CITS2200 Data Structures and Algorithms

Topic 1

Welcome

Dr. Tim French

Room 2.14
tim@csse.uwa.edu.au

A unit about *space*, *time*, and *integrity*.

Handbook Description

At the core of most computer applications is the storage and retrieval of information. The way that the stored data is structured has a strong impact on *what can be retrieved*, *how quickly it can be retrieved*, and *how much space it occupies*. The use of generic structures, or abstract data types (ADTs), to encapsulate the data also allows *software engineering principles* of independent modification, extension and reuse.

This unit studies the *specification, implementations and time and space performance of a range of commonly-used ADTs* and corresponding algorithms in an object-oriented setting. The aim is to provide students with the background needed both to implement their own ADTs where necessary, and to select and use appropriate ADTs from object-oriented libraries where suitable.

This Lecture

- Introductory information — teaching sessions, teaching staff, assessment, lab rules, unit software, on-line resources, teaching and learning agreement
- Introduction to ADT's

  Wikipedia disambiguation:

  1. *Abstract data type*: a computer programming term
  2. *Asynchronous data transfer*: a method of transferring data
  3. *Automatic double tracking*: an audio recording technology invented for The Beatles
  4. *American Discovery Trail*: a coast-to-coast hiking trail across the mid-tier of the United States.
  5. *Average Daily Traffic*: to show the volume of traffic on a road in transportation planning
  6. *Active Denial Technology*: also known as the *pain ray*

Timetable

- **Lectures**
  - 2pm-3pm Tuesdays, Austen Lecture Theatre
  - 10pm-11am Fridays, Engineering Lecture Theatre 2
- **Tutorials**
  - 2pm-3pm Thursdays, Engineering Lecture Theatre 2
- **Laboratories**
  - 10am-12pm Thursdays, CSSE Lab 2.01
  - 12pm-2pm Thursdays, CSSE Lab 2.01
  - 3pm-5pm Thursdays, CSSE Lab 2.01
  - **Only the first hour of each lab is supervised**
- **Consultation**
  - Tim: 1pm-2pm Thursdays, CSSE Room 2.14

Help Forum

Aside from the aforementioned activities, you may also get help via the `help2200` electronic forum — a public discussion board for all queries relating to the unit.

Assessment

Assessment	Date	% of Final Mark
Laboratory work	Selected weeks, starting Week 4	10%
Mid-semester test	Tuesday, Week 8	10%
Project	Weeks 10-13	20%
Final examination	June examination period	60%

References

Cara MacNish & Tim French, *Data Structures and Algorithms Notes*, 2008.

Further Reading

There are many different books on the subject of data structures as well as books on the subject of Java, including some which combine the two. A few examples of books worth looking at include:

Kenneth Lambert and Martin Osborne, *Java: A Framework for Program Design and Data Structures*, 2nd Ed, Thomson, 2004 (previous textbook).

Thomas Cormen, Charles Leiserson, Ronald Rivest and Clifford Stein, *Introduction to Algorithms*, 3rd Ed, MIT Press, 2009 (CITS3210 textbook).

Sartaj Sahni, *Data Structures, Algorithms, and Applications in Java*, McGraw Hill, 2000.

Mark Weiss, *Data Structures and Problem Solving using Java*, Addison Wesley, 1998.

Mark Weiss, *Data Structures and Algorithm Analysis in Java*, Addison Wesley, 1999.

Adam Drozdek, *Data Structures and Algorithms in Java*, Thomson, 2005.

Michael Gooodrich and Roberto Tamassia, *Data Structures & Algorithms in Java*, Wiley, 1998.

Robert Lafore, *Data Structures and Algorithms in Java*, Wiley, 2005.

Janet Prichard and Frank Carrano, *Data Abstraction & Problem Solving with Java: Walls & Mirrors*, 3rd Ed, Addison Wesley, 2011.

Topics Of Study

We will study the following topics this semester:

1. Introduction	2. Java Revision
3. Recursive Data Structures	4. Abstract Data Types
5. Queues and Stacks	6. Complexity Analysis
7. Objects and Iterators	8. Lists
9. Maps and Binary Search	10. Trees
11. Sets, Tables, and Dictionaries	12. Priority Queues
13. Hash Tables	14. Revision

What You Should Do This Week

1. Get set up to use the School's Computer Systems:
   `https://secure.csse.uwa.edu.au/run/csentry?pw1=yes`
2. Begin to familiarise yourself with the Unit's web site.
3. Have a go at *An Introduction to MacOSX* and the first labsheet.
4. Start brushing up your Java in preparation.

Laboratory and tutorial classes start Thursday, Week 2.

CITS2200 Data Structures and Algorithms

Topic 1

Introduction

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

Reading: Lambert and Osborne, Sections 1.1–1.6

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.

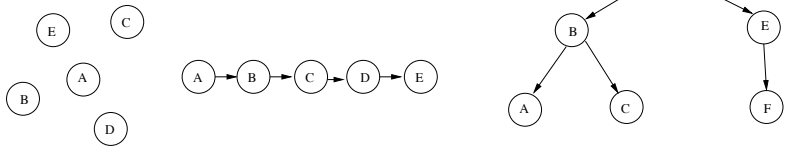
2. What will we Study?

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

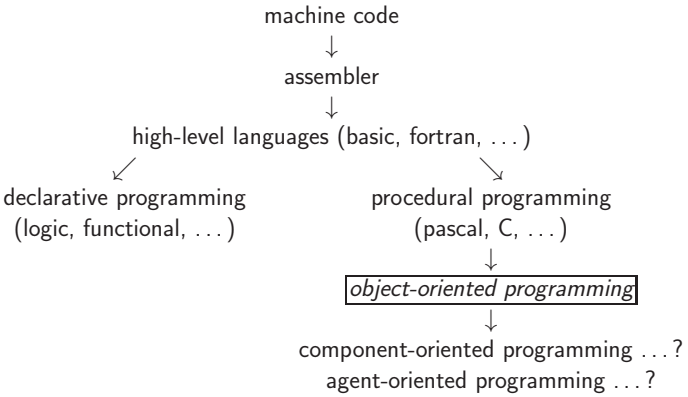
Algorithm 1 (exhaustive search) proportional to $n/2$

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 History

The evolution of programming



- machine code: simple instructions; simple data (1's and 0's)
- assembler: mnemonics and variables; high-level constructs: instructions, data types (ints, chars etc);
- procedural: away from sequences to blocks of code; more structured data (records etc);
- declarative: what, not how;
- Object-oriented: "autonomous" objects or data-types; access through an interface consisting of operations (create, add-to, read, destroy); don't know or care about internal workings, but passive;
- agents: more autonomous, more active; may have intentions and desires; may be able to initiate communications themselves etc,

2.5 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`

.
.
.

2.6 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?

2.7 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

3. What's Ahead?

Specifying, designing, implementing, analysing, and selecting ADTs for collections.

- what data structures are most appropriate for what kinds of tasks
- what choices are available for representing ADTs and what are the trade-offs
  - time efficiency
  - space efficiency
  - flexibility/constraints — bounded vs unbounded etc

3.1 Structure of the Course

Part 1 Foundations, Analysis, and Implementations.

- Introduction
- Java concepts
- Review of recursion and introduction to recursive data structures
- Data abstraction and ADT specification
- Examples of ADTs — *Stacks, Queues, and Lists*
- Performance analysis for data structures
- Objects, Iterators, and Generics

Part 2 Practical Data Structures

- *Maps*
- *Trees*
- *Priority Queues*
- *Sets*, *Tables*, and *Dictionaries*
- *Hash Tables*

CITS2200 Data Structures and Algorithms

Topic 2

Java Primer

- Review of Java basics
- Primitive vs Reference Types
- Classes and Objects
- Class Hierarchies
- Interfaces
- Exceptions

Reading: Lambert and Osborne, Appendix A & Sections 1.2 and 2.1–2.7

1. Review of Java Basics

1.1 Primitive Data Types

Type	default value	size	range
byte	0	8 bits	-128 to 127
short	0	16 bits	-2^{15} to $2^{15}-1$
int	0	32 bits	-2^{31} to $2^{31}-1$
long	0L	64 bits	-2^{63} to $2^{63}-1$
float	0.0f	32 bits	?
double	0.0d	64 bits	?
char	'\u0000'	16 bits	$0-2^{16}$
boolean	false	?	{true, false}

1.2 Local Variables

Scope: block in which defined

```
for (int i=0; i<4; i++) {
  // do something with i
  }
System.out.println(i);
```

Result?

1.3 Expressions

Built from variables, values, and *operators*.

arithmetic: `+, -, *, /, %,...`

logical: `&&, ||, !,...`

relational: `=, !=, <, >, <=, >=,...`
`==, !=, equals`
`instanceOf`

ternary: `?` (e.g. `x > 0 ? x : -x`)

1.4 Control Statements

if and if-else

```
if (<boolean expression>)
  <statement>

if (<boolean expression>)
  <statement>
else
  <statement>
```

where <statement> is a single or compound statement.

while, do-while, and for

```
while (<boolean expression>)
  <statement>

do
  <statement>
while (<boolean expression>)

for (<initialiser list>; <termination list>; <update list>)
  <statement>
```


Example

```
for (int i=0; i<4; i++) System.out.println(i);
```

```
0
1
2
3
```

```
for (String s=""; !s.equals("aaaa"); s=s+"a")
  System.out.println(s.length());
```

In Java 5 we also have an enhanced for loop:

```
int[] array = {0,2,4};
for (int i : array)
  System.out.println("i is: " + i);
```


Arrays

Declaration

```
<type>[] <name>;
<type>[]...[] <name>;
```

Instantiation

```
<name> = new <type>[<int-exp>];
<name> = new <type>[<int-exp>]...[<int-exp>];
```

Example

```
int[][] matrixArray;

matrixArray = new int[rows][columns];

int[] array = {0,2,4};
```

1.5 Methods

Methods have the form (ignoring access modifiers for the moment)

```
<return type> <name> (<parameter list>) {
  <local data declarations and statements>
}
```

Example

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

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


Parameters are *passed by value*:

```
// a method...
void increment (int i) {i++;}

// some code that calls it...
i=7;
increment(i);
System.out.println(i);
```

Result?

2. Primitive Types vs Reference Types

Primitive types

- fixed size
- size doesn't change with reassignment

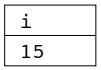
⇒ store *value* alongside variable name

Reference types (eg. Arrays, Strings, Objects)

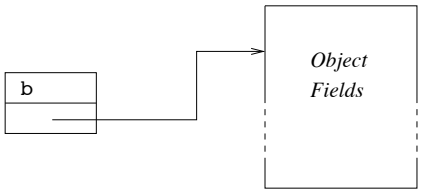
- size may not be known in advance
- size may change with reassignment

⇒ store *address* alongside variable name


```
integer i = 15;
```




```
Object b = new Object();
```



The variable holds a pointer or *reference* to the object's data

⇒ *reference types*

```
int[] a = {0,1,2,3};
int[] b = a;
b[0]++;
System.out.println(a[0]);
```

Result?

```
// a method...
void incrementAll (int[] a) {
  for (int i=0; i<a.length; i++) a[i]++;
}

// some code that calls it...
int[] b = {0,1,2,3};
incrementAll(b);
System.out.println(b[0]);
```

Result?

3. Classes and Objects

3.1 What are they?

Aside from a few built-in types (arrays, strings, etc) all reference types are defined by a *class*.

A class is a chunk of software that defines a type, its attributes or *instance variables* (also known as *member variables*), and its *methods*...


```
class Box {

  // instance variables
  double width, length, height;

  // constructor method
  Box (double w, double l, double h) {
    width = w;
    length = l;
    height = h;
  }

  // additional method
  double volume () {return width * length * height;}
}
```


3.2 Constructors

The runtime engine creates an *object* or *instance* of the class each time the `new` keyword is executed:

```
Box squareBox, rectangularBox;
...
squareBox = new Box(20,20,20);
rectangularBox = new Box(20,30,10);
```

3.3 Different Kinds of Methods

constructor — tells the runtime engine how to initialise the object

accessor — returns information about an object's state without modifying the object

mutator — changes the object's state

3.4 Packages

A collection of related classes. E.g. `java.io`

In Java:

- must be in same directory
- directory name matches package name

Specifying your own package

```
package myMaths;

class Matrix {
  ...
```

If you don't specify a package Java will make a default package from all classes in the directory.

Using someone else's package

```
package myMaths;
import java.io.*;

class Matrix {
  ...
```

Note that `java.lang.*` is automatically imported.

3.5 Access Modifiers

Specify access to classes, variables, and methods.

public — accessible by all

private — access restricted to within class

(none) — access restricted to within package

protected — access to package and subclasses

3.6 The `static` Keyword

Used for methods and variables in classes that *don't* create objects, or for variables *shared* by all instances of a class.

Example:

```
public class MatrixTest {

   static Matrix m;

  public static void main (String[] args) {
    m = new Matrix(2,2);
    m.set(0,0,1);
    ...
```

Called *class variables* and *class methods*.

Also used for "constants".

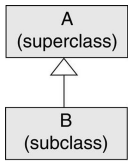
Example:

```
public class Matrix {

  static final int MAX_SIZE=100;

  private int[][] matrixArray;
  ...
```

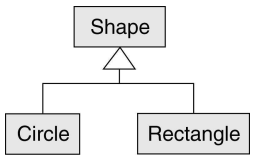
Keyword `final` means the value cannot be changed at runtime.

4. Class Hierarchies

Classes can be built from, or *extend* other classes.



Example:




```
public class Shape {

  private double xPos, yPos;

  public void moveTo (double xLoc, double yLoc) {
    xPos = xLoc;
    yPos = yLoc;
  }

  ...
}
```

(More detail: see Lambert and Osborne, Section 2.5)

```
public class Circle extends Shape {

  private double radius;

  public double area () {
    return Math.PI * radius * radius;
  }
}
```

While we will not be using hierarchies extensively in this unit, we will be using some *very important features of them*...

1. Any superclass reference (variable) can hold and access a subclass object.

Example:

```
public class ShapeTest {

  public static void main (String[] args) {
    Shape sh;            // declare reference of type Shape
    sh = new Circle();   // hold a Circle object in sh
    sh.moveTo(2.0,3.0);  // access a Shape method
    double a=sh.area();  // access a Circle method
    ...
```


2. All Java classes are (automatically) subclasses of `Object`

Example:

```
Object holdsAnything;
holdsAnything = new Circle();
holdsAnything = new Rectangle();
holdsAnything = new Shape();
```

Example:

```
Object[] arrayOfAnythings = new Object[10];
arrayOfAnythings[0] = new Circle();
arrayOfAnythings[1] = new Rectangle();
arrayOfAnythings[2] = new Shape();
```


Java provides *wrappers* for all primitives to allow them to be treated as `Objects`:

⇒ `Character, Boolean, Integer, Float, ...`

See the Java API for details.

Note: A new feature in Java 1.5 is *autoboxing* — automatic wrapping and unwrapping of primitives.

⇒ Compile time feature — doesn't change what is "really" happening.

4.1 Casting

While a superclass variable can be assigned a subclass object, a subclass variable cannot be assigned an object held in a superclass, *even if that object is a subclass object*.

Example:

```
Object o1 = new Object();            // OK
Object o2 = new Character('a');      // OK
Character c1 = new Character('a');   // OK
Character c2 = new Object();         // Error

o1 = c1;                             // OK
c1 = o1;                             // Error
```


In the last statement, even though `o1` is now "holding" something that was created as a `Character`, its reference (ie its class) is `Object`.

To get the "`Character`" back, we have to *cast* it back down the hierarchy:

```
o1 = c1;                             // OK
c1 = (Character) o1;                 // OK - casted back to Character
```


4.2 Object Oriented Programming in Java

Some object oriented features worth remembering are:

- **Abstraction**: the ability to treat different types of object as a common type.
- **Polymorphism**: how one method can change its behavior in different classes.
- **Inheritance**: reusing methods and variables from super classes.
- **Encapsulation**: information hiding, and containing other classes.

To demonstrate these properties, let's reconsider the `Shape` example. This time, we first define a class `Point`...

```
public class Point {

  private double xPos, yPos;

  public Point(double x, double y){
    xPos = x;
    yPos = y;
  }

  public void moveTo (double xLoc, double yLoc) {
    xPos = xLoc;
    yPos = yLoc;
  }
}
```

...and use Point to define Shape. This is **encapsulation**.

```
public abstract class Shape {

  private Point p;

  public Shape(double x, double y) {
    p = new Point(x,y);
  }

  public void moveTo (double xLoc, double yLoc) {
    p.moveTo(xLoc, yLoc)
  }

  public double area();     //an abstract method - more later
}
```


Circle inherits from Shape...

```
public class Circle extends Shape {

  private double radius;

  public Circle(double x, double y, double radius) {
    super(x,y);
    this.radius = radius;
  }

  public double area() {
    return Math.PI * radius * radius;
  }
}
```


as does Rectangle. This demonstrates **inheritance**.

```
public class Rectangle extends Shape {

  private double height;
  private double width;

  public Rectangle(double x, double y, double height, double width) {
    super(x,y);
    this.height = height;
    this.width = width;
  }

  public double area() {
    return width * height;
  }
}
```


We can now treat Rectangles and Circles as the more general type Shape:

Example:

```
public class ShapeTest {

  public static void main (String[] args) {
    Shape[] sA = new Shape[2];
    sA[0] = new Circle(1,1,1);
    sA[1] = new Rectangle(1,1,1,1);    // This is abstraction
    for(int i = 0; i < 2; i++)
      sA[i].moveTo(0,0);
    int totArea = 0;
    for(int i = 0; i < 2; i++)
       totArea += sA[i].area();        // This is polymorphism
  }
}
```

Note:

- We did not need to specify how a `Shape`'s area is calculated. This means that we are never able to construct *just* a `Shape`.
- We chose to encapsulate a `Point`, rather than inherit from it (*a shape is not a point*). Inheritance should be used sparingly. Always consider composition first.
- Once `Circles` and `Rectangles` can be treated as shapes we can have an array that contains both.
- The method `area()` was different for both shapes, but we did not need to cast. The Java virtual machine will determine which method is called.
- A good example of inheritance in the API is `java.awt` (e.g., a `Window` is a `Container` which is a `Component` which is an `Object`). However, in general inheritance hierarchies should be fairly shallow.

5. Interfaces and Abstract Classes

An interface:

- looks much like a class, but uses the keyword `interface`
- contains a list of method headers — name, list of parameters, return type (and exceptions)
- no method contents (they are called *abstract*, but abstract classes may have some methods specified)
- can only have constant variables declared
- no `public`/`private` necessary — they are implicitly `public`
- can implement multiple interfaces

Effectively, interfaces present all the OO advantages above, *except* inheritance.

Example:

```
public interface Matrix {

  public void set (int i, int j, int value);

  public int get (int i, int j);

  public void transpose ();
}
```


Classes can *implement* an interface:

Implementation 1:

```
public class MatrixReloaded implements Matrix {
  private int[][] matrixArray;
  public void transpose () {
    // do it one way
  }
  ...
```

Implementation 2:

```
public class MatrixRevolutions implements Matrix {
  private int[][] somethingDifferent;
  public void transpose () {
    // do it yet another way
  }
```

Why use interfaces?

1. Can be used like a superclass:

   Example:

```
Matrix[] myMatrixHolder = new Matrix[10];
myMatrixHolder[0] = new MatrixReloaded(2,2);
myMatrixHolder[1] = new MatrixRevolutions(20,20);
...
myMatrixHolder[0] = myMatrixHolder[1];
```


2. Specifies the methods that any implementation *must implement*.

   Example:

```
Matrix[] myMatrixHolder = new Matrix[10];
myMatrixHolder[0] = new MatrixReloaded(2,2);
myMatrixHolder[1] = new MatrixRevolutions(20,20);
...
for (int i=0; i<10; i++)
  myMatrixHolder[i].transpose();
```

   Note: this doesn't mean the methods are implemented *correctly*.

This is an important software engineering facility

- follows on from Information Hiding in Topic 1
  — allows independent development and maintenance of libraries and programs that use them
- will be used extensively in this unit to specify ADTs
- also used to add common functionality to all objects, eg *Serializable*, *Cloneable*

More examples — see the Java API

eg. the *Collection* interface, also *Runnable*, *Throwable*, *Iterable*, and *List*.

6. Exceptions

- special built-in classes
- used by Java to determine what to do when something goes wrong
- *thrown* by the Java virtual machine (JVM)

Example program

```
int[] myArray = {0,1,2,3};
System.out.println("The last number is:");
System.out.println(myArray[4]);
```

Output

```
The last number is:
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 4
at Test.main(Test.java:31)

Process Test exited abnormally with code 1
```

See the Java API for ArrayIndexOutOfBoundsException.

We can throw exceptions ourselves.

```
if (<condition>)
  throw new <exception type> (<message string>);
```

Example:

```
double squareRoot (double x) {
  if (x < 0)
    throw new ArithmeticException("Can't find square root
                                   of -ve number.");
  else {
    // calculate and return result
  }
}
```

Have a look for `ArithmeticException` in the Java API.

Two types of exceptions:

checked — most Java exceptions
— must be *caught* by the method, or passed (thrown) to the calling method

unchecked — `RuntimeException` and its subclasses
— don't need to be handled by programmer (JVM will halt)

To catch an exception, we use the code:

```
try {
  codeThatThrowsException();
}
catch(Exception e) {
  codeThatDealsWithException(e);
}
```

For simplicity, we will primarily use unchecked exceptions in this unit.

Compiling and running Java

There are various ways to compile and run Java, but the command line is the most ubiquitous. The command:

```
> javac myClass.java
```

will create the file `myClass.class` in the current directory. The command:

```
> java myClass
```

will execute the main method of the class `myClass`.

