Data Structures & Recursion

COSC1076
Semester 1 2019
Week 08

Admin

- Assignment 2
  - You should be well on your way having started the assignment
  - Remember, there are plenty of people in your group
  - In general, the answer to many questions is:
    - See what the spec already says, OR
    - Make your own choices - that is the “expectation”
- Assignment 1 Marking
  - Underway but not yet completed

Review: Linked Lists

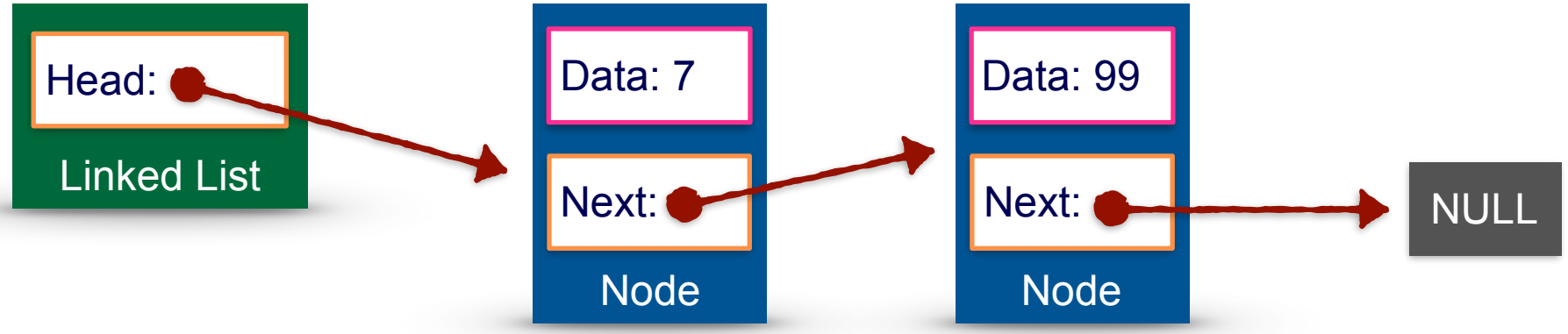
RMIT UNIVERSITY

