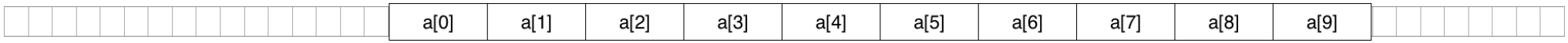
Ulster University

COM498 Algorithms & Data Structures

3.1 Pointers and Linked Chain Structures

Problems with Array Implementation

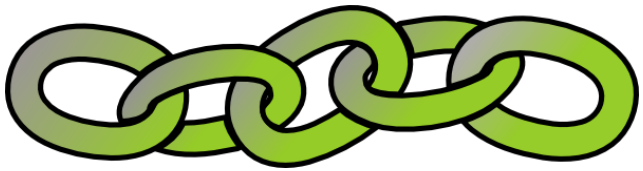
- Previously we used a fixed-size array to implement the **Bag** ADT



- Some (potential) issues with such an implementation:
  - Array has a fixed size
  - The array may become full
  - Alternatively may have wasted space
  - Resizing is possible but requires overhead of time

Linked Data Organisation

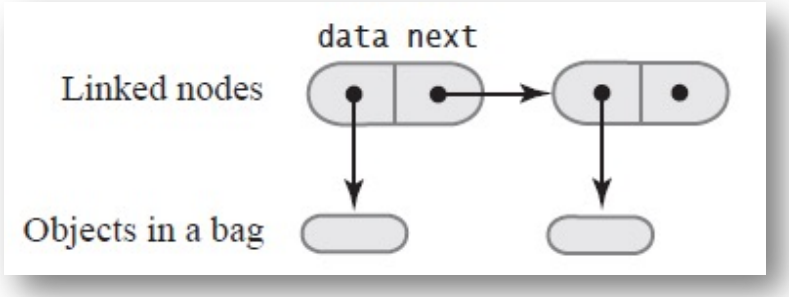
- The section introduces an implementation approach that uses memory only as needed (for a new entry) and returns unneeded memory to the system (after an entry is removed)
- By using a linked data organisation to implement the Bag ADT we avoid moving data when adding or removing bag entries



- A linked list (linked chain) is a linear data structure often used to implement other data structures (such as stacks, queues, trees)

Linked Data Organisation

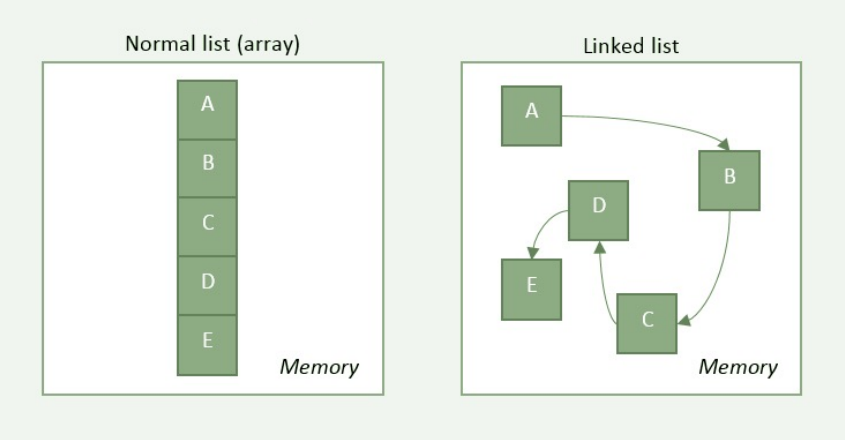
- The linked list (linked chain) is formed from a sequence of nodes
- Each node typically stores a reference to a piece of data (an entry in a bag) and a reference to another node (address of the next node in the chain)



- In a linked list the last node points to null which signifies the end of the chain

Linked List vs. Array

- A linked list is similar to an array in its approach to sequence and order
- Unlike an array, a linked list:
  - Is not restricted to a fixed-size number of elements
  - Is not stored contiguously in memory
  - Nodes can be inserted and removed without reallocation of memory



- Arrays are quicker at accessing elements
- Arrays are slower at inserting or removing elements
- A linked list can grow and shrink dynamically at run-time