CITS2200 Data Structures and Algorithms

Topic 3

Recursive Data Structures and Linked Lists

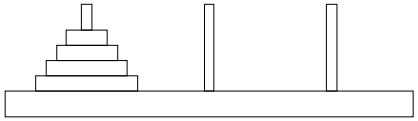
- Review of recursion: mathematical functions
- Recursive data structures: lists
- Implementing linked lists in Java
- Java and pointers
- Trees

Reading: Lambert and Osborne, Sections 10.1 and 5.3–5.4

1. Recursion

Powerful technique for solving problems which can be expressed in terms of smaller problems of the same kind.

eg. *Towers of Hanoi*



Aim: move all disks to the middle peg, moving one disk at a time, without ever putting a larger disk on a smaller one.

Exercise: Provide a recursive strategy for solving the Towers of Hanoi for arbitrary numbers of disks.

The Towers of Hanoi is also a good example of computational explosion.

It is alleged that the priests of Hanoi attempted to solve this puzzle with 64 disks. Even if they were able to move one hundred disks every second, this would have taken them more than 5,000,000,000 years!

1.1 Example: Common Mathematical Functions

Start with just increment and decrement...

```
// Class for doing recursive maths. Assumes all integers
// are non-negative (for simplicity no checks are made).

public class RMaths {

  // method to increment an integer
  public static int increment(int i) {return i + 1;}

  // method to decrement an integer
  public static int decrement(int i) {return i - 1;}

  // more methods to come here...
```

Note: All methods are:

- `public` — any program can access (use) the methods
- `static` — methods belong to the class (*class methods*), rather than objects (instances) of that class

In fact, we are not using objects here at all.

`increment` and `decrement` take `int` arguments and return `int`s.

They are "called" by commands of the form `RMaths.increment(4)`

— that is, the method `increment` belonging to the class `RMaths`.

```
public class RMathsTest {

  // simple method for testing RMaths
  public static void main(String[] args) {
    System.out.println(RMaths.increment(4));
  }
}
```


Addition: express what it means to add something to y in terms of adding something to $y-1$ (the decrement of y)

$$x + y = (x + 1) + (y - 1)$$

```
/*
 * add two integers
 */
public static int add(int x, int y) {
  if (y == 0) return x;
  else return add(increment(x), decrement(y));
}
```


Multiplication

$$x \times y = x + (x \times (y - 1))$$

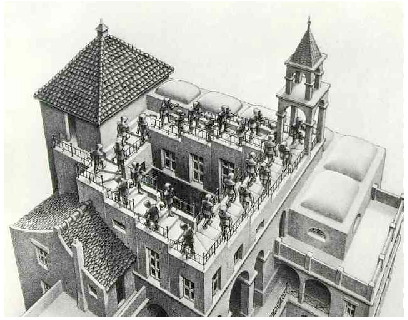
```
/*
 * multiply two integers
 */
public static int multiply(int x, int y) {
  if (y == 0) return 0;
  else return add(x, multiply(x, decrement(y)));
}
```

Similar code can be written for other functions such as `power` and `factorial`
⇒ see Exercises

Recursive programs require:

- one or more *base cases* or *terminating conditions*
- one or more *recursive cases* or *steps* — routine "calls itself"

Q: What if there is no base case?



Recursion is:

- powerful — can solve arbitrarily large problems
- concise — code doesn't increase in size with problem
- closely linked to the very important proof technique called *mathematical induction*.
- not necessarily efficient
  - we'll see later that the time taken by this implementation of multiplication increases with approximately the square of the second argument
  - long multiplication taught in school is approximately linear in the number of digits in the second argument

2. Recursive Data Structures

Recursive programs usually operate on *recursive data structures*

$\Rightarrow$ data structure *defined in terms of itself*

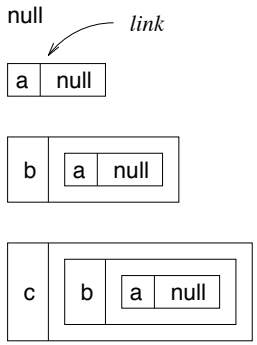
2.1 Lists

A *list* is defined recursively as follows:

- an empty list (or *null list*) is a list
- an item followed by (or *linked to*) a list is a list

Notice that the definition is like a recursive program — it has a base case and a recursive case!

Building a list...



2.2 List ADT

As an *abstract data type*, a list should allow us to:

1. Construct an empty list
2. Insert an element into the list
3. Look at an element in the list
4. Delete an element from the list
5. Move up and down the list

We will specify the List ADT more formally later . . .

For now, we will just look at a simple list that allows us to insert, delete, and examine only at the front of the list.

3. A LinkedList Class in Java

3.1 The Links

Defined recursively. . .

```
// link class for chars
class LinkChar {

  char item;             //  the item stored in this link
  LinkChar successor;    //  the link stored in this link

  LinkChar (char c, LinkChar s) {item = c; successor = s;}
}
```

Notice that the constructor makes a new link from an item and an existing link.

3.2 The Linked List

Next we need an object to "hold" the links. We will call this `LinkedListChar`.

Contains a variable which is either equal to "`null`" or to the first link (which in turn contains any other links), so it must be of type `LinkChar`. . .

```
class LinkedListChar {
  LinkChar first;
}
```

Now the methods. . .

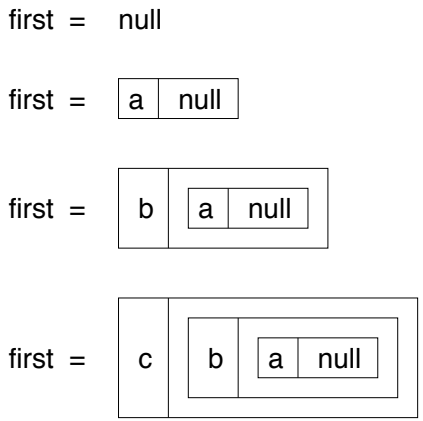
• **Constructing an empty list**

```
class LinkedListChar {

  LinkChar first;

  LinkedListChar () {first = null;}     // constructor
}
```

Conceptually, think of this as assigning a "null object" (a null list) to `first`. (Technically it makes `first` a null-reference, but don't worry about this subtlety for now.)

- **Adding to the list**

```
class LinkedListChar {
  LinkChar first;
  LinkedListChar () {first = null;}

  // insert a char at the front of the list
  void insert (char c) {first = new LinkChar(c, first);}
}
```

To create the list shown above, the class that *uses* `LinkedListChar`, say `LinkedListCharTest`, would include something like...

```
LinkedListChar myList;          // myList is an object
                                // of type LinkedListChar
myList = new LinkedListChar();  // call constructor to
                                // create empty list
myList.insert('a');
myList.insert('b');
myList.insert('c');
```


- **Examining the first item in the list**

```
// define a test for the empty list
boolean isEmpty () {return first == null;}

// if not empty return the first item
char examine () {if (!isEmpty()) return first.item;}
```

• Deleting the first item in the list

```
void delete () {if (!isEmpty()) first = first.successor;}
```

`first` then refers to the "*tail*" of the list.

Note that we no longer have a reference to the previous first link in the list (and can never get it back). We haven't really "deleted" it so much as "abandoned" it. Java's automatic *garbage collection* reclaims the space that the first link used.

⇒ This is one of the advantages of Java — in C/C++ we have to reclaim that space with additional code.

The Complete Program

```
package DAT;                    // It's part of Cara's DAT package.

import Exceptions.*;            // Use a package of
                                // exceptions defined elsewhere.
/**
 * A basic recursive (linked) list of chars.
 * @author Cara MacNish        // Lines between /** and */ generate
 */                             // automatic documentation.

public class LinkedListChar {
  /**
   * Reference to the first link in the list, or null if
   * the list is empty.
   */
  private LinkChar first;       // Private - users cannot access
                                // this directly.
```



```
  /**
   * Create an empty list.
   */
  public LinkedListChar () {first = null;}   // The constructor.

  /**
   * Test whether the list is empty.
   * @return true if the list is empty, false otherwise
   */
  public boolean isEmpty () {return first == null;}

  /**
   * Insert an item at the front of the list.
   * @param c the character to insert
   */
  public void insert (char c) {first = new LinkChar(c, first);}
```



```
  /**
   * Examine the first item in the list.
   * @return the first item in the list
   * @exception Underflow if the list is empty
   */
  public char examine () throws Underflow {
    if (!isEmpty()) return first.item;
    else throw new Underflow("examining empty list");
  }
          // Underflow is an example of an exception that
          // occurs (or is ``thrown'') if the list is empty.
  /**
   * Delete the first item in the list.
   * @exception Underflow if the list is empty
   */
  public void delete () throws Underflow {
    if (!isEmpty())  first = first.successor;
    else throw new Underflow("deleting from empty list");
  }
```

```
            // Many classes provide a string representation
            // of the data, for example for printing,
            // defined by a method called ``toString()''.
  /**
   * construct a string representation of the list
   * @return the string representation
   */
  public String toString () {
    LinkChar cursor = first;
    String s = "";
    while (cursor != null) {
      s = s + cursor.item;
      cursor = cursor.successor;
    }
    return s;
  }
}
```

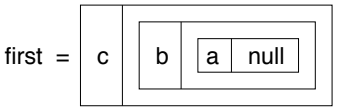

4. Java and Pointers

Conceptually, the successor of a list *is* a list.

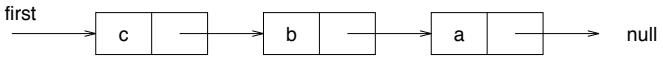
One of the great things about Java (and other suitable object oriented languages) is that the program closely reflects this "theoretical" concept — from a programmer's point-of-view the successor of a `LinkChar` *is* a `LinkChar`.

Internally, however, all instance variables act as *references*, or "*pointers*", to the actual data.

Therefore, a list that looks conceptually like



internally looks more like



For simplicity of drawing, we will often use the latter type of diagram for representing recursive data structures.

5. Trees

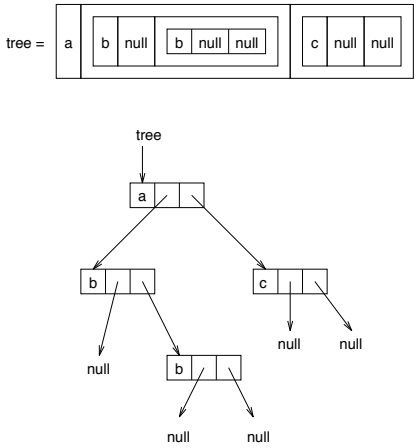
A tree is another example of a recursive data structure. It might be defined as follows:

- a *null* tree (or *empty tree*) is a tree
- an item followed by one or more trees is a tree

Some examples of trees:

- family trees
- XML files
- inheritance hierarchies

Graphical representations...



More on trees later.

6. Summary

Recursive data structures:

- can be arbitrarily large
- support recursive programs
- are a fundamental part of computer science — they will appear again and again in this and other units

$\Rightarrow$ You need to understand them. If not, seek help!

We will see many in this unit, including more on lists and trees.

CITS2200 Data Structures and Algorithms

Topic 4

Data Abstraction and Specification of ADTs

- Example — The "Reversal Problem" and a non-ADT solution
- Data abstraction
- Specifying ADTs
- Interfaces
- `javadoc` documentation
- An ADT solution to the Reversal Problem

Reading: Lambert and Osborne, Chapter 7.

1. Aims

The aims of this topic are to:

1. provide a more detailed example of data type abstraction
2. introduce two example data types: the Queue and Stack
3. show how data types will be specified in this unit

2. The Reversal Problem and a non-ADT Solution

As a more detailed example of ADTs we consider the reversal problem:

Given two character sequences A and B, is A the reverse of B?

One solution: store in arrays, scan and compare from either end $\cdots \triangleright$


```
import java.io.*;

/*
 * Reversal program (not using ADTs).
 * Accepts two character strings from the terminal, separated by
 * whitespace, and determines whether one is the reverse of the
 * other.
 */
public class Reversal {

  // constant for maximum length of the input sequences
  public final static int MAX_SEQUENCE = 100;

  // main program
  public static void main(String[] args) throws IOException {
```

```
    // arrays for storing input sequences
    char[] sequence1 = new char[MAX_SEQUENCE];
    char[] sequence2 = new char[MAX_SEQUENCE];

    // indices for first and second sequences
    int index1 = 0;
    int index2 = 0;

    // other local variables
    boolean isReverse = true;
    char c;
```



```
    // Read in the first sequence and store
    c = (char) System.in.read();
    while (c != ' ') {
      sequence1[index1] = c;
      index1++;
      c = (char) System.in.read();
    }

    // Clear white space.
    while (c == ' ')  c = (char) System.in.read();

    // Read in the second sequence and store
    while (c != ' ' && c != '\n' && c != '\r') {
      sequence2[index2] = c;
      index2++;
      c = (char) System.in.read();
    }
```



```
    // Compare the two sequences.
    isReverse = index1 == index2;
    index1 = 0;
    index2--;

    while (isReverse && index1 <= index2) {
      isReverse = isReverse &&
        sequence1[index1] == sequence2[index2-index1];
      index1++;
    }

    if (isReverse) System.out.println("Yes that is the reverse.");
    else  System.out.println("No that's not the reverse.");
  }
}
```


Notice that this program mixes

- "low-level" details of data storage (in arrays) and manipulation (using indices), with
- the "high-level" goals of inputting and comparing sequences.

$\Rightarrow$ difficult to modify, maintain, reuse, etc

Better solution — use ADTs!

3. Data Abstraction

The above program integrates:

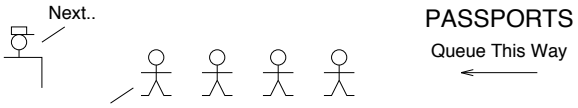
- data, and instructions to access it
- "higher-level" role of the program

We wish to take a more abstract view. . . can we use generic, reusable data structures?

When dealing with the first sequence we. . .

- "Create" an empty sequence
- Append characters to the end
- Scan from beginning to end
- Don't reuse scanned characters

But this is just what a *queue*, or *FIFO* (first-in, first-out buffer), does!



In general, the operations on a queue include:

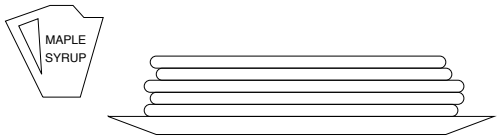
1. Create an empty queue
2. Test whether the queue is empty
3. Add a new latest element
4. Examine the earliest element
5. Delete the earliest element

From a user point-of-view, we *don't care how its implemented* — all we need in order to write our reversal program is what operations are available to us.

(Implementations will be considered later.)

Operations needed for the second sequence are the same as the first, except the elements added *last* are taken off first.

This is the operation of a *stack*, or *LIFO* (last-in first-out buffer).



Operations on a stack:

1. Create an empty stack
2. Test whether the stack is empty
3. Add (*push*) a new element on the top
4. Examine (*peek* at) the top element
5. Delete (*pop*) the top element

Implementation of a stack — see Lab Exercises!

4. Specifying ADTs

We saw in Topic 1 that ADTs consist of a set of operations on a set of data values. We can *specify* ADTs by listing the operations (or *methods*).

The lists of operations on the previous pages are very informal and not sufficient for writing code. For example

2. Test whether the queue is empty

doesn't tell us the name of the method, what arguments it is called with, what is returned, and whether it can throw an exception.

In these notes, we will specify ADTs by providing at least:

- the *name* of each operation
- example *parameters* (the implementation may use different parameter names, but will have the same number, type and order)
- an explanation of *what the operation does* — in particular, any constraints on or changes to the parameters, changes to the ADT instance on which the method operates, what is returned, and any exceptions thrown

Thus, a Queue ADT might be specified by the following operations:

1. *Queue()*: create an empty queue
2. *isEmpty()*: return `true` if the queue is empty, `false` otherwise
3. *enqueue(e)*: e is added as the last item in the queue
4. *examine()*: return the first item in the queue, or throw an exception if the queue is empty
5. *dequeue()*: remove and return the first item in the queue, or throw an exception if the queue is empty

Similarly, the specification of a Stack ADT:

1. *Stack()*: create an empty stack
2. *isEmpty()*: return `true` if the stack is empty, `false` otherwise
3. *push(e)*: item e is pushed onto the top of the stack
4. *peek()*: return the item on the top of the stack, or throw an exception if the stack is empty
5. *pop()*: remove and return the item on the top of the stack, or throw an exception if the stack is empty

Note: The use of upper and lowercase in method names should follow the rules described in the document *Java Programming Conventions*.

5. Interfaces

As we have seen, Java itself provides a rigorous way of specifying the methods in classes: *interfaces*.

Interfaces provide a natural way of specifying ADTs in programs and enforcing those specifications.

Example · · · ▷


```
// Interface for a Queue of characters.
public interface QueueChar {

  /*
   * test whether the queue is empty
   * return true if the queue is empty, false otherwise
   */
  public boolean isEmpty ();

  /*
   * insert an item at the back of the queue
   */
  public void enqueue (char a);
```



```
  /*
   * examine and return the item at the front of the queue
   * throw an Underflow exception if the queue is empty
   */
  public char examine () throws Underflow;

  /*
   * remove the item at the front of the queue
   * return the removed item
   * throw an Underflow if the queue is empty
   */
  public char dequeue () throws Underflow;
}
```

Note: This interface specifies a queue of characters (`chars`). This can be seen in the argument to `enqueue` and the return types of `examine` and `dequeue`.

6. javadoc Documentation

Many texts will describe ADT operations in terms of *preconditions* and *postconditions*.

preconditions — constraints on variable values for the operations to work correctly

post-conditions — what the operation does, in particular changes to the input variables

In this unit we will replace these, as far as possible, with the facilities provided by the documentation program `javadoc`.

The documentation for each method should include:

- a short general description of the method
- a `@param` statement describing each parameter
- a `@return` statement describing the value/object returned (except where the return type is void)
- an `@exception` statement describing each exception thrown

The `javadoc` program automatically generates HTML on-line documentation from these comments.

Notes:

- Full `javadoc` documentation must be included with all code that you submit in this unit.
- We will sometimes omit documentation (or break formatting rules) in lectures to fit programs on slides.

Example

```
/**
 * remove the item at the front of the queue
 * @return the removed item
 * @exception Underflow if the queue is empty
 */
public char dequeue () throws Underflow;
```

Here the "precondition" is that the queue must be non-empty, the "postcondition" is that the front element is deleted.

The final `QueueChar` interface ···▷


```
package DAT;           // make this interface part of a package
                       // (or library) called DAT

import Exceptions.*;   // use a package of exceptions called
                       // Exceptions (contains Underflow)
/**
 * Interface for Queue of characters.
 * @author Cara MacNish              // some other javadoc fields
 */
public interface QueueChar {

  /**
   * test whether the queue is empty
   * @return true if the queue is empty, false otherwise
   */
  public boolean isEmpty ();
```

```
  /**
   * insert an item at the back of the queue
   * @param a the item to insert
   */
  public void enqueue (char a);

  /**
   * examine the item at the front of the queue
   * @return the first item
   * @exception Underflow if the queue is empty
   */
  public char examine () throws Underflow;

  /**
   * remove the item at the front of the queue
   * @return the removed item
   * @exception Underflow if the queue is empty
   */
  public char dequeue () throws Underflow;}
```


7. An ADT Solution to the Reversal Problem

Given specifications for Queue and Stack ADTs, which we assume for the moment are implementations of interfaces `QueueChar` and `StackChar` called `QueueCharImplementation` and `StackCharImplementation` respectively, the Reversal program can be rewritten at a more abstract level.

Program · · · ▷


```
package DAT;
import java.io.*;
import Exceptions.*;

/**
 * Reversal program using ADTs.
 * Accepts two character strings from the terminal, separated by
 * whitespace and determines whether one is the reverse of the other
 * @author Cara MacNish
 */
public class ReversalADT {

  /**
   * main program
   * @param args command line arguments
   * @exception Exception passed to interpreter
   */
  public static void main(String[] args) throws Exception {
```



```
    // queue for storing first input sequence
    QueueChar q = new QueueCharImplementation();

    // stack for storing second input sequence
    StackChar s = new StackCharImplementation();

    // other local variables
    boolean isReverse = true;
    char c;

    // Read in the first sequence and store characters in a queue.
    c = (char) System.in.read();
    while (c != ' ' && c != '\n' && c != '\r') {
      q.enqueue(c);
      c = (char) System.in.read();
    }

    // Clear white space.
    while (c == ' ')   c = (char) System.in.read();
```

```
    // Read in the second sequence and store characters in a stack.
    while (c != ' ' && c != '\n' && c != '\r') {
      s.push(c);
      c = (char) System.in.read();
    }

    // Compare the two sequences.
    while (isReverse && !q.isEmpty() && !s.isEmpty())
      isReverse = isReverse && q.dequeue() == s.pop();

    if (isReverse && q.isEmpty() && s.isEmpty())
      System.out.println("Yes that is the reverse.");
    else System.out.println("No that's not the reverse.");
  }
}
```


Advantages over previous version

- Program 'reads' better
  - more 'declarative'
  - easier to follow and debug
- Modular
  - Implementation independent — easier to change/upgrade
  - Division of work-load

8. Summary

- When programming we should look for *abstractions* of the data — could we use a generic data structure (ADT) rather than "re-implement the wheel"?
- ADTs can be specified by listing operations and explaining how the object and arguments are affected
- More rigorous specifications can be enforced in Java using interfaces
- ADT operations (methods) should be described within the implementation using `javadoc` comments

Next we will look at implementations for the Queue...