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

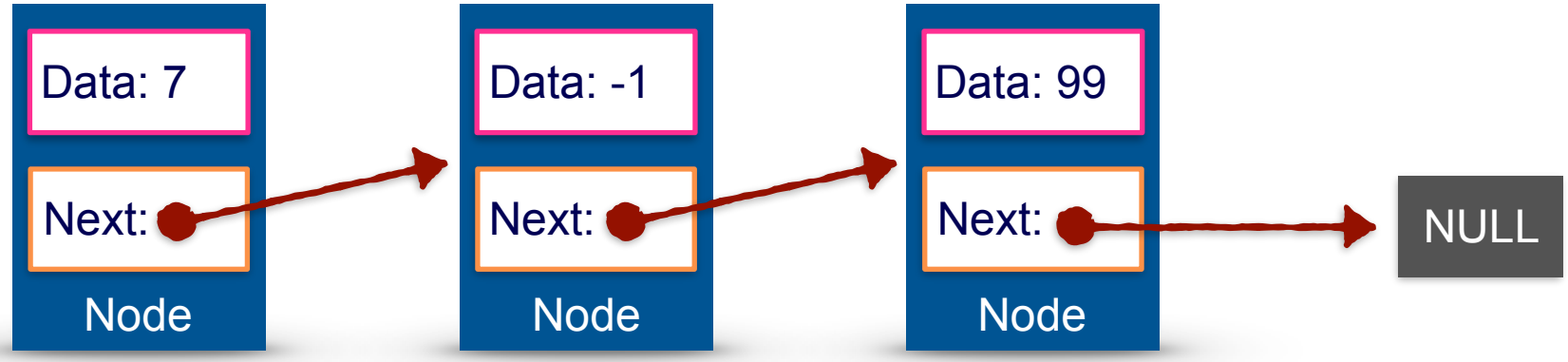
Linked List ADT

- 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

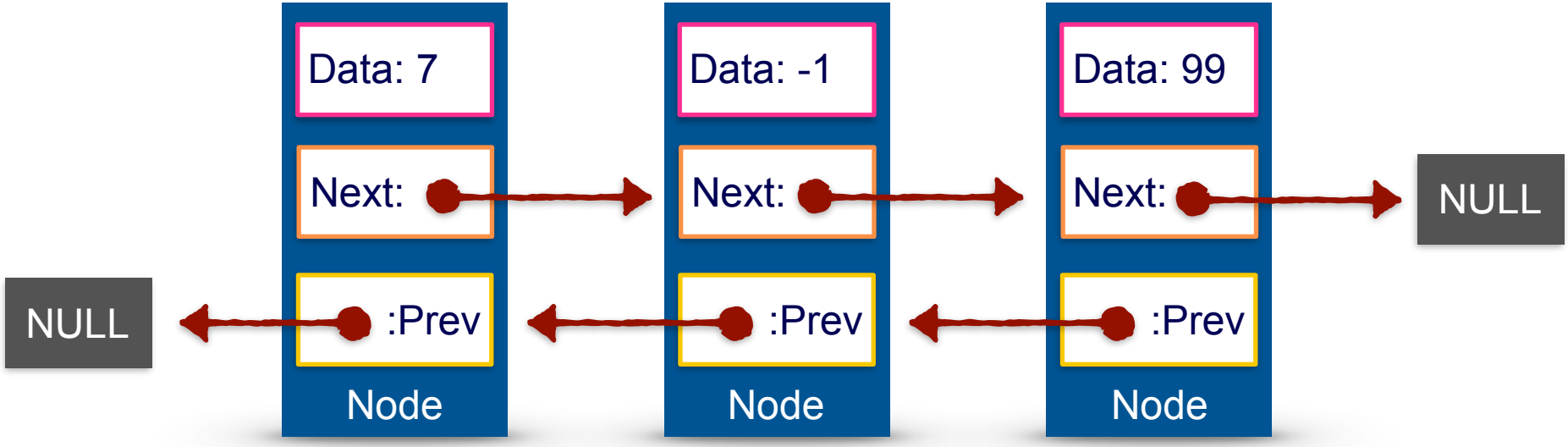
	Array/Vector	Linked List
Element Types	All elements are same type	All elements are same type
Resizeable	Yes Requires Copy	Yes
Storage Layout	Contiguous	Non-Contiguous
Cell lookup	Constant Time Random Access	Linear Time Random Access

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)

