Data Structures and Abstract Data Types

Programming Fundamentals

Dr Kevin Chalmers

School of Computing
Edinburgh Napier University
Edinburgh

k.chalmers@napier.ac.uk

Notes

Outline

Notes

Outline

Notes

Basic Data Types

- Many languages support a range of basic (primitive) data types. For example:
  - `int`, `float`, `long`, `double` (`signed`, `unsigned`)
  - `char`, string
  - pointer, reference
  - `bool`
- Exact set of types depends upon the language.
  - Lots of discussion about both what types are necessary and how they should behave (typing systems).
- Many programs can be written using just basic data types.

Notes

Composite Data Types

- As soon as we need to work with more data instead of individual instances we begin considering collections.
  - Arrays (linear collection of a basic data type).
  - Records:
    - Structs (collection of basic data types, including other structs).
    - Classes (collections + encapsulation, information hiding, inheritance, polymorphism).
- Often for prototyping or simple programs is all we need.
- Generally quick to implement and can make a lot of progress with just these simple collections.
- But what happens when our data becomes high volume and/or complex or we need specific performance guarantees?
- Always have to make a trade-off between competing aspects:
  - Performance, storage, maintainability.

Notes

Questions?

Notes

Outline

Notes

Data Structures

- More complex collections of data.
  - For example, what you have been working on in the coursework.
- Could just combine arrays, structs, and classes in an *ad hoc* manner to suit our problem . . .
  - . . . but many programmers have already done this . . .
    - . . . and they notice some things.
- There are recognisable and repeating patterns for organising data.
  - Note - relate this to **Design Patterns** (important topic in object-orientated design which you will study in Software Development 2).
- We can investigate these patterns and learn their characteristics under a variety of circumstances . . .
  - . . . then we can learn to use them most effectively.

Notes

Abstract Data Types

- Important to distinguish between the functional definition of a data structure (abstract) and it's implementation (concrete).
- An ADT can be considered in terms of *elements* and a set of *operations* on those elements.
- An ADT may often be implemented (made concrete) in a number of ways - which can effect the performance characteristics of the resulting structure.
- ADTs may even be implemented in terms of other ADTs.

Notes

Why Data Structures?

- Selecting the right (or wrong) data structure will have an impact on your program.
  - Note - if you want to understand and handle that impact then you will need to profile or otherwise make measurements of your program.
- Many data structures have known characteristics which we can learn about . . .
  - . . . and infer their effects upon our programs.
  - Note - without metrics we can't know for sure (and we will rarely be **optimal**) but with experience we get a feel for good initial approaches.

Notes

Effects

- Performance:
  - Memory consumption.
  - Speed (time to add data, remove data, search for data).
  - Consistency (does performance alter in relation to size or complexity of data?)
- Ease of understanding and longer term maintainability.
- Note - there is a reason that people usually refer to data structures **and algorithms**. The two often work together. Often their are particular algorithms that will give specific performance characteristics when used with particular data structures.
  - You will study data structures and algorithms in your third year.

Notes

Questions?

Notes

Outline

Notes

Linear Lists

- Most basic ADT.
- Linear lists (or just lists).
- An ordered sequence of elements of length `n`:

$$< a_1, a_2, \ldots, a_n >$$

- Come in a variety of types:
  - Sequential allocation (like `ArrayList` in Java, and `vector` in C++).
  - Linked lists (used in functional languages (Haskell), `list` data type in C++).
    - Internal.
    - External.
    - Multilists (sparse matrices).
- Note - abstract, so we won't discuss the **type** of the elements now.

Notes

Lists

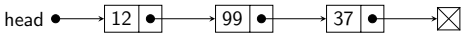
- Generally supports operations such as:

`get(i)`	return element a_i (remember operator overloading as well).
`set(i, x)`	set the element `i` to `x`.
`length()`	return the list length.
`insert(i,x)`	add element `x` just before element a_i.
`delete(i)`	remove element `i` from the list and update all subsequent indices.
`others`	potential operations such as searching, splitting, concatenating, sublists, empty, etc.

Notes

Implementing Lists

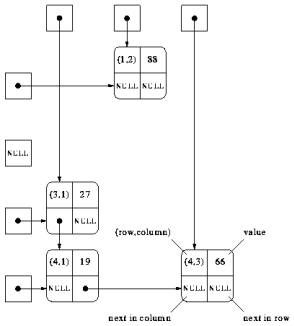
- Most important question - how should we implement our list?
  - Sequential allocation (basically using arrays).
  - Linked allocation (basically a linked list - below).
    - Singly or double linked? Circularly linked?
    - Internally linked or externally linked list.
    - Does the node containing the link(s) also contain the data that the structure is supposed to hold?