CITS2200 Data Structures and Algorithms

Topic 5

Queues

- Implementations of the *Queue* ADT
- Queue specification
- Queue interface
- Block (array) representations of queues
- Recursive (linked) representations of queues

Reading: Lambert and Osborne, Sections 8.1–8.4.

1. Educational Aims

The aims of this topic are to:

1. Introduce two main ways of implementing collection classes:
   - block (array-based) implementations, and
   - linked (recursive) implementations
2. Introduce pros and cons of the two structures.
3. Develop basic skills in manipulating these two kinds of structures.

Note the ADT just specifies the operations available, and the results of applying those operations. There are many different ways to implement any given ADT.

2. Specification

Recall that in a *queue*, or *FIFO*, elements are added to one end, and read/deleted from the other, in chronological order.

1. *Queue()*: create an empty queue
2. *isEmpty()*: return *true* if the queue is empty, *false* otherwise
3. *enqueue(e)*: e is added as the last item in the queue
4. *examine()*: return the first item, error if the queue is empty
5. *dequeue()*: remove and return first item, error if queue empty

For simplicity, we will begin with queues containing only `char`s.

2.1 Classification of ADT Operations:

constructors are used to create data structure instances
eg. *Queue*

checkers report on the "state" of the data structure
eg. *isEmpty*

manipulators examine and modify data structures
eg. *enqueue, examine, dequeue*

3. Interface

```
import Exceptions.*;
                                // Character queue interface.
public interface QueueChar {    // some javadoc comments omitted

  /**
   * test whether the queue is empty
   * @return true if the queue is empty, false otherwise
   */
  public boolean isEmpty ();

  /**
   * add a new item to the queue
   * @param a the item to add
   */
  public void enqueue (char a);
```

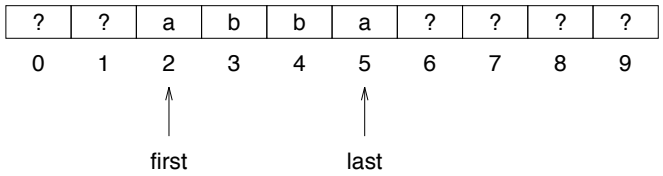


```
  /**
   * examine the first item in the queue
   * @return the first item
   * @exception Underflow if the queue is empty
   */
  public char examine () throws Underflow;

  /**
   * remove the first item in the queue
   * @return the first item
   * @exception Underflow if the queue is empty
   */
  public char dequeue() throws Underflow;
```


4. Block Representations

Simplest representation:

- sequence of elements stored in array
- indices (counters) indicating first and last element



Disadvantage: queue will be bounded! — can only implement a variation on the spec:

3. *enqueue(e)*: e is added as the last item in the queue, *or error if the queue is full*

For convenience, we will include another checker:

6. *isFull()*: return *true* if the queue is full, *false* otherwise

4.1 Class Declaration

```
import Exceptions.*;

/**
 * Block representation of a character queue.
 * The queue is bounded.
 */
public class QueueCharBlock implements QueueChar {
```

Notice implementing interface — class will only compile without error if it provides all methods specified in the interface. (Otherwise you can declare the class as *abstract*).


```
/**
 * an array of queue items
 */
private char[] items;

/**
 * index for the first item
 */
private int first;

/**
 * index for the last item
 */
private int last;
```


4.2 Modifiers

enqueue, *examine* and *dequeue* are straightforward...

```
/**
 * add a new item to the queue
 * @param a the item to add
 * @exception Overflow if queue is full
 */
public void enqueue (char a) throws Overflow {
  if (!isFull()) {
    last++;
    items[last] = a;
  }
  else throw new Overflow("enqueuing to full queue");
}
```

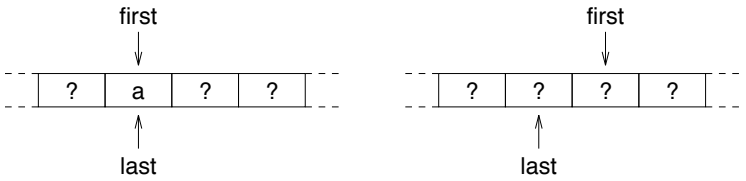


```
/**
 * examine the first item in the queue
 * @return the first item
 * @exception Underflow if the queue is empty
 */
public char examine () throws Underflow {
  if (!isEmpty())  return items[first];
  else throw new Underflow("examining empty queue");
}
```

```
/**
 * remove the first item in the queue
 * @return the first item
 * @exception Underflow if the queue is empty
 */
public char dequeue() throws Underflow {
  if (!isEmpty()) {
    char a = items[first];
    first++;
    return a;
  }
  else throw new Underflow("dequeuing from empty queue");
}
```


4.3 Constructors and Checkers

To see how to code the constructor and *isEmpty* consider successive deletions until `first` catches `last`.



The queue has one element if `first == last`, and is therefore empty when `first == last + 1` ...


```
/**
 * test whether the queue is empty
 * @return true if the queue is empty, false otherwise
 */

public boolean isEmpty () {return first == last + 1;}
```

Java arrays number from 0, so `first` is initialised to 0...

```
/**
 * initialise a new queue
 * @param size the size of the queue
 */
public QueueCharBlock (int size) {
  items = new char[size];
  first = 0;
  last = -1;
}
```


The queue is full if there is simply no room left in the array...

```
/**
 * test whether the queue is full
 * @return true if the queue is full, false otherwise
 */
public boolean isFull () {return last == items.length - 1;}
```

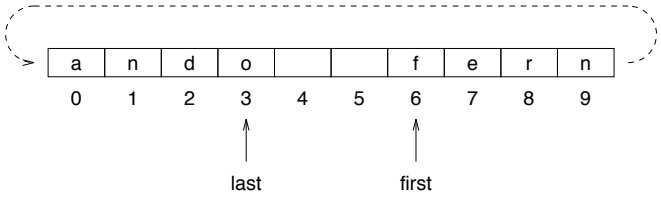
Notes

- `length` is an instance variable of an array object, and contains the size of the array.
- Since arrays number from 0, the n^{th} element has index $n - 1$.

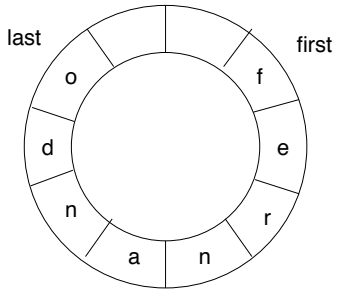
4.4 Alternative Block Implementations

Problem: as elements are deleted the amount of room left for the queue is eroded — the space in the array is not reused.

Solution: wrap queue around...



Conceptually, this forms a *cyclic queue* (or *cyclic buffer*)...



Effects on the above program...

- `first` and `last` must be incremented until they reach the end of the array, then reduced to 0. This can be achieved in a concise way using the % ("mod") operation. eg:

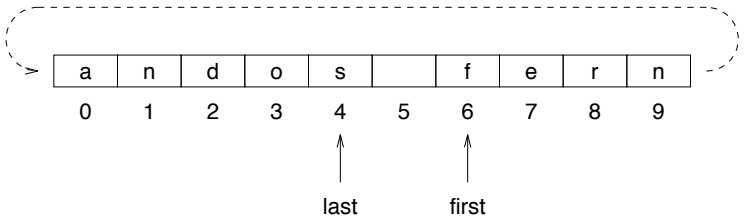
```
public void enqueue (char a) {
  if (!isFull()) {
    last = (last + 1) % items.length;
    items[last] = a;
  }
  else throw new Overflow("enqueuing to full queue");
}
```


- A queue is now empty when:

```
first == (last + 1) % items.length
```

Problem: The above condition also represents a full queue!
One solution — define queue as full when it contains `items.length-1` elements and use the condition:

```
first == (last + 2) % items.length
```



But now a queue created to hold n objects only has room for $n-1$ objects

$\Rightarrow$ modify the constructor...

```
public QueueCharCyclic (int size) {
  items = new char[size+1];    // add 1 to array size
  first = 0;
  last = size;                 // start last at end of block
}
```

Another solution — instead of two indices, keep one index for the first element, and a count of the size of the queue.

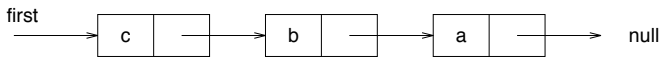
$\Rightarrow$ Exercises!

5. Recursive (Linked) Representation

Biggest problem with block representation — predefined queue length

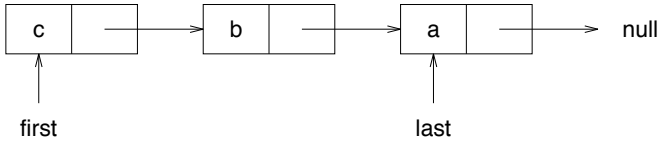
Solution: use a recursive structure!

Recall singly linked list...



For a queue, we need to be able to access both ends — one to insert and one to delete.

Although the end can be accessed by following the references down the list, it is more efficient to store references to both ends...



Note, it is important that the arrows point from first to last.

5.1 Class Declaration

```
import Exceptions.*;
/**
 * Linked list representation of a queue of characters.
 */
public class QueueCharLinked implements QueueChar {

  /**
   * the front of the queue, or null if queue's empty
   */
  private LinkChar first;

  /**
   * the back of the queue, or null if queue's empty
   */
  private LinkChar last;
}
```

5.2 Constructors and Checkers

Empty queue:

first → null
last → null

Queue and *isEmpty* are easy...


```
/**
 * initialise a new Queue
 */
public QueueCharLinked () {
  first = null;
  last = null;
}

/**
 * test whether the queue is empty
 * @return true if the queue is empty, false otherwise
 */
public boolean isEmpty () {return first == null;}
```

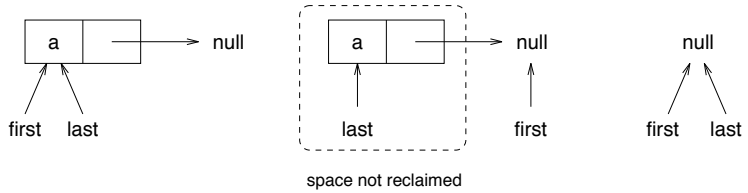

5.3 Examining and Dequeueing

Examining and dequeueing are easy!

Examining is the same as for the linked list...

```
public char examine () throws Underflow {
  if (!isEmpty())  return first.item;
  else throw new Underflow("examining empty queue");
}
```

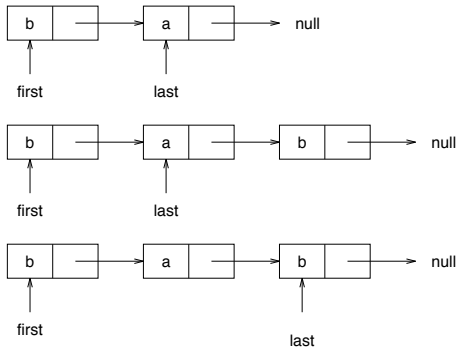

Dequeueing is the same as deleting in the linked list, except that when the last item is dequeued, `last` must be assigned null...



```
public char dequeue () throws Underflow {
  if (!isEmpty()) {
    char c = first.item;
    first = first.successor;
    if (isEmpty())  last = null;
    return c;
  }
  else throw new Underflow("dequeuing from empty queue");
}
```


5.4 Enqueueing

Enqueueing is also easy! Just reassign the null reference at the end of the queue to a reference to another link, and move `last` to the new last element...



...unless the queue is empty, then `first` and `last` must both reference a new link...

```
public void enqueue (char a) {
  if (isEmpty()) {
    first = new LinkChar(a,null);
    last = first;
  }
  else {
    last.successor = new LinkChar(a,null);
    last = last.successor;
  }
}
```


6. Summary

- block (array with indices to endpoints)
  - bounded
  - may reserve space unnecessarily
  - 'eroded' with use
- block with wrap around (cyclic)
  - bounded
  - space reserved unnecessarily
  - not 'eroded'
- recursive (linked list with references to endpoints)
  - unbounded
  - no unnecessary space wasted
  - no 'erosion' of space — garbage collection

CITS2200 Data Structures and Algorithms

Topic 6

Performance Analysis 1: Introduction

- Why analyse performance?
- Types of performance measurement
  - empirical
  - analytical
- An example of analytical analysis using Queue
- Introduction to growth rates

Reading: Lambert and Osborne, Section 4.1.

1. Educational Aims

The aims of this topic are to:

1. begin thinking about the *implications of the choices you make* for ADT performance
2. introduce simple metrics for assessing algorithm performance, which will later lead to mathematically-based techniques

2. Why Performance Analysis?

- Comparison
  - choice of ADT
  - choice of implementation
  - trade-offs — may be no clear winner/depend on calling program
- Improvement
  - identification of expensive operations, bottlenecks
  - improved implementations within ADTs
  - improved implementation of calling programs

3. Types of Performance Measurement

Empirical measurement

We will see that the most efficient queue ADT to use depends on the program that uses it — which operations are used most often.

If we have access to the program(s), we may be able to measure the performance in those programs, on real data — called *evaluation in context*.

This is the "get yer hands dirty" approach. Run the system with real-world input and observe, or monitor (automatically), the results.

Can compare data structures on the same problems (same machine, same compiler, etc)

$\Rightarrow$ *benchmark* programs

- Useful if test input is close to expected input.
- Not much use if we are developing eg a library of modules for use in many different contexts

In some cases, it is not feasible to test a programme "in the field" (e.g. nuclear weapons systems). Here, we may construct a (computer) model of the system and evaluate performance with simulated data.

A computer program normally acts as its own model — run on simulated data (often generated using pseudo-random numbers).

However, a simplified model may be built or the program modified to fit the simulated data.

Advantages

- nondestructive
- cheap
- fast
- reproducible

Disadvantages

- only as good as the simulations
- can never be sure it matches reality

Analytical Measurement

Construct a mathematical or theoretical model — use theoretical techniques to estimate system performance.

Usually

- coarse estimates
- growth rates, complexity classes rather than 'actual' time
- *worst case* or *average case*

But. . . !

- *fundamental view of behaviour* — less susceptible to
  - speed of hardware, number of other processes running, etc
  - choice of data sets
  - unrepresentative examples, spurious responses
- gives a better understanding of the problems
  - why is it slow?
  - could it be improved?

We will concentrate on analytical analyses.

4. Example: A Basic Analysis of the Queue ADTs

As an example of comparison of ADT performance we consider different representations of queues using a crude time estimate

Simplifying assumptions:

- each high-level operation (arithmetic operation, Boolean operation, subscripting, assignment) takes 1 time unit
- conditional statement takes 1 time unit + time to evaluate Boolean expression + time taken by most time consuming alternative (*worst-case* assumption)
- field lookup ("dot" operation) takes 1 time unit
- method takes 1 (for the call) plus 1 for each argument (since each is an assignment)
- creating a new object (from a different class) takes T_c time units

4.1 Block Representation Queues (Without Wraparound)

```
public QueueCharBlock (int size) {                //2
  items = new char[size];                         //1+Tc
  first = 0;                                      //1
  last = -1;                                      //1
}
```

5 + T_c time units

```
public boolean isEmpty () {return first == last + 1;}
```

4 time units

```
public boolean isFull () {return last == items.length - 1;}
```

5 time units


```
public void enqueue (char a) throws Overflow {      //2
  if (!isFull()) {                                  //7
    last++;                                         //1
    items[last] = a;                                //2
  }
  else throw new Overflow("enqueuing to full queue");
}
```

12 time units

Exercise:

How many time units for each of the following...

```
public char examine () throws Underflow {
  if (!isEmpty())  return items[first];
  else throw new Underflow("examining empty queue");
}

public char dequeue() throws Underflow {
  if (!isEmpty()) {
    char a = items[first];
    first++;
    return a;
  }
  else throw new Underflow("dequeuing from empty queue");
}
```

Summary for Block Implementation

`isEmpty`, `enqueue`, `examine` and `dequeue` are *constant time* operations

`Queue` is constant time *if* T_c is constant time

4.2 Recursive (Linked) Representation Queues

```
public QueueCharLinked () {
  first = null;
  last = null;
}
```

3 time units

```
public boolean isEmpty () {return first == null;}
```

3 time units


```
public void enqueue (char a) {                  //2
  if (isEmpty()) {                              //4
    first = new LinkChar(a,null);               //1+Tc
    last = first;                               //1
  }
  else {
    last.successor = new LinkChar(a,null);      //2+Tc
    last = last.successor;                      //2
  }
}
```

$10 + T_c$ time units

```
public char examine () throws Underflow {
  if (!isEmpty())  return first.item;
  else throw new Underflow("examining empty queue");
}
```

8 time units


```
public char dequeue () throws Underflow {          //1
  if (!isEmpty()) {                                //5
    char c = first.item;                           //2
    first = first.successor;                       //2
    if (isEmpty())  last = null;                   //5
    return c;                                      //1
  }
  else throw new Underflow("dequeuing from empty queue");
}
```

16 time units

Summary for Linked Implementation

Again all are constant time, assuming T_c is.

Comparison...

	block	recursive
Queue	$5 + T_c$	3
isEmpty	4	3
enqueue	12	$10 + T_c$
examine		8
dequeue		16

...shows no clear winner, especially given

- estimates are very rough — many assumptions
- dependent on relative usage of operations in the programs calling the ADT — eg. is *isEmpty* used more or less than *dequeue*

We will generally not be interested in these "small" differences (eg 5 time units vs 3 time units) — given the assumptions made these are not very informative.

Rather we will be interested in *classifying* operations according to *rates of growth*...

5. Growth Rates

For comparative purposes, exact numbers are pretty irrelevant! It is the *rate of growth* that is important.

We will abstract away from inessential detail...

- ignore specific values of input and just consider the number of items, or "size" of input
- ignore precise duration of operations and consider the number of (specific) operations as an abstract measure of time
- ignore actual storage space occupied by data elements and consider the number of items stored as an abstract measure of space

6. Summary

Two main types of performance measurement — empirical and analytical.

We will concentrate on analytical:

- fundamental view of behaviour
- abstracts away from machine, data sets, etc
- helps in understanding data structures and their implementations

Rather than attempting 'fine grained' analysis that compares small differences, we will concentrate on a coarser (but more robust) analysis in terms of *rates of growth*.

CITS2200 Data Structures and Algorithms

Topic 7

Performance Analysis 2: Asymptotic Analysis

- Choosing abstract performance measures
  - worst case, expected case, amortized case
- Asymptotic growth rates
  - Why use them? Comparison in the limit. "Big O"
- Analysis of recursive programs

Reading: Lambert and Osborne, Sections 4.2–4.3.

1. Educational Aims

The aims of this topic are:

1. to develop a mathematical competency in describing and understanding algorithm performance, and
2. to begin to develop an intuitive feel for these mathematical properties.

It is essential for a programmer to be able to understand the capabilities and limitations of different data structures. Asymptotic analysis provides the foundation for this understanding (even though you would not expect to do such analysis on a regular basis).

2. Worst Case, Expected Case, Amortized Case

Abstract measures of time and space will still depend on actual input data.

eg Exhaustive sequential search

```
public int eSearch(...) {
  ...
  i = 0;
  while (a[i] != goal && i < n)  i++;
  if (i == n)  return -1;              // goal not found
  else  return i;
}
```


Abstract time

- goal is first element in array — a units
- goal is last element in array — $a + bn$ units

for some constants a and b.

Different growth rates — second measure increases with n.

What measure do we use? A number of alternatives...

2.1 Worst Case Analysis

Choose data which have the largest time/space requirements.

In the case of `esearch`, the worst case complexity is $a + bn$

Advantages

- relatively simple
- gives an upper bound, or *guarantee*, of behaviour — when your client runs it it might perform better, but you can be sure it won't perform any worse

Disadvantages

- worst case could be unrepresentative — might be unduly pessimistic
  - knock on effect — client processes may perform below their capabilities
  - you might not get anyone to buy it!

Since we want behaviour guarantees, we will *usually consider worst case analysis* in this unit.

(Note there is also 'best case' analysis, as used by second-hand car sales persons and stock brokers.)

2.2 Expected Case Analysis

Ask what happens in the average, or "expected" case.

For `eSearch`, $a + \frac{b}{2}n$, assuming a uniform distribution over the input.

Advantages

- more 'realistic' indicator of what will happen in any given execution
- reduces effects of spurious/non-typical/outlier examples

For example, Tony Hoare's `Quicksort` algorithm is generally the fastest sorting algorithm in practice, despite it's worst case complexity being significantly higher than other algorithms.

Disadvantages

- only possible if we know (or can accurately guess) probability distribution over examples (with respect to size)
- more difficult to calculate
- often does not provide significantly more information than worst case when we look at growth rates
- may also be misleading...

2.3 Amortized Case Analysis

Amortized analysis is a variety of worst case analysis, but rather than looking at the cost of doing the operation once, it examines the cost of repeating the operation in a sequence.

That is, we determine the worst case complexity $T(n)$ of performing a sequence of n operations, and report the amortized complexity as $T(n)/n$.

An alternative view is the accounting method: determine the individual cost of each operation, including both its execution time and its influence on the running time of future operations. The analogy: imagine that when you perform fast operations you deposit some "time" into a savings account that you can use when you run a slower operation.

Reading: Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, Chapter 17.

2.4 Amortized Analysis for a Multi-delete Stack

A multi-delete stack is the stack ADT with an additional operation:

1. *mPop(i)*: delete the top i elements from the stack

Assuming a linked representation, the obvious way to execute `mPop(i)` is to perform pop i times.

If each *pop* takes *b* time units, *mPop(i)* will take approximately ib time units — linear in i!

Worst case is nb time units for stack of size n.

But...

Before you can delete i elements, need to (somewhere along the way...) individually insert i elements, which takes i operations and hence ic time for some constant c.