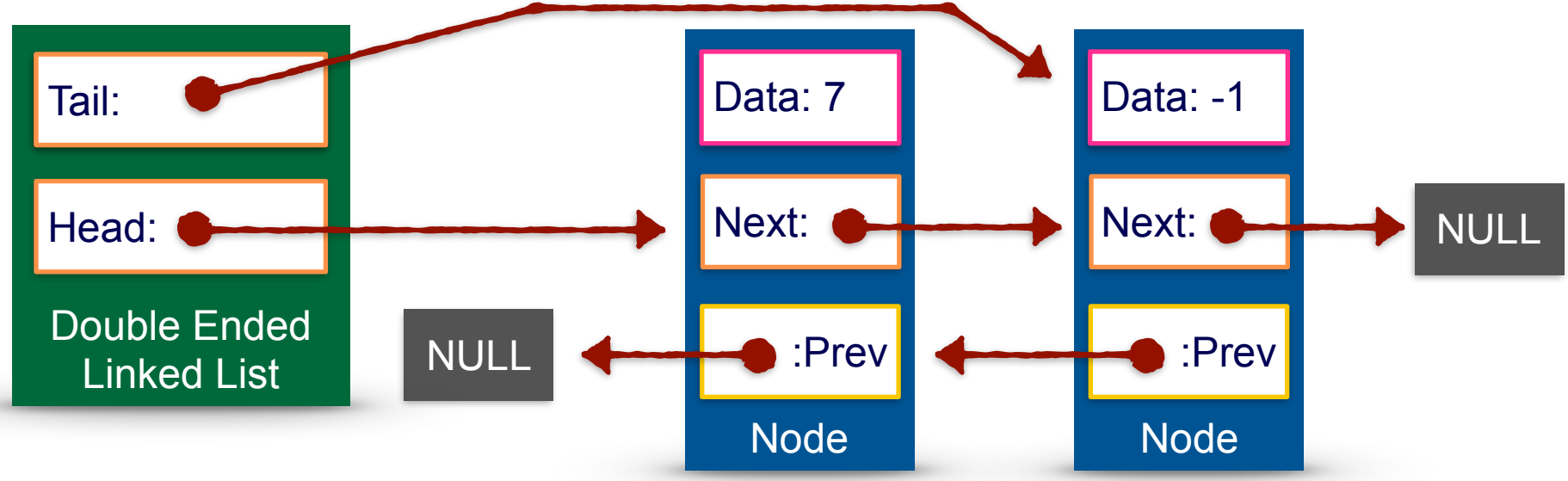
Double Ended Linked List

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



Double Ended 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 Double Ended Linked List

	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 Double Ended Linked List

	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

Review:
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

Recursion

RMIT UNIVERSITY

Recursion

- ***Recursion*** is a function or method that calls itself
  - It provides an alternative form of iteration
- Recursion makes heavy use of:
  - Scope
  - Program call stack
- Recursion is not suited to all problems
  - It is best suited to problems that are have identical subproblems

Writing Recursive functions

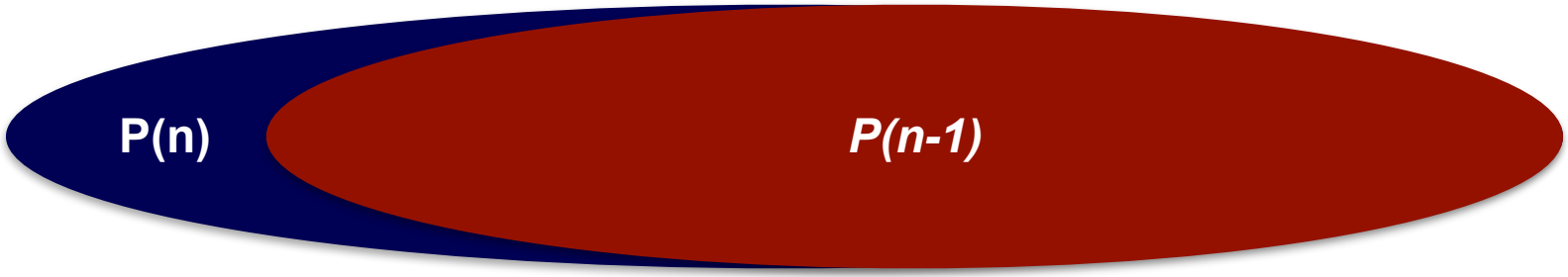
- It is almost impossible to use dynamic reasoning to think about recursive functions. That is,
  - Trying to “execute” the recursive function
  - Trying to “unravel” the recursive loop
- Instead, a ***static reasoning*** approach is far simpler, if not trivial. That is, ask questions such as:
  - What is true when the function is called?
  - What is true when the function returns?
  - If I get a result from a function, how can I modify it?
- Recursion is essentially mathematical ***induction is reverse***
  - BUT! You write it like writing an inductive proof in the forward direction!

Writing Recursive functions

- Recursive functions have two key parts:
    1. The base case(s)
    2. The Recursive/Inductive step
        - (I personally prefer the term inductive step, but this will often be written as the recursive step)
- If both parts are correctly implemented, then the function is correct.