Notes

Multilists and Sparse Matrices

- Lists are very basic data structures.
- But they can be combined in non-trivial ways.
  - Create multilists (2 or more sets of inter-linked lists).
  - Can be used to represent a sparse matrix.



Notes

Questions?

Notes

Outline

Notes

Stack

- **Don't confuse with memory stack - same operations but different purpose**.
- A special kind of list.
- Can insert data onto the top of the list (push).
- Can delete data from the top of the list (pop).
- A key data structure:
  - Processing tree structures.
  - Nested structures.
  - Implementing recursion.
  - Compiler and parser writing.
- When data is popped off the stack it is in reverse order to the way it went in.
  - Last In, First Out (FILO) ordering.

Notes

Queue

- Another special kind of list.
- Supports insertion (enqueue) at one end (the tail).
- Supports deletions (dequeue) from other end (the head).
- Another core data structure used in operating systems and networking:
  - Store a list of items that are waited to be processed in order.
- First In, First Out (FIFO) ordering.

Notes

Deque

- Play on words.
- d-e-que - double-ended queue.
- Pronounced deck
  - Acts like a deck of cards.
  - You can deal from the top or the bottom.
- Supports insertions and deletions from either end.

Notes

Questions?

Notes

Outline

Notes

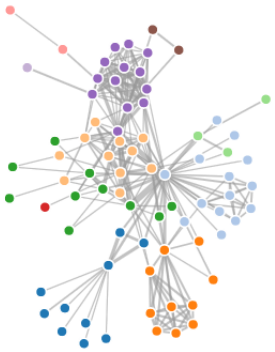
Sets

- A collection of elements that are:
  - Unordered.
  - Unique (no repeated values).
- Operations:
  - Add
  - Remove
  - Is empty
  - Size
  - Is element of
  - Union
  - Intersection
  - Difference
  - Subset

Notes

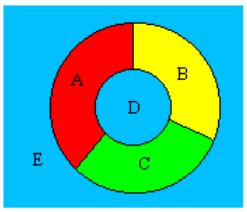
Graphs

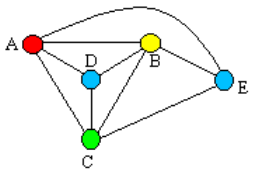
- A collection of elements (nodes, vertices) and links between them (edges).
- If we don't care about the direction of the edge then the graph is *undirected*.
- Used *a lot* in mathematics and very popular in computing due to flexibility.
  - Whenever you have a set of elements and a relation between pairs of elements.



Notes

Map Colouring

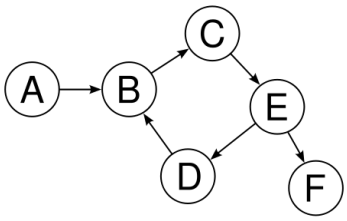




Notes

Digraphs

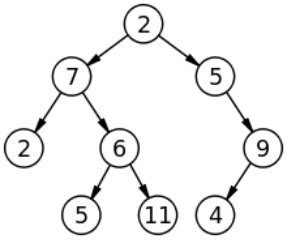
- Directed graphs.
- The edges are *ordered* pairs.
  - We have a convention on which direction the edge *goes* in.
  - $(A, B), (B, C), (C, E), (E, D), (E, F), (D, B)$ below.
- Can include self-loops (but sometimes prohibited).



Notes

Trees

- Tree are a sub-type of graph (in terms of primitive definition).
- Directed graphs in which there are no cycles and each node may have only one *parent* (node pointing to it).
- Many types of trees and many algorithms for traversing them.
- Very popular for storing hierarchical data.



Notes

Questions?

Notes

Outline

Notes

Summary

- We've covered quite a bit of material, a lot of which you will return to over the next two years of your studies.
- We looked at basic data types.
- We discussed basic data structures.
- We introduced the concepts of Abstract Data Types (ADTs).
- Introduced a good selection of core ADTs.

Notes

To do...

That's it. The module is coming to an end. You have coursework and exams to do, and you should try and complete as much of the work as possible.
Don't let the holidays get in the way of your practice. You need to develop your programming skills. Pick something you want to develop and keep working. *Become a software crafts person*. It takes time and practice to develop these skills, but it will pay off in the end.

Notes