Total for those $i+1$ operations is $i(c+b)$. The time for i operations is approximately linear in i. The *average* time for each operation

$$\frac{i}{i+1}(c+b)$$

is approximately constant — independent of i.

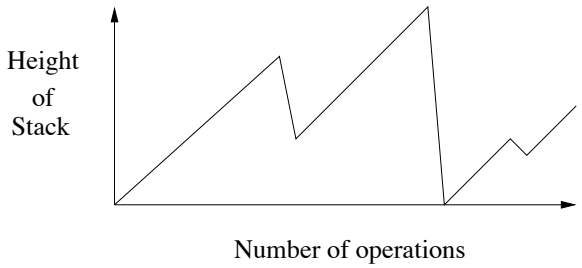
More accurate for larger i, which is also where its more important!

$$\left(\lim_{i \to \infty} \frac{i}{i+1}(c+b) = c+b \right)$$

This is called an *amortized analysis*. The cost of an expensive operation is amortized over the cheaper ones which *must* accompany it.

The Accounting Method for the Multi-delete Stack

Every time `push` is called we take a constant time (say a) to perform the operation, but we also put a constant amount of time (say b) in our "time-bank". When it comes time to perform multi-pop $mPop(i)$, if there are i items to delete, we must have at least ib time units in the bank.



Where Amortized Analysis Makes a Difference

In the block implementations of the data structures we have seen so far, we simply throw an exception when we try to add to a full structure.

Several implementations (e.g. `Java.util.ArrayList`) do not throw an exception in this case, but rather create an array *twice* the size, copy all the elements in the old array across to the new array, and then add the new element to the new array.

This is an expensive operation, but it can be shown that the *amortized* cost of the *add* operation is constant.

3. Asymptotic Growth Rates

We have talked about comparing data structure implementations — using either an empirical or analytical approach.

Focus on analytical:

- independent of run-time environment
- improves understanding of the data structures

We said we would be interested in comparisons in terms of *rates of growth*.

Theoretical analysis also permits a deeper comparison which the other methods don't — *comparison with the performance barrier inherent in problems. . .*

Wish to be able to make statements like:

> Searching for a given element in a block of n distinct elements using only equality testing takes n comparisons in the worst case.
>
> Searching for a given element in an ordered list takes at least $\log n$ comparisons in the worst case.

These are *lower bounds* (on the *worst case*) — they tell us that we are never going to do any better *no matter what algorithm we choose*.

Again they reflect growth rates (linear, logarithmic)

In this section, we formalise the ideas of analytical comparison and growth rates.

3.1 Why Asymptopia

We would like to have a *simple description* of behaviour for use in comparison.

- Evaluation may be misleading.
  Consider the functions $t_1 = 0.002m^2$, $t_2 = 0.2m$, $t_3 = 2\log m$.

  Evaluating at $m = 5$ gives $t_1 < t_2 < t_3$. This could be misleading — for "serious" values of m the picture is the opposite way around.
  Want a description of behaviour over the full range.

- Want a *closed form*.

  eg. $\dfrac{n(n+1)}{2}$ not $n + (n-1) + \cdots + 2 + 1$

  Some functions don't have closed forms, or they are difficult to find — want a closed form approximation

- Want simplicity.

  Difficult to see what $2^{n-\frac{1}{n}} \log n^2 + \dfrac{3}{2} n^{2-n}$ does. We want to abstract away from the smaller perturbations...

  What simple function does it behave *like*?

Solution

Investigate what simple function the more complex one *tends to* or *asymptotically approaches* as the argument approaches infinity, ie *in the limit*.

Choosing large arguments has the effect of making less important terms fade away compared with important ones.

eg. What if we want to approximate $n^4 + n^2$ by n^4 ?

How much error?

n	n^4	n^2	$\dfrac{n^2}{n^4+n^2}$
1	1	1	50%
2	16	4	20%
5	625	25	3.8%
10	10 000	100	1%
20	160 000	400	0.25%
50	6 250 000	2 500	0.04%

3.2 Comparison "in the Limit"

How well does one function approximate another?

Compare growth rates. Two basic comparisons...

1. $$\frac{f(n)}{g(n)} \to 0 \quad \text{as} \quad n \to \infty$$

   $\Rightarrow$ $f(n)$ grows more slowly than $g(n)$.

2.

$$\frac{f(n)}{g(n)} \to 1 \quad \text{as} \quad n \to \infty$$

$\Rightarrow$ $f(n)$ is *asymptotic* to $g(n)$.

In fact we won't even be this picky — we'll just be concerned whether the ratio approaches a constant $c > 0$.

$$\frac{f(n)}{g(n)} \to c \quad \text{as} \quad n \to \infty$$

This really highlights the distinction between different orders of growth — we don't care if the constant is 0.00000000001 !

3.3 'Big O' Notation

In order to talk about comparative growth rates more succinctly we use the *'big O'* notation...

Definition

$f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer $n_0 \geq 1$ such that, for all $n \geq n_0$,

$$f(n) \leq cg(n).$$

— f "grows" no faster than g, for sufficiently large n

— growth rate of f is *bounded from above* by g

Example:

Show (prove) that n^2 is $O(n^3)$.

Proof

We need to show that for some $c > 0$ and $n_0 \geq 1$,

$$n^2 \leq cn^3$$

for all $n \geq n_0$. This is equivalent to

$$1 \leq cn$$

for all $n \geq n_0$.

Choosing $c = n_0 = 1$ satisfies this inequality. □

Exercise:

Show that $5n$ is $O(3n)$.

Exercise:

Show that 143 is $O(1)$.

Exercise:

Show that for any constants a and b, an^3 is $O(bn^3)$.

Example:

Prove that n^3 is not $O(n^2)$.

Proof (by contradiction)

Assume that n^3 is $O(n^2)$. Then there exists *some* $c > 0$ and $n_0 \geq 1$ such that

$$n^3 \leq cn^2$$

for all $n \geq n_0$.

Now for *any* integer $m > 1$ we have $mn_0 > n_0$, and hence

$$(mn_0)^3 \leq c(mn_0)^2.$$

Re-arranging gives

$$\begin{aligned} m^3 n_0^3 &\leq cm^2 n_0^2 \\ mn_0 &\leq c \\ m &\leq \frac{c}{n_0} \end{aligned}$$

This is contradicted by any choice of m such that $m > \frac{c}{n_0}$. Thus the initial assumption is incorrect, and n^3 is *not* $O(n^2)$. □

From these examples we can start to see that big O analysis focuses on *dominating terms*.

For example a polynomial

$$a_d n^d + a_{d-1} n^{d-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

— $O(n^d)$

— is $O(n^m)$ for any $m > d$

— is not $O(n^l)$ for any $l < d$.

Here $a_d n^d$ is the dominating term, with *degree* d.

For non-polynomials identifying dominating terms may be more difficult.

Most common in CS

- polynomials — $1, n, n^2, n^3, \ldots$
- exponentials — $2^n, \ldots$
- logarithmic — $\log n, \ldots$

and combinations of these.

3.4 'Big Ω' Notation

Big O bounds from above. For example, if our algorithm operates in time $O(n^2)$ we know it grows *no worse* than n^2. But it might be a lot better!

We also want to talk about lower bounds — eg

> No search algorithm (among n distinct objects) using only equality testing can have (worst case time) growth rate better than linear in n.

We use *big* Ω.

Definition

$f(n)$ is $\Omega(g(n))$ if there are a constant $c > 0$ and an integer $n_0 \geq 1$ such that, for all $n \geq n_0$,

$$f(n) \geq cg(n).$$

— f grows no slower than g, for sufficiently large n

— growth rate of f is *bounded from below* by g

Note $f(n)$ is $\Omega(g(n))$ if and only if $g(n)$ is $O(f(n))$.

4. Analysis of Recursive Programs

Previously we've talked about:

- The power of recursive programs.
- The unavoidability of recursive programs (they go hand in hand with recursive data structures).
- The potentially high computational costs of recursive programs.

They are also the most difficult programs we will need to analyse.

It may not be too difficult to express the time or space behaviour recursively, in what we call a *recurrence relation* or *recurrence equation*, but general methods for solving these are beyond the scope of this unit.

However some can be solved by common sense!

Example:

What is the time complexity of the recursive addition program from Topic 3?

```
public static int increment(int i) {return i + 1;}

public static int decrement(int i) {return i - 1;}

public static int add(int x, int y) {
  if (y == 0) return x;
  else return add(increment(x), decrement(y));
}
```

- `if`, `else`, `==`, `return`, etc — constant time
- `increment(x)`, `decrement(y)` — constant time
- `add(increment(x), decrement(y))`? — depends on size of `y`

Recursive call is same again, except `y` is decremented. Therefore, we know the time for `add(...,y)` in terms of the time for

`add(...,decrement(y))`.

More generally, we know the time for size n input in terms of the time for size $n-1$...

$$\begin{aligned} T(0) &= a \\ T(n) &= b + T(n-1), \quad n > 1 \end{aligned}$$

This is called a *recurrence relation*.

We would like to obtain a *closed form* — $T(n)$ in terms of n.

If we list the terms, its easy to pick up a pattern...

$$\begin{aligned} T(0) &= a \\ T(1) &= a + b \\ T(2) &= a + 2b \\ T(3) &= a + 3b \\ T(4) &= a + 4b \\ T(5) &= a + 5b \\ &\vdots \end{aligned}$$

From observing the list we can see that

$$T(n) = bn + a$$

Example:

```
public static int multiply(int x, int y) {
  if (y == 0) return 0;
  else  return add(x, multiply(x, decrement(y)));
}
```

- `if`, `else`, `==`, `return`, etc — constant time
- `decrement(y)` — constant time
- `add` — linear in size of 2nd argument
- `multiply` — ?

We use:

a	const for `add` terminating case
b	const for `add` recursive case
a'	const for `multiply` terminating case
b'	const for `multiply` recursive case
x	for the size of `x`
y	for the size of `y`
$T_{add}(y)$	time for `add` with 2nd argument y
$T(x, y)$	time for multiply with arguments x and y

Tabulate times for increasing y...

$$\begin{aligned}
T(x,0) &= a' \\
T(x,1) &= b' + T(x,0) + T_{add}(0) = b' + a' + a \\
T(x,2) &= b' + T(x,1) + T_{add}(x) = 2b' + a' + xb + 2a \\
T(x,3) &= b' + T(x,2) + T_{add}(2x) = 3b' + a' + (xb + 2xb) + 3a \\
T(x,4) &= b' + T(x,3) + T_{add}(3x) = 4b' + a' + (xb + 2xb + 3xb) + 4a \\
&\vdots
\end{aligned}$$

Can see a pattern of the form

$$T(x,y) = yb' + a' + [1 + 2 + 3 + \cdots + (y-1)]xb + ya$$

We would like a closed form for the term $[1 + 2 + 3 + \cdots + (y-1)]xb$.

Notice that, for example

$$1 + 2 + 3 + 4 = (1 + 4) + (2 + 3) = \frac{4}{2}.5$$

$$1 + 2 + 3 + 4 + 5 = (1 + 5) + (2 + 4) + 3 = \frac{5}{2}.6$$

In general,

$$1 + 2 + \cdots + (y-1) = (\frac{y-1}{2}).y = \frac{1}{2}y^2 - \frac{1}{2}y$$

(Prove inductively!)

Overall we get an equation of the form

$$a'' + b''y + c''xy + d''xy^2$$

for some constants a'', b'', c'', d''.

Dominant term is xy^2:
- — linear in x (hold y constant)
- — quadratic in y (hold x constant)

There are a number of well established results for different types of problems. We will draw upon these as necessary.

5. Summary

Choosing performance measures

- worst case — simple, guarantees upper bounds
- expected case — averages behaviour, need to know probability distribution
- amortized case — may 'distribute' time for expensive operation over those which *must* accompany it

Asymptotic growth rates

- compare algorithms
- compare with inherent performance barriers
- provide simple closed form approximations
- big O — upper bounds on growth
- big Ω — lower bounds on growth

Analysis of recursive programs

- express as recurrence relation
- look for pattern to find closed form
- can then do asymptotic analysis

CITS2200 Data Structures and Algorithms

Topic 8

Objects and Iterators

- Generalising ADTs using objects
  — wrappers, casting
- Iterators for Collection Classes
- Inner Classes

Reading: Lambert and Osborne, Sections 6.3–6.5 and 2.3.5

1. Generalising ADTs to use Objects

Our ADTs so far have stored primitive types.

eg. block implementation of a queue from Topic 5.

```
public class QueueCharBlock {

  private char[] items;
  private int first, last;

  public char dequeue() throws Underflow {
    if (!isEmpty()) {
      char a = items[first];
      first++;
      return a;
    }
    ...
```


This queue will *only* work for characters. We would need to write another for integers, another for a queue of strings, another for a queue of queues, and so on.

Far better would be to write a single queue that worked for *any* type of object.

In object-oriented languages such as Java this is easy, providing we recall a few object-oriented programming concepts from Topic 4.

— inheritance, casting, and wrappers.

1.1 Objects in the ADTs

The easiest part is changing the ADT. (The more subtle part is using it.)

Recall that:

- *every* class is a subclass of the class `Object`
- a variable of a particular class can hold an *instance* of *any subclass* of that class

This means that if we define our ADTs to hold things of type `Object` they can be used with objects from *any other class*!

```
/**
 * Block representation of a queue (of objects).
 */
public class QueueBlock {

  private Object[] items;                    // array of Objects
  private int first;
  private int last;

  public Object dequeue() throws Underflow {  // returns an Object
    if (!isEmpty()) {
      Object a = items[first];
      first++;
      return a;
    }
    else throw new Underflow("dequeuing from empty queue");
  }
```


1.2 Wrappers

The above queue is able to hold any type of object — that is, an instance of any subclass of the class `Object`. (More accurately, it can hold any reference type.)

But there are some commonly used things that are not objects — the primitive types.

In order to use the queue with primitive types, they must be "wrapped" in an object.

Recall from Topic 4 that Java provides wrapper classes for all primitive types.

Autoboxing — Note for Java 1.5

Java 1.5 provides *autoboxing* and *auto-unboxing* — effectively, automatic wrapping and unwrapping done by the compiler.

```
Integer i = 5;
int j = i;
```

However:

- Not a change to the underlying language — the *compiler* recognises the mismatch and substitutes code for you:

  ```
  Integer i = Integer.valueOf(5)
  int j = i.intValue();
  ```


- Can lead to unintuitive behaviour. Eg:

  ```
  Long w1 = 1000L;
  Long w2 = 1000L;
  if (w1 == w2) {
    // do something
  }
  ```

  may not work. Why?

- Can be slow. Eg. if a, b, c, d are Integers, then

  ```
  d = a * b + c
  ```

  becomes

  ```
  d.valueOf(a.intValue() * b.intValue() + c.intValue())
  ```

For more discussion see:

http://chaoticjava.com/posts/autoboxing-tips/

1.3 Casting

Recall that in Java we can assign "up" the hierarchy — a variable of some class (which we call its reference) can be assigned an object whose reference is a subclass.

However the converse is not true — a subclass variable cannot be assigned an object whose reference is a superclass, even if that object is a subclass object.

In order to assign back down the hierarchy, we must use *casting*.

This issue occurs more subtly when using ADTs. Recall our implementation of a queue...


```
public class QueueBlock {
  private Object[] items;                        // array of Objects
  ...
  public Object dequeue() throws Underflow {  // returns an Object
    if (!isEmpty()) {
      Object a = items[first];
      first++;
      return a;
    }
    else...
```

Consider the calling program:

```
QueueBlock q = new QueueBlock();
String s = "OK, I'm going in!";
q.enqueue(s);                          // put it in the queue
s = q.dequeue();                       // get it back off ???
```

The last statement fails. Why?

The queue holds `Object`s. Since `String` is a subclass of `Object`, the queue can hold a `String`, but its reference in the queue is `Object`. (Specifically, it is an element of an array of `Object`s.)

`dequeue()` then returns the "`String`" with reference `Object`.

The last statement therefore asks for something with reference `Object` (the superclass) to be assigned to a variable with reference `String` (the subclass), which is illegal.

We have to cast the `Object` back "down" the hierarchy:

```
s = (String) q.dequeue();       // correct way to dequeue
```


1.4 Generics

Java 1.5 provides an alternative approach. *Generics* allow you to specify the type of a collection class:

```
Stack<String> ss = new Stack<String>();
String s = "OK, I'm going in!";
ss.push(s);
s = ss.pop()
```

Like autoboxing, generics are handled by compiler rewrites — the compiler checks that the type is correct, and substitutes code to do the cast for you.

Writing Generic Classes

```
/**
* A simple generic block stack for
* holding object of type E
**/
class Stack<E> {

  private Object[] block;
  private int size;

  public Stack(int size) {block = new Object[size];}

  public E pop() {return (E) block[--size];}

  public void push(E el) {block[size++] = el;}
}
```


Using Generic Classes

```
public static void main(String[] args){
  //create a Stack of Strings
  Stack<String> s = new Stack<String>(10);
  s.push("abc");
  System.out.println(s.pop());

  //create a stack of Integers
  Stack<Integer> t = new Stack<Integer>(1);
  t.push(7);
  System.out.println(t.pop());
}
```


How Generics Work

The program:

```
Stack<String> ss = new Stack<String>(10);
String s = "OK, I'm going in!";
ss.push(s);
s = ss.pop();
```

is converted to:

```
Stack<Object> ss = new Stack<Object>(10);
String s = "OK, I'm going in!";
ss.push(s);
s = (String) ss.pop();
```

at compile time. Generics allow the compiler to ensure that the casting is correct, rather than the runtime environment.

Some Tricks with Generics...

Note that `Stack<String>` is not a subclass of `Stack<Object>` (because you can't put an `Integer` on a stack of `String`s).

Therefore, polymorphism won't allow you to define methods for all stacks of subclasses of String. e.g.

```
public int printAll(Stack<Object>);
```

Java 5 allows *wildcards* to overcome this problem:

```
public int printAll(Stack<?>);
```

or even

```
public int printAll(Stack<? extends Object>);
```

Generics in Java are complex and are the subject of considerable debate. While you may not need to write them often, it is important you understand them as the Java Collection classes all use generics.

Some interesting articles:

http://www-128.ibm.com/developerworks/java/library/j-jtp01255.html

http://weblogs.java.net/blog/arnold/archive/2005/06/generics_consid_1.html

http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

2. Iterators

It is often necessary to *traverse* a collection — ie look at each item in turn.

Example:

In the lab exercises, you were asked to get characters out of a basic LinkedListChar one at a time and print them on separate lines. Doing this using the supplied methods destroyed the list.

We now know this to be the behaviour of a Stack, which has no public methods for accessing items other than the top one.

Example:

In Topic 3, we developed the simple linked list class. In order to print out the items in the list (without destroying it), we provided the following `toString` method:

```
  public String toString () {
    LinkChar cursor = first;
    String s = "";
    while (cursor != null) {
      s = s + cursor.item;
      cursor = cursor.successor;
    }
    return s;
  }
}
```


This is not a generic approach. If we wanted to look at the items for another purpose — say to print on separate lines, or search for a particular item — we would have to write another method using another loop to do that.

A more standard, generic approach is to use an *iterator*.

An iterator is a companion class to a collection (known as the iterator's *backing collection*), for traversing the collection (ie examining the items one at a time).

An iterator uses standard methods for traversing the items, independently of the backing collection. In Java, these methods are specified by the Iterator interface in `java.util`.

Interface Iterator<E>

The interface has the methods:

- `boolean hasNext()` — return `true` if the iterator has more items
- `E next()` — if there is a next item, return that item and advance to the next position, otherwise throw an exception
- `void remove()` — remove from the underlying collection the last item returned by the iterator. Throws an exception if the immediately preceding operation was not `next`.

  Note: some iterators do not provide this method, and throw an `UnsupportedOperationException` (arguably a poor use of interfaces).

Interface Collection<E>

The underlying collection must also have a method for "spawning" a new iterator over that collection. In Java's Collection interface, this method is called `iterator`.

```
Iterator<E> interator()
```

Have a look at the collection classes:

```
http://www.csse.uwa.edu.au/programming/java/jdk1.5/api/
```

Here, you will find specifications for some of the data structures we have already seen and many we are yet to discuss. You may wonder why we are bothering to implement these data structures at all!

3. Using Java Collections

Built-in (API) classes can be accessed in two ways:

1. by providing their "full name"

   ```
   java.util.LinkedList<String> b = new java.util.LinkedList();
   ```

   Here `LinkedList` is a class in the API package `java.util`.
2. by *importing* the class

   ```
   import java.util.LinkedList;
   .
   .
   .
   LinkedList<String> b = new LinkedList();
   ```
3. You can also import all classes in a package

   ```
   import java.util.*;
   ```


4. java.util

Most general data structures in the Java API are in the `util` package. There are:

1. **Collections**:

   ```
   LinkedList<E>, ArrayList<E>, PriorityQueue<E>,
   Set<E>, Stack<E>, TreeSet<E>
   ```
2. **Maps**:

   ```
   SortedMap<K, V>, TreeMap<K,V>, HashMap<K, V>
   ```
3. and others:

   ```
   Iterator<E>, BitSet
   ```

These allow you to create most of the data structures you will ever need. However, it is important to be able to compare the performance and understand the limitations of each. We will also examine some data structures that are not in the Java API.

