# Notes 16.0: Linked lists

COMP9021 Principles of Programming

*School of Computer Science and Engineering*
*The University of New South Wales*

2014 session 1

# Dynamic data structures

We often need to represent sequences of objects of the same nature that grow and shrink over time, such as sequences of natural numbers that might evolve from $(1, 3, 5)$ to $(1, 3, 5, 7)$ to $(1, 5, 7)$.

Arrays are appropriate to represent fixed sequences, but not so appropriate to represent dynamic sequences:

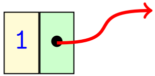
- If the array is larger than the number of objects in the sequence, space is wasted.
- When new objects join the list, the array might become too small, requiring a new array to be created and values in the old array to be transferred into the new one.
- ...

Linked lists offer the most fundamental data structure to represent dynamic sequences.

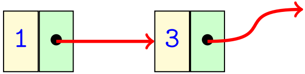
# Linked lists

Consider objects coded as data items of some type `Value` (e.g., `int`).

A structure of type say `Node` could have two members: a data item of type `Value` and a pointer...




...to a structure of type `Node`



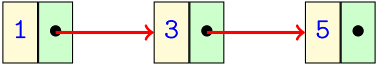

The end pointer could naturally be set to `NULL` to indicate the end of the list.

# Adding to a list

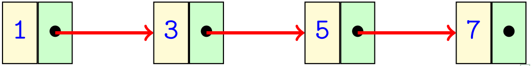
One key operation is to add some node to a list.

- We might want to add it only if it provides some information that is not already stored in the list, or we might welcome duplicates.
- We might want to add it to the front of the list, or to the end of the list, or we might not care.
- We might want to keep the nodes in a natural order, or we might not care about order.

For instance, adding a node 7 to the end of



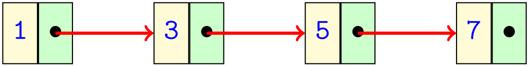

would yield



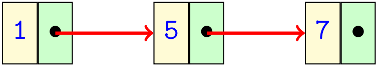
# Removing from a list

Another key operation is to remove a node from a list. If the same information can be represented in many nodes, we might want all nodes that store that information to be removed, or only one node.

For instance, removing the node 3 from




would yield



# From data types to abstract data types

- A data type is a set of properties and a set of operations. For instance, the type `int`
  - has the property of representing an integer value;
  - has six operations: changing the sign of an `int`, and adding, subtracting, multiplying, dividing, and taking the modulus of two `int`s.
- An abstract data type (ADT) is an abstract description of the properties of a type and of the operations that can be performed on it, that is independent of any implementation or programming language.
- A programming interface is designed, based on a choice of structures (how data will be represented) and functions (how data will be manipulated), for implementing the data type or the abstract data type.
- Code is written to implement the interface.

# Creating and implementing the interface

We create a header file tailored_list.h to declare the list data type. To protect against multiple inclusion of a file, the `#ifndef` directive is used.

C does not allow the data type to be purely abstract. We do the best we can by defining in tailored_list.h two types:

- `Value` with a default `typedef` expression which makes `Value` an alias for `int`, which can be changed by the user.
- `Node`, which we would hide from the user if we could.

A static function `are_same_nodes()` is meant to test whether two nodes store the same information; it has a default implementation for the default definition of `Node`.

We implement the operations in tailored_list.c. The function `assert()`, declared in `<assert.h>`, is used to abort the program when memory allocation fails.

# Using the interface

- The exact data representation of `Node` would be a detail of the implementation that would be invisible at the interface level if the data type could be made made purely abstract.
- Data hiding is achieved when the programmer can only know how to use functions of the interface, as described in tailored_list.h, without knowing anything about the internal representation of data.

The program test_tailored_list.c illustrates. It is meant to be compiled with tailored_list.c. As `Node` might be changed by the user of the interface, it cannot be precompiled.

# An alternative approach

An alternative is to define `Node` as a structure that can hold a pointer to `void`, to point to a datum of unspecified type, whose contents should not be accessed before the pointer to `void` has been cast to a pointer to the right type, and to define a `List` as a structure with three fields:

- a pointer to the first node;
- the number of bytes to allocate at each datum address;
- a pointer to a function that can compare two data.

This is illustrated with the files generic_list.h, generic_list.c and test_generic_list.c.

We use the same interface for this alternative approach.

The price to pay is that we do not store data, but pointers to data, which incurs a cost and is awkward for data of primitive type.