Writing Recursive functions

- The “trick” is to use ***static reasoning***
  - Start with a problem, *P*, or size *n*, denoted *P(n)*
  - Find an identical sub-problem of *P* that is slightly smaller, such as *P(n-1)*
  - Assume you have the solution to *P(n-1)*
  - Transform the solution of *P(n-1)* into *P(n)*
  - Implement the base case, typically *P(0)* or *P(1)*
  - QED



$$P(n) = P(n-1) + ??$$

Practical use of Recursion

- In C++ recursive solutions are generally less efficient than iterative (loop) based implementations of the same algorithm
  - This is due to the overhead of setting up the call-stack
- However, practice with recursion is good for developing static reasoning skills
  - Further, iterative versions of recursive solutions are often more easily interpreted
- In other languages, recursion is often:
  - Far more efficient
  - Produces more easily interpretable code
  - The only option! (such as in functional and logic languages)

C-style Structs

RMIT UNIVERSITY

C-Struct

- A C-struct is a data structure that is very similar to a class
  - It contains a series of fields
  - All of the fields are public
- Structs and class function in a very similar manner
  - C++ classes essentially supersede structs
  - However, since these are still frequently used, it is useful to see them

```
struct test {
    int a;
    int b;
};
typedef struct test Test;
```

Data Structures

RMIT UNIVERSITY

Practical use of Recursion

- The C++ STL provides a number of containers to represent common and useful data structures
  - Sequence Containers
    - Array
    - Vector
    - Deque
    - List
  - Associative Containers
    - Set
    - Map
    - Tuple

Sequence Containers

- Store elements of the container is a defined sequence
  - The order of the container is important
  - The implementations guarantee the order of the container
- Each element is accessed one-at-a-time

Array

- Fixed sized array
  - Wrapper around a primitive C++ array
- Documentation: https://en.cppreference.com/w/cpp/container/array

```
std::array<int, 3> arr{ {1, 2, 3} };
arr[1] = 100;
cout << "Array [" << arr.size() << "]: " << arr[0] << endl;
```

Vector

- Dynamically sized array
  - Contiguous storage
  - Starts with a fixed sized array and grows if the array “runs-out” of space
    - Copies entire vector on expansion
  - Constant time random access
  - Constant time insertion at back
  - Linear time general insert/delete
- Documentation: https://en.cppreference.com/w/cpp/container/vector

```
std::vector<int> vec;
vec.push_back(1);
vec[0] = 100;
cout << "Vector [" << vec.size() << "]: " << vec[0] << endl;
```

Deque

- Double-ended Queue - Dynamically sized “array”
  - Non-Contagious storage
    - Conceptually uses a series of fixed-size arrays
  - Starts with a fixed size and grows if the array “runs-out” of space
    - Doesn’t copy entire deque on expansion
  - Constant time random access
  - Constant time insert front/back
  - Linear time general insert/delete
  - Compared to Vector, less efficient for looping through each item
- Documentation: https://en.cppreference.com/w/cpp/container/deque

Deque

- Double-ended Queue - Dynamically sized “array”
- Documentation: https://en.cppreference.com/w/cpp/container/deque

```
std::deque<int> deq;
deq.push_back(1);
deq.push_front(2);
deq[1] = 100;
cout << "Deque [" << deq.size() << "]: " << deq[0] << endl;
```

List

- Double Ended Linked List
  - Non-Contagious storage
  - Same performance and the Double Ended Linked List we have discussed in lectures
- Documentation: https://en.cppreference.com/w/cpp/container/list

```
std::list<int> list;
list.push_back(1);
list.push_front(2);
```

Associative Containers

- Provide relationships (associations) between elements of the container
- Containers are un-ordered
  - While the containers can be “looped through” to access all of the associations of the container, the implementations do not guarantee the order of the container
- Each element is accessed one-at-a-time