4.1 Using Iterators

The following code creates an iterator to access the elements of a queue.

```
public static void main(String[] args) {
  Queue q = new QueueCyclic();
  q.enqueue(new Character('p'));
  q.enqueue(new Character('a'));
  q.enqueue(new Character('v'));
  q.enqueue(new Character('o'));
  Iterator it = q.iterator();
  while(it.hasNext())
    System.out.println(it.next());
}
```


4.2 Implementation — Backing Queue

```
import java.util.Iterator;
public class QueueCyclic implements Queue {

  Object[] items;           // package access for
  int first, last;          // companion class

  public QueueCyclic(int size) {
    items = new Object[size+1];
    first = 0;
    last = size;
  }

  public Iterator iterator() {
    return new BasicQueueIterator(this);
  }
  ...
```


4.3 Implementation — Iterator

```
class BasicQueueIterator implements Iterator {
  private QueueCyclic backingQ;
  private int current;

  BasicQueueIterator(QueueCyclic q) {
    backingQ = q;
    current = backingQ.first;
  }

  public boolean hasNext() {
    return !backingQ.isEmpty() &&
     ((backingQ.last >= backingQ.first &&
       current <= backingQ.last) ||
      (backingQ.last <  backingQ.first &&
       (current >= backingQ.first || current <= backingQ.last)));
  }
```



```
  public Object next() {
    if (!hasNext())
      throw new NoSuchElementException("No more elements.");
    else {
      Object temp = backingQ.items[current];
      current = (current+1)%backingQ.items.length;
      return temp;
    }
  }

  public void remove() {
    throw new UnsupportedOperationException
      ("Cannot remove from within queue.");
  }
}
```

4.4 Fail-fast Iterators

Problem: What happens if backing collection changes during use of an iterator?

eg. multiple iterators that implement `remove`

$\Rightarrow$ can lead to erroneous return data, or exceptions (eg null pointer exception)

One Solution: Disallow further use of iterator (throw exception) when an unexpected change to backing collection has occurred — *fail-fast* method

Changes to the backing collection...

```
public class QueueCyclic implements Queue {

  Object[] items;
  int first, last;
  int modCount;              // number of times modified

  public void enqueue(Object a) {
    if (!isFull()) {
      last = (last + 1) % items.length;
      items[last] = a;
      modCount++;
    }
    else throw new Overflow("enqueuing to full queue");
  }
  ...
```


Changes to the iterator...

```
class BasicQueueIterator implements Iterator {
  private QueueCyclic backingQ;
  private int current;
  private int expectedModCount;

  public Object next() {
    if (backingQ.modCount != expectedModCount)
      throw new ConcurrentModificationException();
    if (!hasNext())
      throw new NoSuchElementException("No more elements.");
    else {
      Object temp = backingQ.items[current];
      current = (current+1)%backingQ.items.length;
      return temp;
    }
  }
```


5. Inner Classes — A Better Way to Iterate

From a software engineering point-of-view, the way we have implemented our iterator is not ideal:

- private variables of `QueueCyclic` were given "package" access so they could be accessed from `BasicQueueIterator` — now they can be accessed from elsewhere too
- `BasicQueueIterator` is only designed to operate correctly with `QueueCyclic` (implementation-specific) but there is nothing preventing applications trying to use it with other implementations

Java provides a tidier way... *inner classes*.

Inner classes are declared within a class:

```
public class MyClass {

  // fields

  // methods

  private class MyInnerClass extends MyInterface {

    // fields

    // methods
  }

  ...

  public MyInterface getMyInnerClass() {...}
}
```


The Inner class is able to access all of the private fields and methods of the outer class.

This gives us a very powerful method to control access to a data structure. The code of the inner class has free range over the instance variables of the outer class, but users can only access the inner class through the prescribed interface.

Inner classes are used extensively in Object Oriented Programming for *call backs*, *remote method invocation*, or *listener* classes.

Cyclic queue implementation using an inner class...

```
import java.util.Iterator;
public class QueueCyclic implements Queue {

  private Object[] items;        // private again
  private int first, last;       //

  ...

  public Iterator iterator() {
    return new BasicQueueIterator();   // no "this"
  }

  private class BasicQueueIterator implements Iterator {

    private int current;
                              // no need to store backing queue
```



```
    private BasicQueueIterator() { // constructed by outer class
      current = first;             // variable accessed directly
    }                              // no passing of backing queue

    public boolean hasNext() {
      return !isEmpty() &&
        ((last >= first && current <= last) ||
         (last < first && (current >= first || current <= last)))
    }
  }  // end of inner class

}  // end of QueueCyclic
```

Q: What other structures have we seen where the use of inner classes would be appropriate?

5.1 Foreach

Another Java 5 feature is the new control structure *foreach*. This can be used for iteration through a collection of elements. For example, the following code:

```
int[] array = {0,2,4};
for (int i : array)
  System.out.println(i);
```

means the same as:

```
int[] array = {0,2,4};
for (int i = 0; i < array.length; i++)
  System.out.println(array[i]);
```

But its use is not just restricted to arrays. Indeed, any object with a `iterator()` method can be used!

6. Summary

- ADTs can be made more general through the use of objects and casting
- generics provide a "cleaner" mechanism of achieving the same functionality
- iterators are a means of traversing a collection of elements
- the use of inner classes simplifies the construction of iterators

CITS2200 Data Structures and Algorithms

Topic 9

Lists

- Why lists?
- List windows
- Specification
- Block representation
- Singly linked representation
- Performance comparisons

Reading: Lambert and Osborne, Sections 9.1–9.4

1. Introduction

Queues and stacks are restrictive — they can only access one position within the data structure ("first" in queue, "top" of stack)

In some applications we want to access a sequence at many different positions:

eg. Text editor — sequence of characters, read/insert/delete at any point
eg. Bibliography — sequence of bibliographic entries
eg. Manipulation of polynomials — see Wood, Section 3.3.
eg. List of addresses
⋮

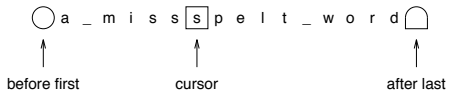
In this section, we introduce the List ADT which generalises stacks and queues.

2. List Windows

We will use the word "window" to refer to a specific position in the list:

- maintain a distinction from "reference" or "index" which are specific implementations
- maintain a distinction from "cursor" which is most commonly used as an application of a window in editing

May be several windows, eg. . .



Our List ADT will provide explicit operations for handling windows.

The following specification assumes that *w* is a Window object, defined in a separate class. Different window objects will be needed for different List implementations

⇒ a List class and a companion Window class will be developed together.

Note: A window class is generally not good software engineering practice as there is no coupling between the *List* and the *window*. Instead, modern ADTs specify list operations in terms of iterators.

3. List Specification

□ **Constructors**

1. *List()*: Initialises an empty list with two associated window positions, *before first* and *after last*.

□ **Checkers**

2. *isEmpty()*: Returns *true* if the list is empty.
3. *isBeforeFirst(w)*: True if *w* is over the before first position.
4. *isAfterLast(w)*: True if *w* is over the after last position.

□ **Manipulators**

5. *beforeFirst(w)*: Initialises *w* to the before first position.
6. *afterLast(w)*: Initialises *w* to the after last position.
7. *next(w)*: Throws an exception if *w* is over the after last position, otherwise moves *w* to the next window position.

8. *previous(w)*: Throws an exception if *w* over is the before first position, otherwise moves *w* to the previous window position.
9. *insertAfter(e,w)*: Throws an exception if *w* is over the after last position, otherwise an extra element *e* is added to the list after *w*.
10. *insertBefore(e,w)*: Throws an exception if *w* is over the before first position, otherwise an extra element *e* is added to the list before *w*.
11. *examine(w)*: Throws an exception if *w* is in the before first or after last position, otherwise returns the element under *w*.
12. *replace(e,w)*: Throws an exception if *w* is in the before first or after last position, otherwise replaces the element under *w* with *e* and returns the old element.
13. *delete(w)*: Throws an exception if *w* is in the before first or after last position, otherwise deletes and returns the element under *w*, and places *w* over the next element.

3.1 Simplifying Assumptions

Allowing multiple windows can introduce problems. Consider the following use of the List ADT:

```
Window w1 = new Window();
Window w2 = new Window();

beforeFirst(w1);            // Initialise first window
next(w1);                   // Place over first element
beforeFirst(w2);            // Initialise second window
next(w2);                   // Place over first element
delete(w1);                 // Delete first element
```

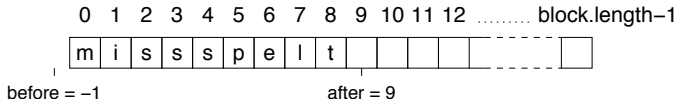
Our specification doesn't say what happens to the second window!

For now, we will assume only one window will be used at a single time.

4. Block Representation

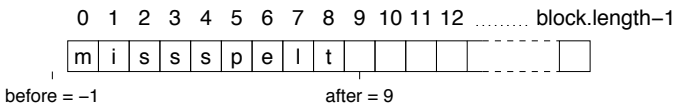
List is defined on a block (array)...

```
public class ListBlock {

  private Object[] block;   // Holds general objects
  private int before;       // An index to the before first position
  private int after;        // iAn ndex to the after last position
```



Constructor

```
public ListBlock (int size) {
  block = new Object[size];
  before = -1;
  after = 0;
}
```



Windows

Some ADTs we have created have implicit windows — eg Queue has a "window" to the first item.

There was a fixed number of these, and they were built into the ADT implementation — eg a member variable `first` held an index to the block holding the queue.

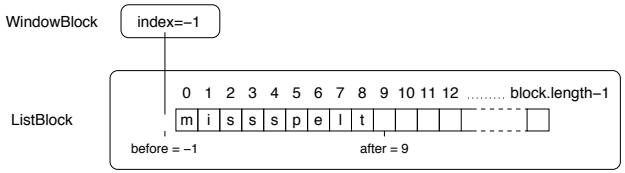
For lists, the user needs to be able to create arbitrarily many windows $\Rightarrow$ we define these as separate objects.

For the block representation, they just hold an index...

```
public class WindowBlock {
  public int index;
  public WindowBlock () {}
}
```

The index is then initialised by a call to *beforeFirst* or *afterLast*.

```
public void beforeFirst (WindowBlock w) {w.index = before;}
```



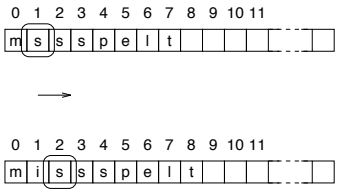
`next` and `previous` simply increment or decrement the window position...

```
public void next (WindowBlock w) throws OutOfBounds {
  if (!isAfterLast(w))  w.index++;
  else
    throw new OutOfBounds("Calling next after list end.");
}
```

`examine` and `replace` are simple array assignments/lookups.

Insertion and deletion may require moving many elements
⇒ worst-case performance — *linear* in size of block

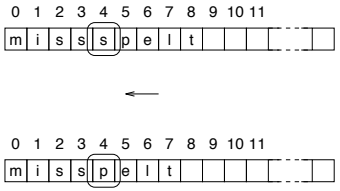
eg. `insertBefore`



From an 'abstract' point of view, the window hasn't moved — it's still over the same element. However, the 'physical' location has changed.

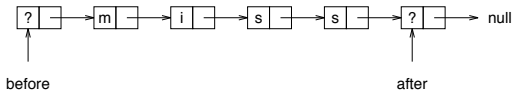

```
public void insertBefore (Object e, WindowBlock w) throws
OutOfBounds, Overflow {
  if (!isFull()) {
    if (!isBeforeFirst(w)) {
      for (int i = after-1; i >= w.index; i--)
        block[i+1] = block[i];
      after++;
      block[w.index] = e;
      w.index++;
    }
    else throw new OutOfBounds ("Inserting before start.");
  }
  else throw new Overflow("Inserting in full list.");
}
```


eg. `delete`



The window has moved from an 'abstract' point of view, although the 'physical' location is the same.

5. Singly Linked Representation



Uses two *sentinel* cells for before first and after last:

- *previous* and *next* always well-defined, even from first or last element
- Constant time implementation for *beforeFirst* and *afterLast*

Empty list just has two sentinel cells. . .

```
public class ListLinked {

  private Link before;
  private Link after;

  public ListLinked () {
    after = new Link(null, null);
    before = new Link(null, after);
  }

  public boolean isEmpty () {return before.successor == after;}
```


Windows

```
public class WindowLinked {
  public Link link;
  public WindowLinked () {link = null;}
}
```

eg.

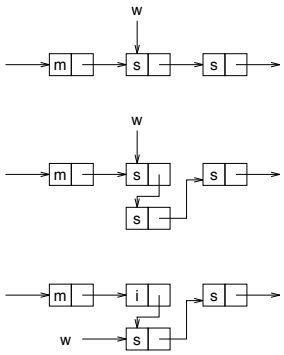
```
  public void beforeFirst (WindowLinked w) {w.link = before;}

  public void next (WindowLinked w) throws OutOfBounds {
    if (!isAfterLast(w))  w.link = w.link.successor;
    else
      throw new OutOfBounds("Calling next after list end.");
  }
```

Why don't we just use a `Link` here?

`insertBefore` and `delete`

Problem — need *previous* cell! To find this takes linear rather than constant time.

One solution: insert *after* and swap items around




```
  public void insertBefore (Object e, WindowLinked w) throws
  OutOfBounds {
    if (!isBeforeFirst(w)) {
      w.link.successor = new Link(w.link.item, w.link.successor);
      if (isAfterLast(w))  after = w.link.successor;
      w.link.item = e;
      w.link = w.link.successor;
    }
    else throw new OutOfBounds ("inserting before start of list");
  }
```

Alternative solution: define window value to be the link to the cell previous to the cell in the window — Exercise.

5.1 Implementing previous

To find the previous element in a singly linked list we must start at the first sentinel cell and traverse the list to the current window, while storing the previous position...

```
public void previous (WindowLinked w) throws
OutOfBounds {
  if (!isBeforeFirst(w)) {
    Link current = before.successor;
    Link previous = before;
    while (current != w.link) {
      previous = current;
      current = current.successor;
    }
    w.link = previous;
  }
  else throw new OutOfBounds ("Calling previous before start of list.");
}
```

This is called *link coupling* — linear in size of list!

Note: We have assumed (as in previous methods) that the window passed is a valid window to *this* List.

In this case if this is not true, Java will throw an exception when the `while` loop reaches the end of the list.

6. Performance Comparisons

Operation	Block	Singly linked
List	1	1
isEmpty	1	1
isBeforeFirst	1	1
isAfterLast	1	1
beforeFirst	1	1
afterLast	1	1
next	1	1
previous	1	n
insertAfter	n	1
insertBefore	n	1
examine	1	1
replace	1	1
delete	n	1

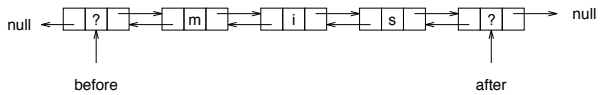
In addition to a *fixed maximum length*, the block representation takes *linear time for insertions and deletions*.

The singly linked representation wins on all accounts except *previous*, which we address in the next sections...

7. Doubly Linked Lists

Singly linked list: *previous* is linear in worst case — may have to search through the whole list to find the previous window position.

One solution — keep references in both directions!



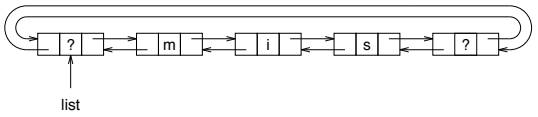
Called a *doubly linked list*.

Advantage: *previous* is similar to *next* — easy to program and constant time.

Disadvantage: extra storage required in each cell, more references to update.

8. Circularly Linked Lists

The doubly linked list has two wasted pointers. If we link these round to the other end...



Called a *circularly linked list*.

Advantages: (over doubly linked)

- Only need a reference to the first sentinel cell.
- Elegant!

Redefine Link

```
public class LinkDouble {
  public Object item;
  public LinkDouble successor;
  public LinkDouble predecessor;         // extra cell
```

Redefine List

```
public class ListLinkedCircular {
  private LinkDouble list;              // just one reference
```


Code for previous

```
public void previous (WindowLinked w) throws
OutOfBounds {
  if (!isBeforeFirst(w))  w.link = w.link.predecessor;
  else throw
    new OutOfBounds("calling previous before start of list ");
}
```

Cf. previous *previous*!

9. Performance — List

Operation	Block	Singly linked	Doubly linked
List	1	1	1
isEmpty	1	1	1
isBeforeFirst	1	1	1
isAfterLast	1	1	1
beforeFirst	1	1	1
afterLast	1	1	1
next	1	1	1
previous	1	n	1
insertAfter	n	1	1
insertBefore	n	1	1
examine	1	1	1
replace	1	1	1
delete	n	1	1

We see that the doubly linked representation has superior performance. This needs to be weighed against the additional space overheads.

Rough rule

- *previous* commonly used $\Rightarrow$ doubly (circularly) linked
- *previous* never or rarely used $\Rightarrow$ singly linked

10. The Simplist ADT

The List ADT provides multiple explicit windows — we need to identify and manipulate windows in any program which uses the code.

If we only need a single window (eg a simple "cursor" editor), we can write a simpler ADT $\Rightarrow$ Simplist.

- single, implicit window (like Queue or Stack) — no need for arguments in the procedures to refer to the window position

We'll also provide only one window initialisation operation, *beforeFirst*

We'll show that, because of the single window, all operations except *beforeFirst* can be implemented in constant time using a singly linked list! Uses a technique called *pointer reversal* (or *reference reversal*).

We also give a useful amortized result for *beforeFirst* which shows it will not be too expensive over a collection of operations.

10.1 Simplist Specification

□ **Constructor**

1. *Simplist()*: Creates an empty list with two window positions, before first and after last, and the window over before first.

□ **Checkers**

2. *isEmpty()*: Returns true if the simplist is empty.
3. *isBeforeFirst()*: True if the window is over the before first position.
4. *isAfterLast()*: True if the window is over the after last position.

□ **Manipulators**

5. *beforeFirst()*: Initialises the window to be the before first position.
6. *next()*: Throws an exception if the window is over the after last position, otherwise the window is moved to the next position.
7. *previous()*: Throws an exception if the window is over the before first position, otherwise the window is moved to the previous position.

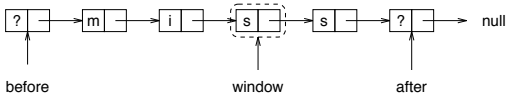
8. *insertAfter(e)*: Throws an exception if the window is over the after last position, otherwise an extra element *e* is added to the simplist after the window position.
9. *insertBefore(e)*: Throws an exception if the window is over the before first position, otherwise an extra element *e* is added to the simplist before the window position.
10. *examine()*: Throws an exception if the window is over the before first or after last positions, otherwise returns the value of the element under the window.
11. *replace(e)*: Throws an exception if the window is over the before first or after last positions, otherwise replaces the element under the window with *e* and returns the replaced element.
12. *delete()*: Throws an exception if the window is over the before first or after last positions, otherwise the element under the window is removed and returned, and the window is moved to the following position.

10.2 Singly Linked Representation

Again block and doubly linked versions are possible — same advantages/disadvantages as the List ADT. Our aim is to show an improvement in the singly linked representation.

Since the window position is not passed as an argument, we need to store it in the data structure...

```
public class SimplistLinked {

  private Link before;
  private Link after;
  private Link window;
```



10.3 Reference (or "Pointer") Reversal

The window starts at *before first* and can move up and down the list using *next* and *previous*.

Problem

As for the singly linked representation, *previous* can be found by link coupling, but this takes linear time.

Solution

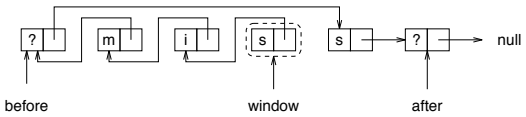
Q: What do you always do when you walk into a labyrinth?

Solution...

- point successor fields behind you backwards
- point successor fields in front of you forwards

Problem: window cell can only point one way.

Solution: the before first successor no longer needs to reference the first element of the list (we can always follow the references back). Instead, use it to reference the cell after the window, and point the window cell backwards.



⇒ *reference (pointer) reversal*

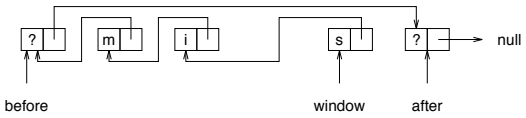
Exercise

```
public void previous() {
  if (!isBeforeFirst) {




  }
  else throw
    new OutOfBounds("calling previous before start of list");
}
```

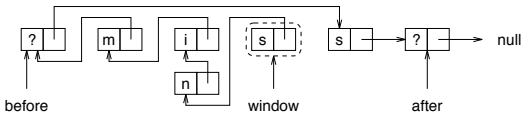
What is the performance of *previous*?

Other operations also require reference reversal.

delete...



insertBefore...



Disadvantage(?): A little more complex to code.

Advantage: Doesn't require the extra space overheads of a doubly linked list.

Advantage outweighs disadvantage — you only code once; might use many times!

Problem: These operations only reverse one or two references, but what about *beforeFirst*? Must reverse references back to the beginning. (Note that *previous* and *next* now modify the list *structure*.)

$\Rightarrow$ linear in worst case

What about amortized case?...

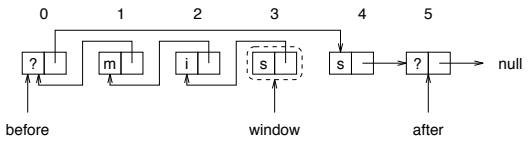
10.4 Amortized Analysis

Consider the operation of the window prior to any call to *beforeFirst* (other than the first one).

Must have started at the before first position after last call to *beforeFirst*.

Can only have moved forward by calls to *next* and *insertBefore*.