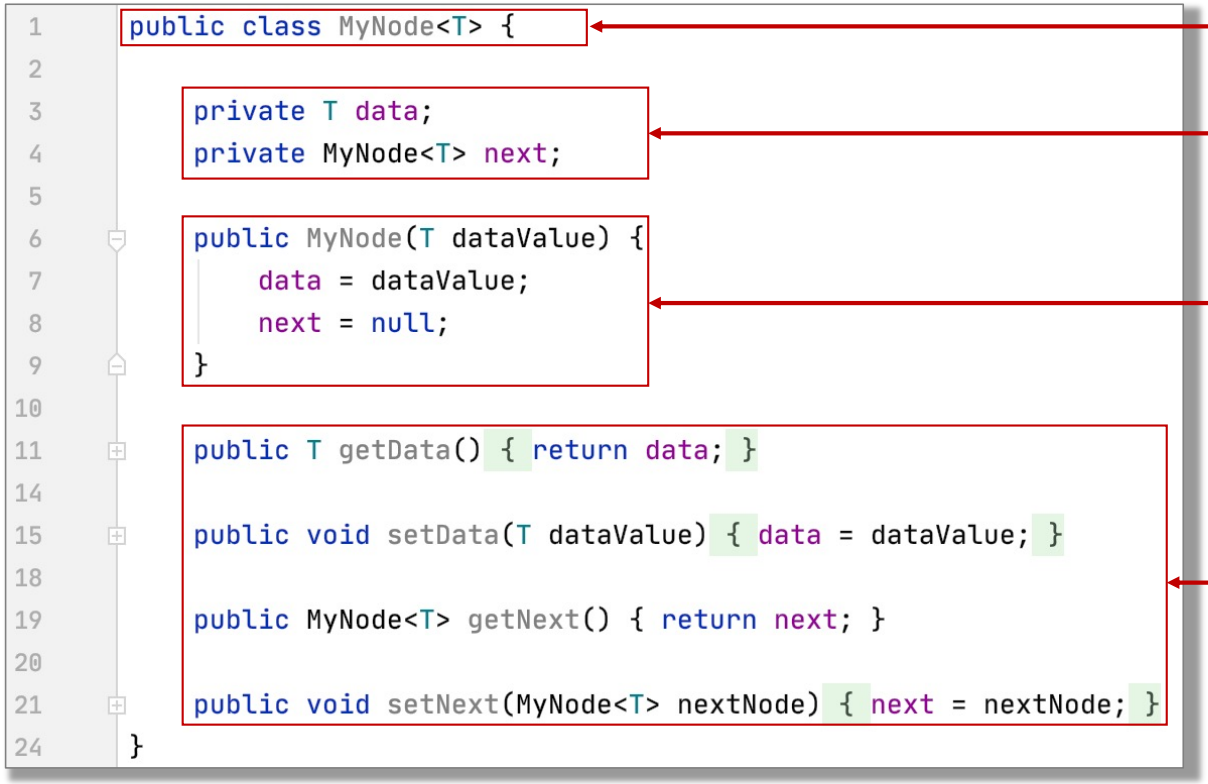
The Class `MyNode`

Ulster University

- To provide a linked implementation of the **Bag** ADT we first need to define a **Node**
- Private class with **two data fields**, **constructor** and both **accessor** and **mutator** methods
- Data field **data** contains a reference to one of the objects in the bag (uses generic type T)
- Data field **next** contains a reference to another node
- Constructor creates a new node setting the **data** field supplied and initialising the **next** field to `null`
- Accessor methods returns the values of the **data** and **next** fields
- Mutator methods set/update the values of the **data** and **next** fields

- Note: the class name used is `MyNode` to avoid confusion with the built-in Java class `Node`

The MyNode Class



```
1  public class MyNode<T> {
2
3      private T data;
4      private MyNode<T> next;
5
6      public MyNode(T dataValue) {
7          data = dataValue;
8          next = null;
9      }
10
11     public T getData() { return data; }
14
15     public void setData(T dataValue) { data = dataValue; }
18
19     public MyNode<T> getNext() { return next; }
20
21     public void setNext(MyNode<T> nextNode) { next = nextNode; }
24 }
```

Uses the Generic Type **T**

Instance variables – a data payload and a reference to the next node

Constructor to create a new Node

Public methods to

- return the data payload
- set the data payload
- return the next node reference
- set the next node reference

Scenario

- In your **Bag** class, create a new file *MyNode.java* and implement the `MyNode` class definition
- Test the definition in the `main()` method of `MyNode` by creating three nodes called `node1`, `node2` and `node3` with data values 1, 2 and 3.
- Set the **next** fields of the nodes so that `node1` points to `node2` and `node2` points to `node3`.
- Now, without referring directly to `node2` or `node3`,write code to print the values of all 3 nodes.