Set

- Set (as in the mathematical sense)
  - Each item in the set is unique
  - Requires the type of the set to have a well-defined operation for determining if two elements of the set are unique
  - Easy to work with primitive types or other STL elements
  - For user-defined classes, requires operator overloading (Week 10)
- Documentation: https://en.cppreference.com/w/cpp/container/set

```
std::set<int> set;
set.insert(1);
set.insert(2);
set.insert(1);
bool contain1 = set.find(1) != set.end();
cout << "Set [" << set.size() << "]: " << contain1 << endl;
```

Map

- Associates pairs of elements
  - Associates by key-value
    - The key is the primary entity
    - Each key has an associated value
  - Keys must be unique
    - Key work like elements of the set container
  - Values may not be unique
- Documentation: https://en.cppreference.com/w/cpp/container/map

```
std::map<int, std::string> map;
map[0] = "hello";
map[1] = "world";
cout << "Map [" << map.size() << "]: " << map[0] << endl;
```

Tuple

- Associates a group of elements together
  - Association has not defined “key”
  - Only associates 1 set of elements
    - Unlike a maps which is a container of multiple associations
  - In general, permits an *n*-tuple, where *n* is the user choice
  - Located in `<utility>` header file
- Documentation: https://en.cppreference.com/w/cpp/utility/tuple

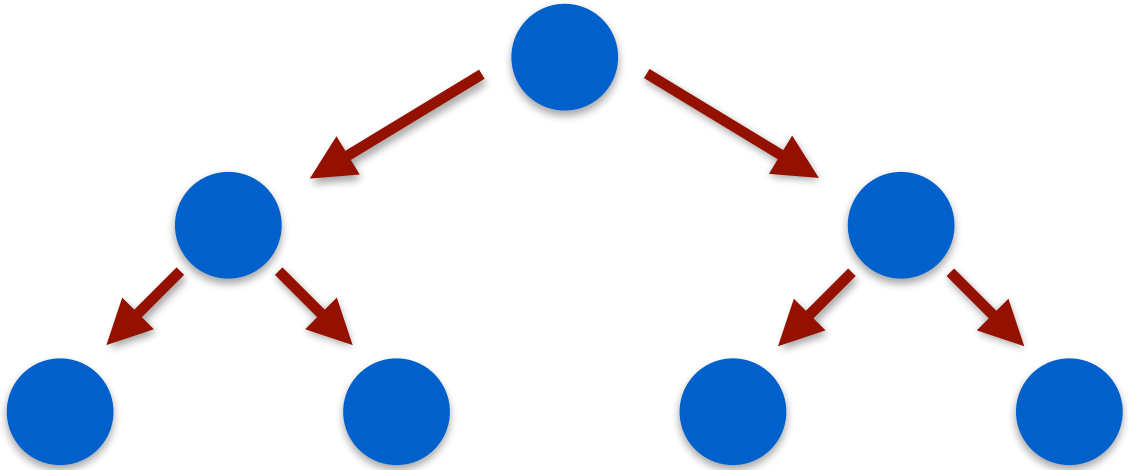
```
std::tuple<int, double, std::string> tuple(1, -3.3f, "str");
std::get<0>(tuple) = 10;
std::get<1>(tuple) = 5.6f;
std::get<2>(tuple) = "hello world";
cout << "3-Tuple: " << std::get<0>(tuple) << endl;
```