If window is over the ith cell (numbering from 0 at before first), there must have been i calls to *next* and *insertBefore*. Each is constant time, say 1 "unit".



beforeFirst requires i constant time "operations" (reversal of i pointers) — takes i time "units".

Total time: $2i$. Total number of operations: $i + 1$.

Average time per operation: ≈ 2

Average time over a sequence of operations is (roughly) constant!

Formally: *Each sequence of n operations takes $O(n)$ time; ie each operation takes constant time in the amortized case.*

10.5 Performance Comparisons — Simplist

Operation	Block	Singly linked	Doubly linked
Simplist	1	1	1
isEmpty	1	1	1
isBeforeFirst	1	1	1
isAfterLast	1	1	1
beforeFirst	1	1^a	1
next	1	1	1
previous	1	1	1
insertAfter	n	1	1
insertBefore	n	1	1
examine	1	1	1
replace	1	1	1
delete	n	1	1

a — amortized bound

11. Summary

- Lists generalise stacks and queues by enabling insertion, examination, and deletion at any point in the sequence.
- Insertion, examination, and deletion are achieved using *windows* on the list.
- Explicit window manipulation is included in the specification of our List ADT.
- A block representation restricts the list size and results in linear time performance for insertions and deletions.
- A singly linked representation allows arbitrary size lists, and offers constant time performance in all operations except *previous*.

Doubly (and Circularly) Linked Lists

- Constant time performance on all operations
- Needs extra space

Simplists

- Block, doubly, and circularly linked representations
  - same performance as the List ADT
- Singly linked representation with reference reversal
  - constant amortized case performance in all operations

CITS2200 Data Structures and Algorithms

Topic 10

Maps and Binary Search

- Definitions — what is a map (or function)?
- Specification
- List-based representation (singly linked)
- Sorted block representation
  – binary search, performance of binary search
- Performance comparison

Reading: Lambert and Osborne, Sections 13.1, 13.3, and 10.2

1. What is a Map (or Function)?

Some definitions...

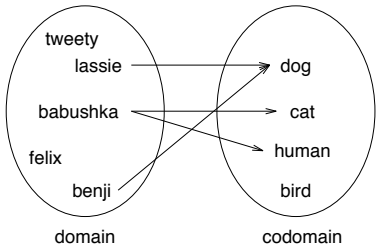
relation — set of n-tuples
eg. $\{\langle 1, i, a\rangle, \langle 2, ii, b\rangle, \langle 3, iii, c\rangle, \langle 4, iv, d\rangle, \ldots\}$

binary relation — set of pairs (2-tuples)
eg. $\{\langle lassie, dog\rangle, \langle babushka, cat\rangle, \langle benji, dog\rangle, \langle babushka, human\rangle, \ldots\}$

domain — set of values which can be taken on by the first item of a binary relation
eg. $\{lassie, babushka, benji, felix, tweety\}$

codomain — set of values which can be taken on by the second item of a binary relation
eg. $\{dog, cat, human, bird\}$

Example



dog is called the *image* of *lassie* under the relation

map (or function) — binary relation in which each element in the domain is mapped to *at most one* element in the codomain (*many-to-one*)
eg.

$$\text{Affiliation} = \{ \begin{array}{lcl} \langle \text{Turing} & , & \text{Manchester} \rangle \\ \langle \text{Von Neumann} & , & \text{Princeton} \rangle \\ \langle \text{Knuth} & , & \text{Stanford} \rangle \\ \langle \text{Minsky} & , & \text{MIT} \rangle \\ \langle \text{Dijkstra} & , & \text{Texas} \rangle \\ \langle \text{McCarthy} & , & \text{Stanford} \rangle \} \end{array}$$

Shorthand notation: eg. affiliation(Knuth) = Stanford

partial map — not every element of the domain has an image under the map (ie, the image is undefined for some elements)

2. Aside: Why Study Maps?

A Java method is a function or map — why implement our own map as an ADT?

- Create, modify, and delete maps during use.

  eg. a map of affiliations may change over time — Turing started in Cambridge, but moved to Manchester after the war.

  A Java program cannot modify itself (and therefore its methods) during execution (some languages, eg Prolog, can!)
- Java methods just return a result — we want more functionality (eg. ask "is the map defined for a particular domain element?")

3. Map Specification

□ **Constructor**

1. *Map()*: create a new map that is undefined for all domain elements.

□ **Checkers**

2. *isEmpty()*: return *true* if the map is empty (undefined for all domain elements), *false* otherwise.
3. *isDefined(d)*: return *true* if the image of *d* is defined, *false* otherwise.

□ **Manipulators**

4. *assign(d,c)*: assign *c* as the image of *d*.
5. *image(d)*: return the image of *d* if it is defined, otherwise throw an exception.
6. *deassign(d)*: if the image of *d* is defined return the image and make it undefined, otherwise throw an exception.

4. List-based Representation

A map can be considered to be a list of pairs. Providing this list is *finite*, it can be implemented using one of the techniques used to implement the list ADT.

Better still, it can be built *using* the list ADT!

(Providing it can be done efficiently — recall the example of *overwrite*, using *insert* and *delete*, in a text editor based on the list ADT.)

Question: Which List ADT should we use?

- Require arbitrarily many assignments.
- Do we need *previous*?

Implementation...

```
public class MapLinked {

  private ListLinked list;

  public MapLinked () {
    list = new ListLinked();
  }









}
```

4.1 Pairs

We said a (finite) map could be considered a list of pairs — need to define a Pair object...

```
public class Pair {

  public Object item1;       // the first item (or domain item)
  public Object item2;       // the second item (or codomain item)

  public Pair (Object i1, Object i2) {
    item1 = i1;
    item2 = i2;
    }
```



```
  // determine whether this pair is the same as the object passed
  // assumes appropriate ''equals'' methods for the components
  public boolean equals(Object o) {
    if (o == null)  return false;
    else if (!(o instanceof Pair))  return false;
    else  return item1.equals( ((Pair)o).item1) &&
                 item2.equals( ((Pair)o).item2);
  }

  // generate a string representation of the pair
  public String toString() {
    return "< "+item1.toString()+" , "+item2.toString()+" >";
  }
}
```


4.2 Example — Implementation of image

```
public Object image (Object d) throws ItemNotFound {
  WindowLinked w = new WindowLinked();
  list.beforeFirst(w);
  list.next(w);
  while (!list.isAfterLast(w) &&
         !((Pair)list.examine(w)).item1.equals(d) )  list.next(w);
  if (!list.isAfterLast(w))  return ((Pair)list.examine(w)).item2;
  else throw new ItemNotFound("no image for object passed");
}
```

Notes:

1. `!list.isAfterLast(w)` must precede `list.examine(w)` in the condition for the loop — why??
2. Note use of parentheses around casting so that the field reference (eg `.item1`) applies to the cast object (`Pair` rather than `Object`).
3. Assumes appropriate *equals* methods for each of the items in a pair.

4.3 Performance

Map and *isEmpty* make trivial calls to constant-time list ADT commands.

The other four operations all require a sequential search within the list $\Rightarrow$ linear in the size of the defined domain ($O(n)$)

Performance using (singly linked) List ADT

Operation	
Map	1
isEmpty	1
isDefined	n
assign	n
image	n
deassign	n

If the maximum number of pairs is predefined, and we can specify a total ordering on the domain, better efficiency is possible...

5. Sorted-block Representation

Some of the above operations take linear time because they need to search for a domain element. The above program does a linear search.

Q: Are any more efficient searches available for arbitrary *linked* list?

5.1 Party Games...

Q: I've chosen a number between 1 and 1000. What is it?

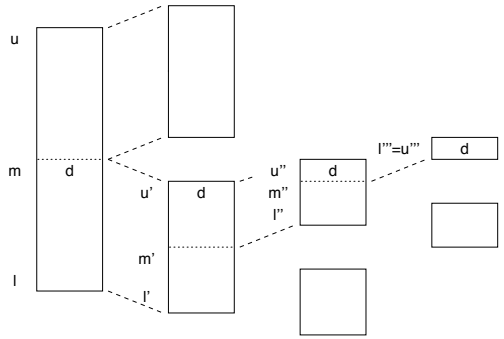
Q: I've chosen a number between 1 and 1000. If you make an incorrect guess I'll tell whether its higher or lower. You have 10 guesses. What is it?

Q: I'm going to choose a number between 1 and n. You have 5 guesses. What is the maximum value of n for which you are certain to get my number right?

Exercise: Write a recursive Java method *guessrange(m)* that returns the maximum number n for which you can always obtain a correct answer with m guesses.

5.2 Binary Search

An algorithm for binary search...



Assume `block` is defined as:

```
private Pair[] block;
```

Then binary search can be implemented as follows...

```
// recursive implementation of binary search
// uses String representations generated by toString()
// for comparison
// returns index to the object if found, or -1 if not found

protected int bSearch (Object d, int l, int u) {
  if (l == u) {
    if (d.toString().compareTo(block[l].item1.toString()) == 0)
      return l;
    else return -1;
  }
  else {
    int m = (l + u) / 2;
    if (d.toString().compareTo(block[m].item1.toString()) <= 0)
      return bSearch(d,l,m);
    else  return bSearch(d,m+1,u);
  }
}
```


Note: `compareTo` is an instance method of String — returns 0 if its argument matches the String, a value < 0 if the String is lexicographically less than the argument, and a value > 0 otherwise.

Exercise: Can `bSearch` be implemented using only the abstract operations of the list ADT?

5.3 Performance of Binary Search

We will illustrate performance in two ways.

One way of looking at the problem, to get a feel for it, is to consider the biggest list of pairs we can find a solution for with m calls to `bSearch`.

Calls to `bSearch`	Size of list
1	1
2	$1 + 1$
3	$2 + 1 + 1$
4	$4 + 2 + 1 + 1$
⋮	
m	$(2^{m-2} + 2^{m-3} + \cdots + 2^1 + 2^0) + 1$
	$= (2^{m-1} - 1) + 1$
	$= 2^{m-1}$

That is, $n = 2^{m-1}$ or $m = \log_2 n + 1$.

This view ignores the "intermediate" size lists — those which aren't a maximum size for a particular number of calls.

An alternative is to look at the number of calls needed for increasing input size. Can be expressed as a recurrence relation...

$$
\begin{array}{lcll}
T_1 & = & 1 & \\
\hline
T_2 & = & 1 + T_1 & = 2 \\
\hline
T_3 & = & 1 + T_2 & = 3 \\
T_4 & = & 1 + T_2 & = 3 \\
\hline
T_5 & = & 1 + T_3 & = 4 \\
T_6 & = & 1 + T_3 & = 4 \\
T_7 & = & 1 + T_4 & = 4 \\
T_8 & = & 1 + T_4 & = 4 \\
\hline
T_9 & = & 1 + T_5 & = 5 \\
 & & \vdots &
\end{array}
$$

The rows for which n is an integer power of 2...

$$
\begin{array}{lcll}
T_1 & = & 1 & \\
\hline
T_2 & = & 1 + T_1 & = 2 \\
\hline
T_4 & = & 1 + T_2 & = 3 \\
\hline
T_8 & = & 1 + T_4 & = 4 \\
\hline
 & & \vdots &
\end{array}
$$

...correspond to those in the earlier table.

For these rows we have

$$
\begin{aligned}
T_{2^0} &= 1 \\
T_{2^m} &= 1 + T_{2^{m-1}} \\
&= 1 + 1 + T_{2^{m-2}} \\
&\vdots \\
&= \underbrace{1 + 1 + \cdots + 1}_{m+1 \texttt{ times}} \\
&= m + 1
\end{aligned}
$$

Substituting $n = 2^m$ or $m = \log_2 n$ once again gives

$$T_n = \log_2 n + 1.$$

What about the cases where n is not an integer power of 2?
$\Rightarrow$ Exercises.

It can be shown (see Exercises) that T_n is $O(\log n)$.

6. Comparative Performance of Operations

isDefined and *image* simply require binary search, therefore they are $O(\log n)$ — much better than singly linked list representation.

However, since the block is sorted, both *assign* and *deassign* may need to move blocks of items to maintain the order. Thus they are

$$\max(O(\log n), O(n)) = O(n).$$

In summary...

Operation	Linked List	Sorted Block
Map	1	1
isEmpty	1	1
isDefined	n	$\log n$
assign	n	n
image	n	$\log n$
deassign	n	n

Sorted block may be best choice if:

1. map has fixed maximum size
2. domain is totally ordered
3. map is fairly static — mostly reading (*isDefined, image*) rather than writing (*assign, deassign*)

Otherwise linked list representation is probably better.

7. Arrays as Maps

We have seen two representations for maps:

- linked list — linear time accesses.
- sorted block — logarithmic for reading, linear for writing.

One very frequently used subtype of the map is an array. An array is simply a map (function) whose domain is a cross product of (that is, tuples from) sets of ordinals.

We will assume all domain items are tuples of integers.

eg. The array

	1	2	3	4	5
false	6.6	2.8	0.4	6.0	0.1
true	3.4	7.2	9.6	4.0	9.9

could be represented by the map

$$\{\langle\langle 0,1\rangle, 6.6\rangle, \langle\langle 0,2\rangle, 2.8\rangle, \langle\langle 0,3\rangle, 0.4\rangle, \ldots, \\ \ldots, \langle\langle 1,4\rangle, 4.0\rangle, \langle\langle 1,5\rangle, 9.9\rangle\}.$$

We will also assume the arrays are bounded in size, so we can store the items in a contiguous block of memory locations. (This can be simulated in Java using a 1-dimensional array.)

An *addressing function* can be used to translate the array indices into the actual location of the item in the block.

Accesses are more efficient for this subtype of maps — *constant time in all operations*.

$\Rightarrow$ good example of a subtype over which operations are more efficient.

7.1 Specification

□ **Constructors**

1. *Array()*: creates a new array that is undefined everywhere.

□ **Manipulators**

2. *assign(d,c)*: assigns c as the image of d.
3. *image(d)*: returns the image of tuple *d* if it is defined, otherwise throws an exception.

7.2 Lexicographically Ordered Representations

Lexicographic Ordering with 2 Indices

Pair $\langle i, j \rangle$ is *lexicographically earlier* than $\langle i', j' \rangle$ if $i < i'$ or ($i = i'$ and $j < j'$).

Best illustrated by an array with indices of type char:
first index: `a,...,d`
second index: `a,...,e`

Then entries are indexed in the order

$$\langle \mathtt{a}, \mathtt{a} \rangle, \langle \mathtt{a}, \mathtt{b} \rangle, \langle \mathtt{a}, \mathtt{c} \rangle, \langle \mathtt{a}, \mathtt{d} \rangle, \langle \mathtt{a}, \mathtt{e} \rangle, \langle \mathtt{b}, \mathtt{a} \rangle, \langle \mathtt{b}, \mathtt{b} \rangle, \ldots \langle \mathtt{d}, \mathtt{d} \rangle, \langle \mathtt{d}, \mathtt{e} \rangle$$

$\Rightarrow$ 'alphabetic' order (*lexicon* $\approx$ dictionary).

	a	b	c	d	e
a	1	2	3	4	5
b	6	7	8	9	10
c	11	12	13	14	15
d	16	17	18	19	20

Also called *row-major* order.

Implementation straightforward — indexed block (from 1 to 20 in the example).

Wish to access entries in constant time.

Addressing function: $\alpha : 1..m \times 1..n \to \mathcal{N}$

$$\alpha(i, j) = (i - 1) \times n + j \qquad 1 \leq i \leq m,\ 1 \leq j \leq n$$

Reverse-lexicographic Order

Similar to lexicographic, but indices swapped around...

Pair $\langle i, j \rangle$ is *reverse-lexicographically earlier* than $\langle i', j' \rangle$ if $j < j'$ or ($j = j'$ and $i < i'$).

	a	b	c	d	e
a	1	5	9	13	17
b	2	6	10	14	18
c	3	7	11	15	19
d	4	8	12	16	20

Also called *column-major* order.

Addressing function:

$$\alpha(i, j) = (j - 1) \times m + i \qquad 1 \leq i \leq m,\ 1 \leq j \leq n$$

7.3 Shell-ordered Representation

An alternative to lexicographic ordering that has advantages in terms of extendibility.

	a	b	c	d	e
a	1	2	5	10	17
b	4	3	6	11	18
c	9	8	7	12	19
d	16	15	14	13	20
	25	*24*	*23*	*22*	*21*

Built up shell by shell.

The kth shell contains indices $\langle i, j \rangle$ such that $k = \max(i, j)$.

Notice that the kth shell "surrounds" a block containing $(k-1)^2$ cells, and forms a block containing k^2 cells.

$\Rightarrow$ To find entries in the first half of the shell, add to $(k-1)^2$. To find entries in the second half of the shell, subtract from k^2.

Addressing function:

$$\alpha(i,j) = \begin{cases} (k-1)^2 + i & i < k \\ k^2 + 1 - j & \text{otherwise} \end{cases} \qquad k = \max(i,j).$$

Disadvantage

May waste a lot of space. Worst case is a one-dimensional array of size n — wastes $n^2 - n$ cells.

A related problem occurs with all these representations when only a small number of the entries are used.

eg. matrices in which most entries are zero.

In this case more complex schemes can be used — trade space for performance. See Wood, Sec. 4.4.

Advantage

- *All* arrays use the same addressing function — independent of number of rows and columns.
- Extendibility...

7.4 Extendibility

In lexicographic ordering new rows can be added (if memory is available) *without changing* the values assigned to existing cells by the addressing function.

	a	b	c	d	e
a	1	2	3	4	5
b	6	7	8	9	10
c	11	12	13	14	15
d	16	17	18	19	20
e	21	22	23	24	25
f	26	27	28	29	30

$$\alpha(i,j) = (i-1) \times \underbrace{n}_{\text{no change}} + j \qquad 1 \le i \le m,\ 1 \le j \le n$$

We say the lexicographic addressing function is *row extendible*.

Adding a row takes $O(\text{size of row})$.

However it is not *column extendible*. Adding a new column means changing the values, *and hence locations*, of existing entries.

Q: What is an example of a worst case array for adding a column?

This is $O(\text{size of array})$ time operation.

Similarly, reverse lexicographic ordering is column extendible...

	a	b	c	d	e	f	g
a	1	5	9	13	17	21	25
b	2	6	10	14	18	22	26
c	3	7	11	15	19	23	27
d	4	8	12	16	20	24	28

$$\alpha(i,j) = (j-1) \times \underbrace{m}_{\text{no change}} + i \qquad 1 \le i \le m,\ 1 \le j \le n$$

...but not row extendible.

Shell ordering, on the other hand, is both row and column extendible...

	a	b	c	d	e	f
a	1	2	5	10	17	26
b	4	3	6	11	18	27
c	9	8	7	12	19	28
d	16	15	14	13	20	29
e	25	24	23	22	21	30
	36	*35*	*34*	*33*	*32*	*31*

This is because the addressing function is independent of m and n...

$$\alpha(i,j) = \begin{cases} (k-1)^2 + i & i < k \\ k^2 + 1 - j & \text{otherwise} \end{cases} \qquad k = \max(i,j).$$

for $1 \le i \le m,\ 1 \le j \le n$.

7.5 Performance

Operation	Lexicographic	Reverse-lexicographic	Shell
Array	1	1	1
Assign	1	1	1
Image	1	1	1

Accesses are more efficient for this subtype of maps — *constant time in all operations*.

Disadvantages:

- restricted domain (integers).
- lexicographical based representations are not easily extendible — they require moving elements should additional columns/rows be added.
- shell-ordered representation waste space for non-square arrays.

8. Summary

- A map (or function) is a many-to-one binary relation.
- Implementation using linked list
  - can be arbitrarily large
  - reading from and writing to the map takes linear time
- Sorted block implementation
  - fixed maximum size
  - requires ordered domain
  - reading is logarithmic, writing is linear
- Arrays are a commonly used subtype of maps which can be treated more efficiently
  - implemented using a 1D block of memory and an addressing function

CITS2200 Data Structures and Algorithms

Topic 11

Trees

- Why trees?
- Binary trees
  - definitions: size, height, levels, skinny, complete
- Trees, forests and orchards
- Tree traversal
  - depth-first, level-order
  - traversal analysis

Reading: Lambert and Osborne, Chapter 11

1. Why Study Trees?

Wood...

"Trees are ubiquitous."

Examples...

genealogical trees	organisational trees
biological hierarchy trees	evolutionary trees
population trees	book classification trees
probability trees	decision trees
induction trees	design trees
graph spanning trees	search trees
planning trees	encoding trees
compression trees	program dependency trees
expression/syntax trees	gum trees
⋮	⋮

Also, *many other data structures are based on trees!*

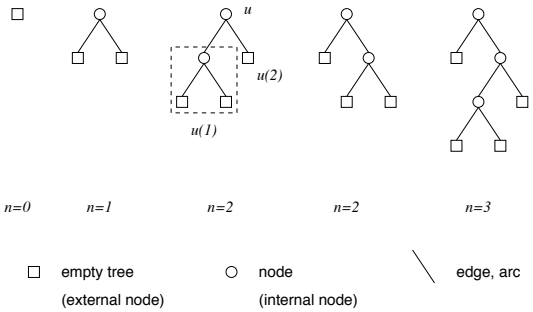
2. Binary Trees

Definition

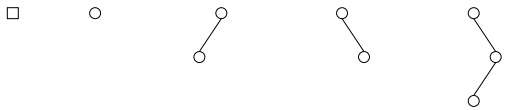
A *binary (indexed) tree* T of n nodes, $n \geq 0$, either:

- *is empty*, if $n = 0$, *or*
- *consists of a* root node u *and two binary trees* $u(1)$ and $u(2)$ of n_1 and n_2 nodes respectively such that $n = 1 + n_1 + n_2$.
  - $u(1)$: *first* or *left subtree*
  - $u(2)$: *second* or *right subtree*

  The function u is called the *index*.



