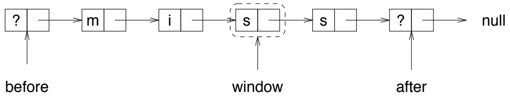
CITS2200 Data Structures and Algorithms

Topic 1

Introduction to Data Structures

- Why study data structures?
- Collections, abstract data types (ADTs), and algorithm analysis
- More on ADTs
- What's ahead?



1. What are Data Structures?

- Data structures are software artifacts that allow data to be stored, organized and accessed.
- They are more high-level than computer memory (hardware) and lower-level than databases and spreadsheets (which associate meta-data and meaning to the stored data).
- Ultimately data structures have two core functions: put stuff in, and take stuff out.

Why?

- software is complex
  — more than any other man made system
  — even more so in today's highly interconnected world
- software is fragile
  — smallest logical error can cause entire systems to crash
- neither you, nor your software, will work in a vacuum
- the world is unpredictable
- clients are unpredictable!

Software must be correct, efficient, easy to maintain, and reusable.

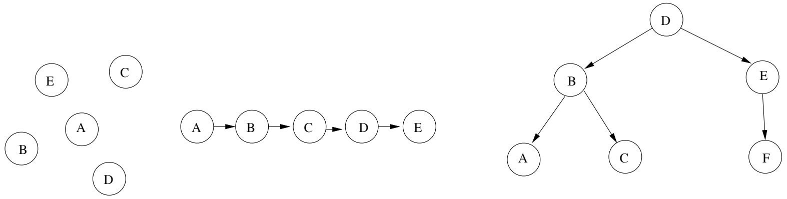
2. What will we Study?

2.1 Collections

...as name suggests, hold a bunch of things...

"nearly every nontrivial piece of software involves the use of collections"

Seen arrays — others include queues, stacks, lists, trees, maps, sets, tables...



Why so many?

Space efficiency

Time efficiency:

- store (add to collection)
- search (find an object)
- retrieve (read information)
- remove or replace
- clone (make a copy)

2.2 Abstract Data Types

Allow user to *abstract* away from implementation detail.

Consider the statement: *I put my lunch in my bag and went to Uni.*

What is meant by the term *bag* in this context?

Most likely it is a *backpack*, or *satchel*, but it could also be a *hand bag*, *shopping bag*, *sleeping bag*, *body bag* ...(but probably not a *bean bag*).

It doesn't actually matter. To parse the statement above, we simply understand that a *bag* is something that we can

1. put things in,
2. carry places, and
3. take things out.

Such a specification is an *Abstract Data Type*.

2.3 Algorithm Analysis

We will consider a number of alternative implementations for each ADT.

Which is best?

Simplicity and Clarity

All things being equal we prefer simplicity, but they rarely are...

Space Efficiency

- space occupied by data — overheads
- space required by algorithm (eg recursion) — can it blow out?

Time Efficiency

Time performance of algorithms can vary greatly.

Example: Finding a word in the dictionary

Algorithm 1:

- Look through each word in turn until you find a match.

Algorithm 2:

- go to half way point
- compare your word with the word found
- if $<$ repeat on earlier half
  else $>$ repeat on later half

Performance

Algorithm 1 (exhaustive search) proportional to $n/2$

Algorithm 2 (binary search) proportional to $\log n$

number of words	Algorithm 1 max. comparisons	Algorithm 2 max. comparisons
10	10	4
100	100	7
1000	1000	10
10000	10000	14
100000	100000	17
1000000	1000000	20

2.4 ADTs and Java

Object-oriented programming was originally based around the concept of abstract data types.

Java classes are ideal for implementing ADTs.

ADTs require:

- Some *references* (variables) for holding the data
  (usually hidden from the user)
- Some *operations* that can be performed on the data
  (available to the user)

A class in Java has the general structure. . .

class declaration

variable declarations `// data held`
.
.
.

method declarations `// operations on the data`
.
.
.

2.5 Information Hiding

- Variables can be made `private`
  — no access by users
- Methods can be made `public`
  — used to create and manipulate data structure

This *encapsulation* is good programming practice

— can change

- the way the data is stored
- the way the methods are implemented

without changing the (external) *functionality*.

Example: A Matrix Class

```
public class Matrix {

  private int[][] matrixArray;

  public Matrix (int rows, int columns) {
    matrixArray = new int[rows][columns];
    for (int i=0; i<rows; i++)
      for (int j=0; j<columns; j++)
        matrixArray[i][j] = 0;
  }
```

```
  public void set (int i, int j, int value) {
    matrixArray[i][j]=value;
  }

  public int get (int i, int j) {return matrixArray[i][j];}

  public void transpose () {
    int rows = matrixArray.length;
    int columns = matrixArray[0].length;
    int[][] temp = new int[columns][rows];
    for (int i=0; i<rows; i++)
      for (int j=0; j<columns; j++)
        temp[j][i] = matrixArray[i][j];
    matrixArray = temp;
  }
```

Q: What is the time performance of `transpose()`?

For a matrix with n rows and m columns, how many (array access) operations are needed?

Can you think of a more efficient implementation? One that doesn't move any data?

```
public class MatrixReloaded {

  private int[][] matrixArray;
  private boolean isTransposed;

  public MatrixReloaded (int rows, int columns) {
    matrixArray = new int[rows][columns];
    for (int i=0; i<rows; i++)
      for (int j=0; j<columns; j++)
        matrixArray[i][j] = 0;
    isTransposed = false;
  }
  public void set (int i, int j, int value) {



  }

  public int get (int i, int j) {



  }

  public void transpose () {



  }
}
```

What is the time performance of `transpose()`?

Does it depend on the size of the array?

How do the changes affect the *user's* program?

2.6 Advantages of ADTs

- modularity — independent development, re-use, portability, maintainability, upgrading, etc
- delay decisions about final implementation
- separate concerns of application and data structure design
- information hiding (encapsulation) — access by well-defined interface

Also other OO benefits like:

- polymorphism — same operation can be applied to different types
- inheritance — subclasses adopt from parent classes