issues with exception handling
- However, smart pointers have limitations:
  - Shared pointers are have issues with circular references, resulting in a memory leak
    - This can be programatically “resolved” through a weak pointers, which you can look up for your own exercise
  - Not every programming language supports smart pointers
    - Thus, in this course, understanding “raw” pointers is an important skill

“Randomness” in Programs

RMIT UNIVERSITY

What is Random?

What is Random?

- So called ***"true" randomness*** is where a sequence lacks any:
  - Identifiable Pattern
  - Predictability
  - Ordering
- Human's are very poor at identifying randomness, typically because of the gambler's fallacy.
- As every algorithm (and Turing complete machine) is a sequence of instructions, it is impossible to generate a truely random sequence

Pseudo-Randomness

- Computers can generate a ***pseudo-random number sequence***.
  - To an outside observer the sequence displays ***statistical randomness***
    - Where any number is the sequence cannot be predicted
- However, under-the-hood, each number is the sequence is algorithmically produced using the preceding number(s)
  - The sequence is initialised by a seed
  - Only if the seed is known is it possible to predict the sequence
  - Hence the name, pseudo-random

C++ Pseudo-random Number Generation

- C++ has very mathematically complex ways to generate random numbers.
  - In this course, we are just going to use a very simple paradigm
  - We will generate random integers between a minimum and maximum
- This requires two entities
  - ***Engine*** - the algorithm that generates random numbers,
    - Typically a uniform distribution of integers
  - ***Distribution*** - a mathematical formula that transforms the output of the engine into another sequence of value with varying properties
- Both come from the `random` header file

C++ Pseudo-random Number Generation

```
int min = 1;
int max = 10;
int seed = 100;

std::default_random_engine engine(seed);
std::uniform_int_distribution<int> uniform_dist(min, max);

int value = -1;
for (int i = 0; i != 10; ++i) {
   value = uniform_dist(engine);
   cout << "Randomly-chosen value: " << value << endl;
}
```

C++ Pseudo-random Number Generation

- If the engine is given the same seed, it will produce the same random sequence
- To get a “random” seed (typically taken from the computer internal clock) a different engine can be used:
  - `std::random_device engine`

Week 8 Online Quiz

RMIT UNIVERSITY

Assignment 2

RMIT UNIVERSITY

RMIT
UNIVERSITY