We will often omit external nodes...



More terminology...

Definition

Let w_1, w_2 be the roots of the subtrees u_1, u_2 of u. Then:

- u is the *parent* of w_1 and w_2.
- w_1, w_2 are the (*left* and *right*) *children* of u. $u(i)$ is also called the i^{th} child.
- w_1 and w_2 are *siblings*.

Grandparent, *grandchild*, etc are defined as you would expect.

A *leaf* is an (internal) node whose left and right subtrees are both empty (external nodes).

The external nodes of a tree define its *frontier*.

In the following assume T is a tree with $n \geq 1$ nodes.

Definition

Node v is a *descendant* of node u in T if:

1. v is u, or
2. v is a child of some node w, where w is a descendant of u.

Proper descendant: $v \neq u$

Left descendant: u itself, or descendant of left child of u

Right descendant: u itself, or descendant of right child of u

Q: How would you define "v is *to the left of* u"?

Q: How would you define descendant without using recursion?

2.1 Size and Height of Binary Trees

The *size* of a binary tree is the number of (internal) nodes.

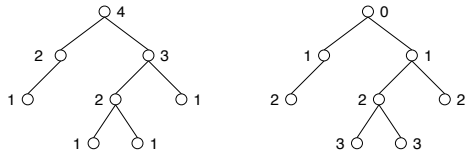
The *height* of a binary tree T is the length of the longest chain of descendants. That is:

- 0 if T is empty,
- $1 + \max(height(T_1), height(T_2))$ otherwise, where T_1 and T_2 are subtrees of the root.

The height of a node u is the height of the subtree rooted at u.

The *level* of a node is the "distance" from the root. That is:

- 0 for the root node,
- 1 plus the level of the node's parent, otherwise.

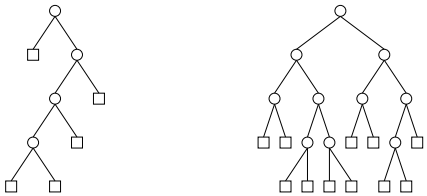


2.2 Skinny and Complete Trees

Since we will be doing performance analyses of tree representations, we will be interested in worst cases for height vs size.

skinny — every node has at most one child (internal) node

complete (fat) — external nodes (and hence leaves) appear on at most two adjacent levels

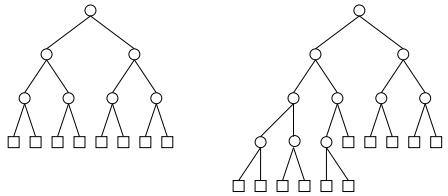


For a given size, skinny trees are the highest possible, and complete trees the lowest possible.

We also identify the following subclasses of complete:

perfect — all external nodes (and leaves) on one level

left-complete — leaves at lowest level are in leftmost position



2.3 Relationships between Height and Size

The above relationships can be formalised/extended to the following:

1. A binary tree of height h has size at least h.
2. A binary tree of height h has size at most $2^h - 1$.
3. A binary tree of size n has height at most n.
4. A binary tree of size n has height at least $\lceil \log(n+1) \rceil$.

Exercise

For each of the above, what class of binary tree represents an upper or lower bound?

Exercise

Prove (2).

3. Trees, Forests, and Orchards

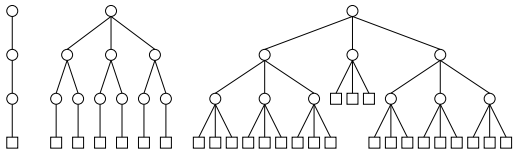
A general *tree* or *multiway (indexed) tree* is defined in a similar way to a binary tree except that a parent node does not need to have exactly two children.

Definition

A *multiway (indexed) tree* T of n nodes, $n \geq 0$, either:

- is empty, if $n = 0$, or
- consists of a root node u, an integer $d \geq 1$ called the *degree* of u, and d multiway trees $u(1), u(2), \ldots, u(d)$ with sizes $n_1, n_2, \ldots, n_d$ respectively such that

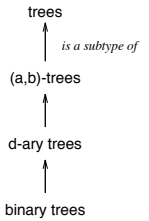
$$n = 1 + n_1 + n_2 + \cdots + n_d.$$

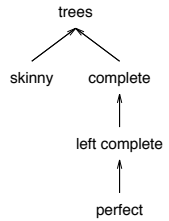


A tree is a d-*ary tree* if $d_u = d$ for all (internal) nodes u. We have already looked at binary (2-ary) trees. Above is a unary (1-ary) tree and a ternary (3-ary) tree.

A tree is an (a, b)-*tree* if $a \leq d_u \leq b$, $(a, b \geq 1)$, for all u. Thus the above are all (1,3)-trees, and a binary tree is a (2,2)-tree.

Some trees of tree types!





3.1 Forests and Orchards

Removing the root of a tree leaves a collection of trees called a *forest*. An ordered forest is called an *orchard*. Thus:

forest — (possibly empty) set of trees

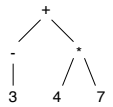
orchard — (possibly empty) queue or list of trees

3.2 Annotating Trees

The trees defined so far have no values associated with nodes. In practice it is normally such values that make them useful.

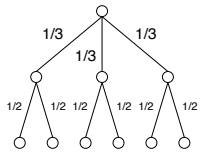
We call these values *annotations* or *labels*.

eg. a *syntax* or *formation* tree for the expression $-3 + 4 * 7$



eg. The following is a probability tree for a problem like:

> "Of the students entering a language course, one half study French, one third Indonesian, and one sixth Warlpiri. In each stream, half the students choose project work and half choose work experience. What is the probability that Björk, a student on the course, is doing Warlpiri with work experience?"



In examples such as this one, it often seems more natural to associate labels with the "arcs" joining nodes. However, this is equivalent to moving the values down to the nodes.

As with the list ADT, we will associate elements with the nodes.

4. Tree Traversals

Why traverse?

- search for a particular item
- test equality (isomorphism)
- copy
- create
- display

We'll consider two of the simplest and most common techniques:

depth-first — follow branches from root to leaves
breadth-first (level-order) — visit nodes level by level

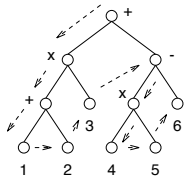
(More in Algorithms or Algorithms for AI...!)

4.1 Depth-first Traversal

Preorder Traversal

(Common garden "left to right", "backtracking", depth-first search!)

```
if(!t.isEmpty()) {
   visit root of t;
   perform preorder traversal of left subtree;
   perform preorder traversal of right subtree;
}
```



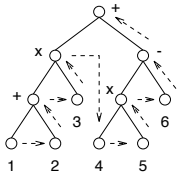
(Generates a *prefix expression*

$$+ \times + 1\ 2\ 3 - \times 4\ 5\ 6$$

Sometimes used because no brackets are needed — no ambiguity.)

Postorder Traversal

```
if(!t.isEmpty()) {
   perform postorder traversal of left subtree;
   perform postorder traversal of right subtree;
   visit root of t;
}
```



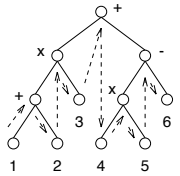
(Generates a *postfix expression*

$$1\ 2 + 3 \times 4\ \ 5 \times 6 - +$$

Also non-ambiguous — as used by, eg. HP calculators.)

Inorder Traversal

```
if(!t.isEmpty()) {
   perform inorder traversal of left subtree;
   visit root of t;
   perform inorder traversal of right subtree;
}
```



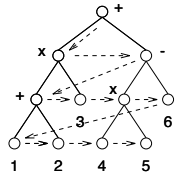
(Generates an *infix expression*

$$1 + 2 \times 3 + 4 \times 5 - 6$$

Common, easy to read, but ambiguous.)

4.2 Level-order (Breadth-first) Traversal

Starting at root, visit nodes level by level (left to right):



Doesn't suit recursive approach. Have to jump from subtree to subtree.

Solution:

- need to keep track of subtrees yet to be visited — ie need a data structure to hold (windows to) subtrees (or Orchard)
- each internal node visited spawns two new subtrees
- new subtrees visited *only after* those already waiting

⇒ Queue of (windows to) subtrees!

Algorithm

```
place tree (root window) in empty queue q;
while (!q.isEmpty()) {
   dequeue first item;
   if (!external node) {
      visit its root node;
      enqueue left subtree (root window);
      enqueue right subtree (root window);
   }
}
```

4.3 Traversal Analysis

Time

The traversals we have outlined all take $O(n)$ time for a binary tree of size n.

Since all n nodes must be visited, we require $\Omega(n)$ time
$\Rightarrow$ asymptotic performance cannot be improved.

Space

Depth-first: Recursive implementation requires memory (from Java's local variable stack) for each method call $\Rightarrow$ proportional to height of tree

- worst case: skinny, size n implies height n
- expected case: much better (depends on distribution considered — see Wood Section 5.3.3)
- best case: *exercise*...

Iterative implementation is also possible.

Level-order: Require memory for queue.

Depends on tree *width* — maximum number of nodes on a single level.

Maximum length of queue is bounded by twice the width.

- best case: skinny, width 2
- worst case: *exercise*...

5. Summary

- Trees are not only common "in their own right", but form a basis for many other data structures.
- Definitions — binary trees, trees, forests, orchards, annotated trees
- Properties — size, height, level, skinny, complete, perfect, d-ary, (a, b)
- Covered important, common traversal strategies
  - depth-first: preorder, postorder, inorder
  - level-order (breadth-first)

Next — tree representations...

CITS2200 Data Structures and Algorithms

Topic 12

Tree Implementations

- Tree Specifications
- Block representation of Bintree
- Recursive representations of Bintree
- Representation of multiway Trees

Reading: Lambert and Osborne, Sections 11.2-11.6

1. Specifications

Binary Tree (Bintree)

Just like the list ADT, we will have *windows* over nodes. The operations are similar, with *previous* and *next* replaced by *parent* and *child* and so on. Some are a little more complex because of the more complex structure. . .

□ **Constructor**

1. *Bintree()*: creates an empty binary tree.

□ **Checkers**

2. *isEmpty()*: returns *true* if the tree is empty, *false* otherwise.
3. *isRoot(w)*: returns *true* if *w* is over the root node (if there is one), *false* otherwise.
4. *isExternal(w)*: returns *true* if *w* is over an external node, *false* otherwise.
5. *isLeaf(w)*: returns *true* if *w* is over a leaf node, *false* otherwise.

□ **Manipulators**

6. *initialise(w)*: set *w* to the window position of the single external node if the tree is empty, or the window position of the root otherwise.
7. *insert(e,w)*: if w is over an external node replace it with an internal node with value e (and two external children) and leave w over the internal node, otherwise throw an exception.
8. *child(i,w)*: throw an exception if *w* is over an external node or *i* is not 1 or 2, otherwise move the window to the *i*-th child.
9. *parent(w)*: throw an exception if the tree is empty or w is over the root node, otherwise move the window to the parent node.
10. *examine(w)*: if w is over an internal node return the value at that node, otherwise throw an exception.
11. *replace(e,w)*: if w is over an internal node replace the value with e and return the old value, otherwise throw an exception.
12. *delete(w)*: throw an exception if *w* is over an external node or an internal node with no external children, otherwise replace the node under w with its internal child if it has one, or an external node if it doesn't.

Alternatives for *child*. . .

1. *left(w)*: throw an exception if *w* is over an external node, otherwise move the window to the left (first) child.
2. *right(w)*: throw an exception if *w* is over an external node, otherwise move the window to the right (second) child.

— can be convenient for binary trees, but does not extend to (multiway) trees.

Note: as with the list ADT, the `Window` class can be replaced by a `treeIterator` to navigate and manipulate the tree.

Tree

Just modify Bintree to deal with more children (higher *branching*)...

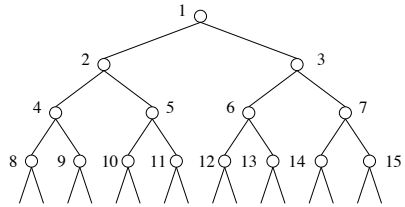
1. *degree(w)*: returns the degree of the node under w.
2. *child(i,w)*: throw an exception if *w* is over an external node or *i* is not in the range $1, \ldots, d$ where d is the degree of the node, otherwise move the window to the *i*-th child.

Orchard

Since an orchard is a list (or queue) of trees, an orchard can be specified simply using List (or Queue) and Tree (or Bintree)!

2. Block Representation of Bintree

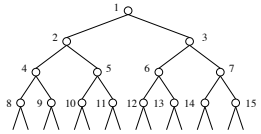
Based on an *infinite binary tree* — every internal node has two internal children...

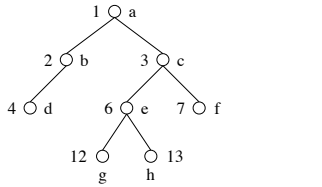


This is called a *level order* enumeration.

Every binary tree is a *prefix* of the infinite binary tree — can be obtained by pruning subtrees.

Example...





The size of block needed is determined by the height of tree.

Level-order representation is *implicit* — branches are not represented explicitly.

2.1 Time Performance

Level-order representation has the following properties:

1. $i(u) = 1$ iff u is the root.
2. Left child of u has index $2i(u)$.
3. Right child of u has index $2i(u) + 1$.
4. If u is not the root, then the parent of u has index $i(u)/2$ (where / is integer division).

These properties are important — allow constant time movement between nodes

$\Rightarrow$ all Bintree operations are constant time!

2.2 Space

Level-order representation can waste a great deal of space.

Q: What is the worst case for memory consumption?

Q: What is the best case for memory consumption?

A binary tree of size n may require a block of size $2^n - 1$

$\Rightarrow$ exponential increase in size!

3. Recursive Representations of Bintree

Basic Structure

Recall List:

- recursive definition
- recursive singly linked structure — one item, one successor

We can do the same with binary trees — difference is we now need *two* "successors".

Recall the (recursive) definition of a binary tree — can be briefly paraphrased as:

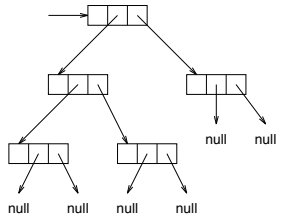
A binary tree either:

- is empty, or
- consists of a root node u and two binary trees $u(1)$ and $u(2)$. The function u is called the *index*.

It can be implemented as follows.

First, instead of `Link`, use a `TreeCell`...


```
public class TreeCell {

  public Object nodeValue;
  public TreeCell[] children;

  public TreeCell(Object v, TreeCell tree1, TreeCell tree2) {
    nodeValue = v;
    children = new TreeCell[2];
    children[0] = tree1;
    children[1] = tree2;
  }
}
```



The `children` array performs the role of the *index* u — it holds the "successors".

An alternative for binary trees is...

```
public class TreeCell {

  public Object nodeValue;
  public TreeCell left;
  public TreeCell right;

  public TreeCell(Object v, TreeCell tree1, TreeCell tree2) {
    nodeValue = v;
    left = tree1;
    right = tree2;
  }
}
```

but this doesn't extend well to trees in general. The previous version can easily be extended to multiway trees by initialising larger arrays of children.

Windows

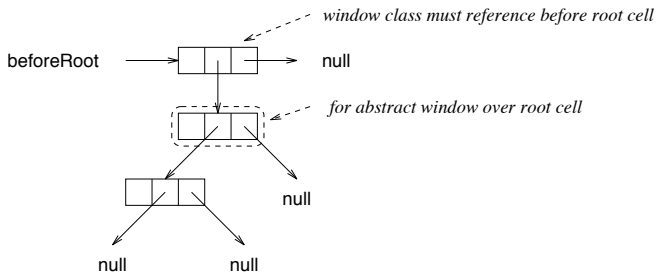
Just like lists, we wish to allow multiple windows for manipulating trees. We will therefore define a "companion" window class.

In the notes and exercises on lists, we considered a representation in which the window contained a member variable that referenced the cell previous to the (abstract) window position. This was so that *insertBefore* and *delete* could be implemented in constant time without moving data around.

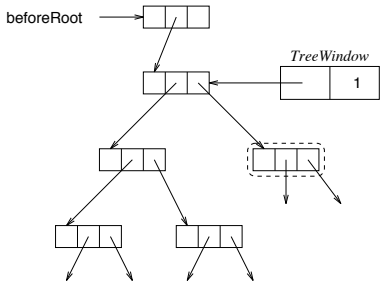
Similar problems arise in trees with *delete*, where we want to point the parent node to a different child.

We will use the same technique — the window class will store a reference to the *parent* of the (abstract) window node

$\Rightarrow$ requires a "before root" cell.



Since the parent has two children, we need to know which the window is over, so we include a branch number...



```
public class TreeWindow {
  public TreeCell parentnode;
  public int childnum;

  public TreeWindow () {}
}
```

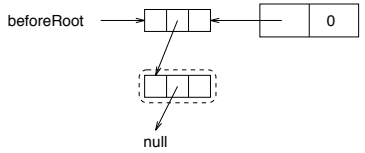
For example...

```
public void initialise(TreeWindow w) {
  w.parentnode = beforeRoot;
  w.childnum = 0;
}
```

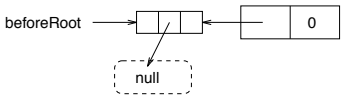

External Nodes

Two choices:

1. If values are attached to external nodes, the external nodes must be represented by cells. They can be distinguished from internal nodes by a null reference as the left child.



2. If external nodes have no values they can be represented simply by null references...



We will assume external nodes do *not* store values, and represent them by null references.

3.1 Examples

Constructor

```
public BintreeLinked () {
  beforeRoot = new TreeCell(null, null, null);
}
```

Checkers

```
public boolean isEmpty() {return beforeRoot.children[0] == null;}
```

```
public boolean isExternal(TreeWindow w) {
  return w.parentnode.children[w.childnum] == null;
}
```

```
public boolean isLeaf(TreeWindow w) {
  return !isExternal(w)
    && w.parentnode.children[w.childnum].children[0] == null
    && w.parentnode.children[w.childnum].children[1] == null;
}
```


Manipulators

Exercises...

```
public Object examine(TreeWindow w) throws OutOfBounds {
  if (!isExternal(w))

  else throw new OutOfBounds("examining external node");
}


public void insert(Object e, TreeWindow w) throws OutOfBounds {
  if (isExternal(w))

  else
    throw new OutOfBounds("inserting over internal node");
}
```


3.2 Performance

Clearly all operations except *parent* can be implemented to run in constant time.

parent in Bintree is like *previous* in List.

Can be achieved in a similar manner to link coupling — search the tree from the before-root node. Recall traversals from Topic 11!

Takes $O(n)$ time in worst case for binary tree of size n.

Q: What representation could we use to obtain a constant time implementation of *parent*?

3.3 Sbintree

Just like the simplist ADT, if a tree only requires one window, we can implement it using reference reversal!

Analogous to Simplist (although a bit more involved):

- implicit window
- constant time implementation of *parent*
- *initialise* is linear time, but constant time in the amortized case
- avoid stack memory for recursion during depth-first traversal

See Wood, Section 5.5.4.

4. Trees

Recursive representation can be extended to multiway trees — just increase the size of the `children` array...

```
public class TreeCell {
  public Object nodeValue;
  public TreeCell[] children;

  public TreeCell(int degree, Object v, TreeCell tree1,...) {
    nodeValue = v;
    children = new TreeCell[degree];
    children[0] = tree1;
    children[1] = tree2;
    .
    .
    .
  }

}
```


5. Summary

- Block representation of Bintree
  - time efficient — constant time in all operations
  - not space efficient — may waste nearly 2^n cells
- Recursive representation of Bintree
  - a generalisation of List
  - choices for window and external node representations
  - *parent* is linear time (traversal), all other operations are constant time
- Tree
  - generalisation of Bintree

CITS2200 Data Structures and Algorithms

Topic 13

Priority Queues

- The PQueue ADT
- A linked implementation
- Heaps
- A heap implementation of a priority queue
- Heapsort

Reading: Lambert and Osborne, Sections 8.8 and 12.1

1. Priority Queues

A priority queue is an extension of the queue ADT. In a priority queue however, when an item is added to the queue, it is assigned a *priority* (an integer) to indicate the relative importance of the item.

Instead of storing items in chronological order, a priority queue archives items with the highest priority before all others.

Items are no longer removed on a first-in-first-out basis — items are removed depending on their priority, with items of equal priority processed in chronological order.

Example uses include:

- scheduling services — eg distributing CPU time among several threads
- optimization algorithms — such as Prim's algorithms, Dijkstra's Algorithm, A*
- sorting

2. PQueue Specification

☐ **Constructors**

1. *PQueue()*: initialises an empty priority queue.

☐ **Checkers**

2. *isEmpty()*: returns *true* if the priority queue is empty.

☐ **Manipulators**

3. *enqueue(e, k)*: places *e* in the priority queue with key (or priority) *k*, or throws an exception if *k* is negative. The item is placed in front of all elements with lesser priority, but behind all others.
4. *examine()*: returns the element at the front of the queue, or throws an exception if the queue is empty.
5. *dequeue()*: removes the element at the front of the queue and returns it, or throws an exception if the queue is empty.