Binary Search Tree

RMIT UNIVERSITY

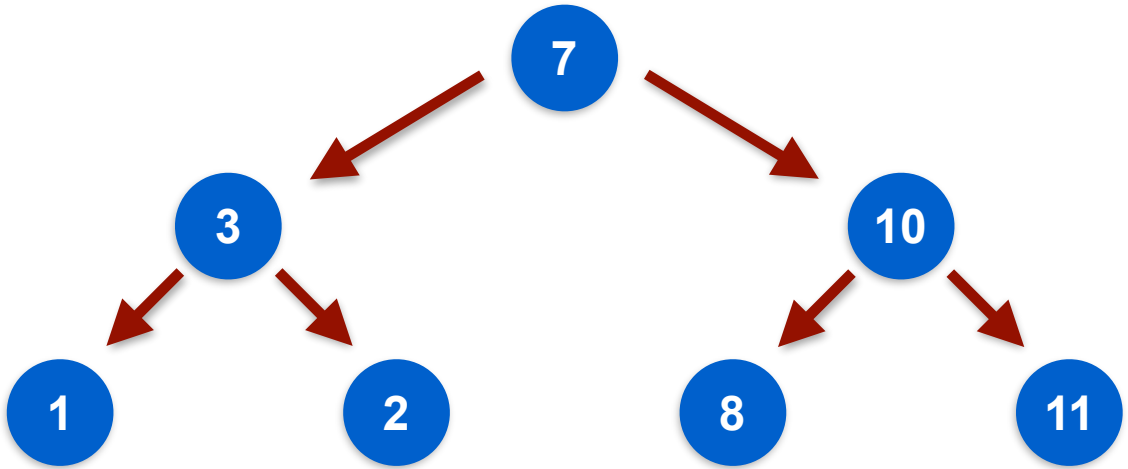
Binary Tree

- A ***Binary Tree*** is structure of nodes where each node:
  - Has two children, termed the left and right child
- If a node has no children, it is termed a ***leaf node***
- The first node is called the ***root node***
- It is termed a tree since that is what the structure looks like



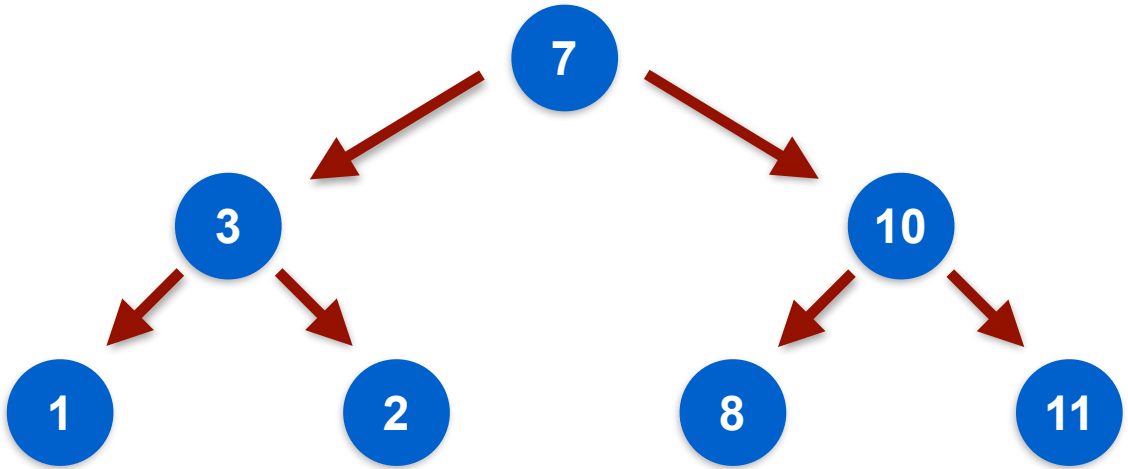
Binary Search Tree (BST)

- A ***Binary Search Tree (BST)*** is binary tree where
  - The value a node's left child is smaller than or equal to the node's value
  - The value a node's right child is greater than the node's value
- A BST is an ordered data structure



Binary Search Tree (BST)

- In general, it is very efficient at storing and retrieving values
  - Inserting or finding elements is done in logarithmic time
  - That is, the amount of work is effectively the depth of the tree



Assignment 2

RMIT UNIVERSITY

RMIT
UNIVERSITY