2.1 Example

```
PQueue p = new PQueue();    []

p.enqueue('a', 1);          [<a,1>]

p.enqueue('b', 3);          [<b,3>, <a,1>]

p.enqueue('c', 3);          [<b,3>, <c,3>, <a,1>]

p.enqueue('d', 2);          [<b,3>, <c,3>, <d,2>, <a,1>]

p.dequeue();                [<c,3>, <d,2>, <a,1>]
                            // b is returned.
```

3. Linked Implementation

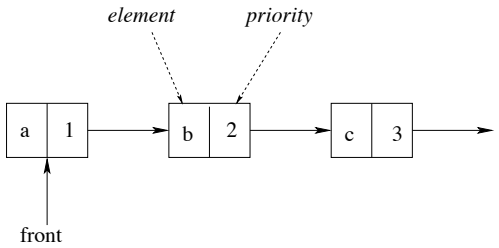
```
/**
 * A Linked Priority Queue for the generic type E
 **/
public class PQueueLinked<E> {

  // Only one link is required!
  private Link<E> front;

  // Constructor
  public PQueueLinked() {
    front = null;
  }
```


3.1 A Link Inner Class

```
/**
 * An inner class to hold the element, the successor,
 * and the priority
 **/
private class Link<E> {
  E element;
  int priority;
  Link<E> next;

  public Link(E e, int p, Link<E> n) {
    this.element = e;
    this.priority = p;
    this.next = n;
  }

}
```


3.2 isEmpty, examine, and dequeue

The process of checking to see if the priority queue is empty, examining the front element, or dequeuing the front element can all be done in the same way as for the queue ADT.




```
public boolean isEmpty() {return front == null;}

public E examine() throws Exception {
  if (!isEmpty()) {
    return (E) front.element;
  } else throw new Exception("Empty Queue");
}

public E dequeue() throws Exception {
  if (!isEmpty()) {
    E temp = (E) front.element;
    front = front.next;
    return temp;
  } else throw new Exception("Empty Queue");
}
```

3.3 Enqueuing

To enqueue, we start at the front of the queue and keep moving back until we find some element of lesser priority, or reach the end of the queue.

We then insert the new element in front of the lesser element.

```
public void enqueue(E e, int p) {
  if (isEmpty() || p > front.priority) {
    front = new Link<E>(e, p, front);
  } else {
    Link<E> l = front;
    while (l.next != null && l.next.priority >= p) {
      l = l.next;
    }
    l.next = new Link<E>(e, p, l.next);
  }
}
```


3.4 Performance

- *enqueue*: This operation is performed by iterating through the queue from the front to the back until the correct location to insert the new element is found. In the worst case, the entire queue will be examined $\Rightarrow$ $O(n)$ where n is the size of the queue.
- *examine*: This operation simply returns the element at the front of the queue $\Rightarrow$ $O(1)$.
- *dequeue*: This operation returns the element at the front of the queue and updates the value of front $\Rightarrow$ $O(1)$.

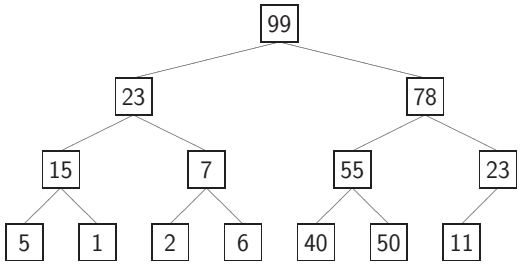
4. Heap Implementation

The *Heap* data structure is based on a binary tree, where each node of the tree contains an *element* and a *key* (an integer) — effectively the priority of the element.

A heap has the added property that the key associated with any node is greater than or equal to the key associated with either of its children.

$\Rightarrow$ the root of the binary tree has the largest key.

Note also that there is no requirement to order the left and right children of a node.



Just like the infinite binary tree, we *store* the heap as a linear array:

99	23	78	15	7	55	23	5	1	2	6	40	50	11

The bottom level of the binary tree, if not complete, is filled from the left.

4.1 Parents and Children

Suppose the array A is indexed from 1.

Then $A[1]$ holds the root of the binary tree, which by the heap property is, the element with the largest key value.

The two children of the root are stored in $A[2]$ and $A[3]$.

In general, the left child of node i is $2i$ and the right child is $2i + 1$.

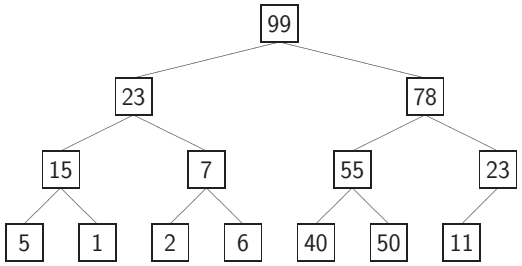
Conversely, the parent of node i is $\lfloor i/2 \rfloor$.

$\Rightarrow$ operations to determine both the parent and children of a node are $O(1)$.

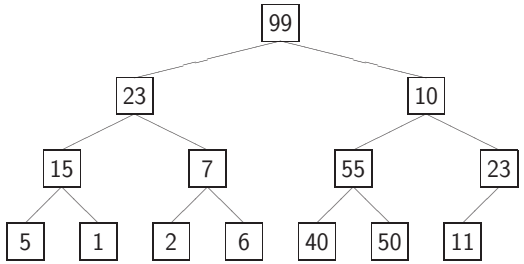
4.2 Heapify

Most of the operations we wish to perform on the heap will alter the data structure. Often these operations will destroy the heap property and it will have to be restored.

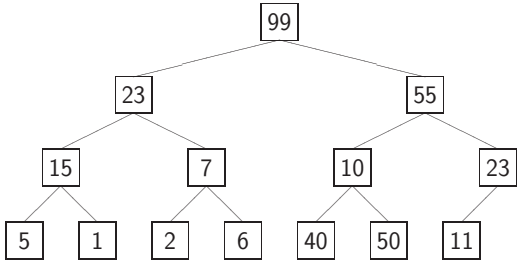
For example, suppose we decide to alter the value associated with one of the nodes — eg we wish to set $A[3] = 10$.



Making the change is trivial, but the resulting structure is no longer a heap — the entry $A[3]$ is no longer larger than both of its children.



We can rectify this by swapping $A[3]$ with the *larger* of its two children.



This means that the problem at $A[3]$ has been fixed — however, we *may* have introduced a problem at $A[6]$.

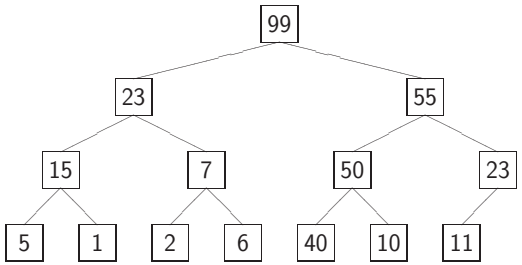
So we now examine $A[6]$ and if it is smaller than its children we perform another exchange.

We continue recursively checking the swapped child until we restore the heap property or reach the end of the tree.

The procedure outlined above, whereby a small element "percolates" down the tree is called *heapify*.

Heapify takes a position i in the tree as an argument and iterates down the tree from i, swapping entries if necessary until the heap property is restored.

Heapify assumes that the two children of i are proper heaps, but that the key value $A[i]$ may not be larger than the key values of its children.



4.3 Complexity of heapify

A balanced binary tree with n elements in it has a height of $\log n$ and hence *heapify* performs at most $\log n$ exchanges.

$\Rightarrow$ *heapify* is $O(\log n)$.

Consider now how all the operations necessary for a priority queue can be accomplished by using a heap together with *heapify*...

4.4 Performance

- *enqueue*: The key is entered at the end of the array (ie, in the last position in the tree). The resulting structure may not be a heap, because the value of the new key may be greater than its parent. If this is the case, exchange the two keys and proceed to examine the parent. In the worst case, $\log n$ exchanges will have to be done $\Rightarrow$ $O(\log n)$.
- *examine*: The root of the binary tree is the entry with the largest key value. Hence, merely return the root of the tree $\Rightarrow$ $O(1)$.
- *dequeue*: In this case, we must also delete the root node from the tree and then restore the heap property. This can be achieved by moving the final entry in the tree to the newly-vacated root position and then calling `heapify(1)` to restore the heap property. This involves a few constant time operations, together with one call to *heapify* $\Rightarrow$ $O(\log n)$.

4.5 Heapsort

The heap data structure allows us to implement a relatively efficient sorting procedure. Suppose we are given an unordered array containing a number of integers (or objects that can be completely ordered) that we wish to arrange from highest to lowest. Suppose there are n elements in the array.

Imagine enqueuing each of the elements into a priority queue with a priority equal to their value. This involves n operations at $O(\log n)$ each $\Rightarrow$ $O(n \log n)$.

We can then dequeue each element into an array; the elements being dequeued in sorted order. This involves n operations at $O(\log n)$ each $\Rightarrow$ $O(n \log n)$.

$\Rightarrow$ overall complexity is $O(n \log n)$, which is optimal for sorting using comparisons.

4.6 Pseudo-code for Heapsort

```
int[] heapSort(int[] arr) {
  // Create a priority queue
  PQueue p = new PQueue();
  // Add in the elements with themselves as the key
  for (int i : arr) {
    p.enqueue(i, i);
  }

  // Create a new array to store the result
  int[] ans = new int[arr.length];
  // Dequeue the elements from the priority queue into ans
  for (int i = 0; i < ans.length; i++) {
    ans[i] = p.dequeue();
  }
  return ans;
}
```


5. Performance Comparison

Operation	Linked	Heap
enqueue	n	$\log n$
examine	1	1
dequeue	1	$\log n$

6. Summary

- Priority queues behave like queues except elements are enqueued with a priority and are returned in order of their priority
- Linked representation
  - based on maintaining the elements in sorted priority order
  - constant time examination and dequeuing
  - linear time enqueuing
- Heap representation
  - based on maintaining elements in a heap: elements are stored in a binary tree with the added property that all nodes have a greater than or equal value to both of its children.
  - constant time examination
  - logarithmic time enqueuing and dequeuing

CITS2200 Data Structures and Algorithms

Topic 14

Sets, Tables, and Dictionaries

- Set specification
- Set representations — characteristic function, lists, ordered lists
- Table specification
- Table representations
- Dictionary specifications
- Dictionary representations — set-based representations, search trees

Reading: Lambert and Osborne, Sections 12.2 and 13.1
Goodrich and Tamassia, Chapter 10

1. Introduction

In this section, we examine three ADTs: *sets*, *tables*, and *dictionaries*, used to store collections of elements with no repetitions.

Note that these names are used (eg in different texts) for a range of similar ADTs — we define them as follows:

Set

- used when set-theoretic operations are required
- elements may or may not be ordered
- includes "membership" operations: *isEmpty*, *insert*, *delete*, *isMember*
- includes "set-theoretic" operations: *union*, *intersection*, *difference*, *size*, *complement*

Table

- simpler version of Set without the set-theoretic operations
- elements assumed to be unordered

Dictionary

- like Table but assumes elements are totally ordered
- includes "order related" operations: *isPredecessor*, *isSuccessor*, *predecessor*, *successor*, *range*

1.1 Elements, Records, and Keys

Elements may be a single items, or "records" with unique *keys* (such as those typically found in databases).

We will usually talk about elements as if they are single items.

eg. "if $e_1 < e_2$ then..."

In the case of record elements, this can be considered shorthand for

"if $k_1 < k_2$, where k_1 is the key of record e_1 and k_2 is the key of record e_2, then..."

1.2 Examples

The following are examples of situations where the ADTs might be used:

Set

"I have one set of students who do CITS2200 and one set of students who do CITS2210. What is the set of students who do both?"

Table

"I begin with the set of students originally enrolled in CITS2200. These two students joined. This one withdrew. Is a particular student currently enrolled?"

Dictionary

"Here is the set of students enrolled in CITS2200 ordered by (exact) age. Which are the students between the ages of 18 and 20?"

2. Set Specification

□ **Constructors**

1. *Set()*: create an empty set.

□ **Checkers**

2. *isEmpty()*: returns *true* if the set is empty, *false* otherwise.
3. *isMember(e)*: returns *true* if *e* is a member of the set, *false* otherwise.

□ **Manipulators**

4. *size()*: returns the cardinality of (number of elements in) the set.
5. *complement()*: returns the complement of the set (only defined for finite universes).
6. *insert(e)*: forms the union of the set with the singleton $\{e\}$
7. *delete(e)*: removes e from the set
8. *union(t)*: returns the union of the set with t.
9. *intersection(t)*: returns the intersection of the set with *t*.
10. *difference(t)*: returns the set obtained by removing any items that appear in t.
11. *enumerate()*: returns the "next" element of the set. Successive calls to *enumerate* should return successive elements until the set is exhausted.

3. Set Representations

3.1 Characteristic Function Representation

Assume A is a set from some universe U.

The *characteristic function* of A is defined by:

$$f(e) = \begin{cases} true \ (\text{or } 1) & e \in A \\ false \ (\text{or } 0) & \text{otherwise} \end{cases}$$

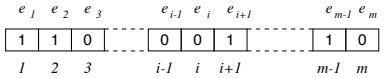
$\Rightarrow$ thus a set can be viewed as a boolean function.

If U is finite and '$\leq$' is a total order on U, the elements of U can be enumerated as the sequence

$$e_1, \ldots, e_m$$

where $e_i \leq e_j$ if $i < j$, and m is the cardinality of U.

The characteristic function maps this sequence to a sequence of 1s and 0s. Thus the set can be represented as a block of 1s and 0s, or a *bit vector*...



Sometimes called a *bitset* — eg. `java.util.BitSet`

Advantage

Translates set operations into efficient bit operations:

- *insert* — *or* the appropriate bit with 1
- *delete* — *and* the appropriate bit with 0
- *isMember* — is the (boolean) value of the appropriate bit
- *complement* — complement of a bit vector
- *union* — *or* two bit vectors
- *intersection* — *and* two bit vectors
- *difference* — *complement* and *intersection*

Also *enumerate* — can cycle through the m positions reporting 1s.

Performance

- *insert*, *delete*, *isMember* — constant providing index can be calculated in constant time
- *complement*, *union*, *intersection*, *difference* — $O(m)$; linear in size of *universe*
- *enumerate* — $O(m)$ for n calls, where n is size of set

  $\Rightarrow$ $O(\frac{m}{n})$ amortized over n calls

Disadvantages

- If the universe is large compared to the size of sets then:
  - the latter operations are expensive
  - large amount of space wasted
- Requires the universe to be bounded, totally ordered, and known in advance.

3.2 List Representation

An alternative is to represent the set as a list using one of the List representations. Here, we assume there is no total ordering on the elements.

Performance

Assume we have a set of size p.

insert, *delete*, *isMember* — take $O(p)$ time; the best that can be achieved in an unordered list (recall `eSearch`)

union — for each item in the first set, check if it is a member of the second, and if not, add it (to the result)

$\Rightarrow$ $O(pq)$ where p and q are the sizes of the two sets

Other set operations (*intersection*, *difference*) behave similarly.

Note that if both sets grow at the same rate (the worst case), the time performance is $O(p^2)$.

Inefficient because one list must be traversed for each element in the other. Can we traverse both at the same time...?

3.3 Ordered List Representation

If the universe is totally ordered, we can obtain more efficient implementations by merging the two in sorted order.

Assume A can be enumerated as $a_1, a_2, \ldots, a_p$ and B can be enumerated as $b_1, b_2, \ldots, b_q$.

Eg. *union*

$i = 1$; $j = 1$;
do {
  if (a_i == b_j) add a_i to C and increment i and j;
  else add smaller of a_i and b_j to C and increment its index;
}
while (i <= p && j <= q);
add any remaining a_i's or $b_j's$ to C

Exercise

Give pseudo-code for the *intersection* and *difference* operations.

Performance

Each list is traversed once $\Rightarrow$ $O(p+q)$ time.

This is much better than $O(pq)$.

If p and q grow at the same rate (worst case), the time performance is now $O(p)$.

Note also that *isMember* is now $O(\log p)$ (recall `bSearch`)

4. Table Specification

The Table operations are a subset of the Set operations:

□ **Constructors**

1. *Table()*: create an empty table.

□ **Checkers**

2. *isEmpty()*: returns *true* if the table is empty, *false* otherwise.
3. *isMember(e)*: returns *true* if *e* is in the table, *false* otherwise.

□ **Manipulators**

4. *insert(e)*: forms the union of the table with the singleton $\{e\}$
5. *delete(e)*: removes e from the table

5. Table Representations

Since the Table operations are a subset of those of Set, the (unordered) List representations can be used.

insert, *delete*, *isMember* therefore take $O(p)$ time.

The more efficient List representations and the characteristic function representation are not available since the elements are assumed to be unordered.

The operations can be made more efficient by considering the probability distribution for accesses over the list and moving more probable (or more frequently accessed) items to the front — see Wood, Section 8.3.

Later, we'll look in detail at a more efficient representation of tables using *hashing*, where such operations are close to constant time.

6. Dictionary Specification

□ **Constructors**

1. *Dictionary()*: creates an empty dictionary.

□ **Checkers**

2. *isEmpty()*: returns *true* if the dictionary is empty, *false* otherwise.
3. *isMember(e)*: returns *true* if *e* is a member of the dictionary, *false* otherwise.
4. *isPredecessor(e)*: returns *true* if there is an element in the dictionary that precedes *e* in the total order, *false* otherwise.
5. *isSuccessor(e)*: returns *true* if there is an element in the dictionary that succeeds *e* in the total order, *false* otherwise.

□ **Manipulators**

6. *insert(e)*: adds e (if not already present) to the dictionary in the appropriate position.
7. *predecessor(e)*: returns the largest element in the dictionary that is smaller than *e* if one exists, otherwise throws an exception.
8. *successor(e)*: returns the smallest element in the dictionary that is larger than *e* if one exists, otherwise throws an exception.
9. *range(p,s)*: returns the dictionary of all elements that lie between *p* and *s* (including *p* and *s* if present) in the ordering.
10. *delete(e)*: removes item e from the dictionary if it exists.

7. Dictionary Representations

7.1 Representations Based on Set

We have already seen two representations that can be used for Sets when there is a total ordering on the universe...

- characteristic function (bit vector) representation
  - time efficiency (eg $O(1)$ for *isMember*) gained by indexing directly to appropriate bits
  - bounded — universe fixed in advance
  - space wasted if universe is large compared with commonly occurring sets

- List based (ordered block) representation
  - time efficiency (eg $O(\log n)$ for *isMember*) comes from binary search
  - bounded
  - space usage may be poor if large block is set aside

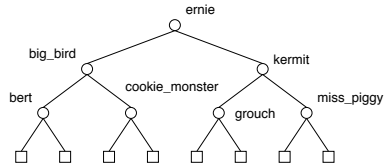
We now examine a representation which supports a binary-like search but is unbounded...

7.2 Binary Search Trees

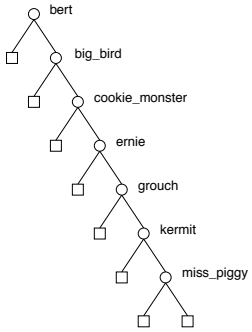
A *binary search tree* is a binary tree whose internal nodes are labelled with elements (or their keys) such that they satisfy the *binary search tree condition*:

> For every internal node u, all nodes in u's left subtree precede u in the ordering and all nodes in u's right subtree succeed u in the ordering.

eg.



eg.



Searching

If information is stored in a binary search tree, a simple recursive "divide and conquer" algorithm can be used to find elements:

```
if (t.isEmpty()) terminate unsuccessfully;
else {
  r becomes the element on the root node of t;
  if (e equals r) terminate successfully;
  else if (e < r) repeat search on left subtree;
  else repeat search on right subtree;
}
```

Performance

Depends on the shape of the tree...

Exercise

- Best case is a perfect binary tree. What is the performance of *isMember*?
- Worst case is a skinny binary tree. What is the performance of *isMember*?

insert and delete

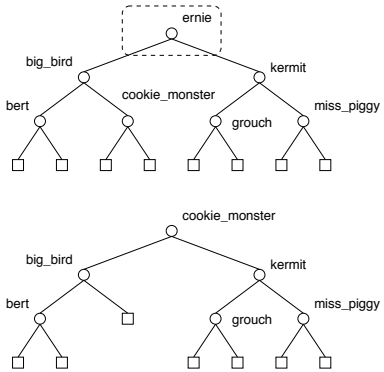
insert is fairly straightforward

- perform a search for the element as above
- if the element is found, take no further action
- if an empty node is reached, insert a new node containing the element

delete is straightforward if the element is found on a node with at least one external child — just use the standard Bintree *delete* operation

Otherwise:

1. replace the deleted element with its predecessor — note that the predecessor will always have an empty right child
2. delete the predecessor



Balancing Trees

Note that the *delete* procedure described here has a tendency over time to skew the tree to the right — as we have seen this will make it less efficient.

Alternative: alternate between replacing with predecessor and successor.

In general, it is beneficial to try to keep the tree as "balanced" or "complete" as possible to maintain search efficiency.

There are a number of data structures that are designed to keep trees balanced — *B-trees*, *AVL-trees* and *Red-black trees*.

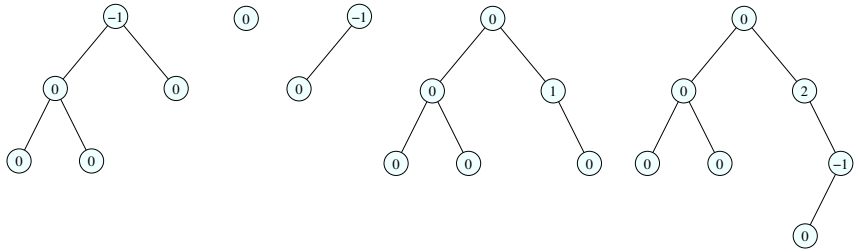
Note: Java.util.TreeMap uses Red-Black trees in its implementation. In general, for search trees with guaranteed logarithmic performance, it is better to use this class than to write your own.

7.3 AVL Trees

We will look at the three types of *self-balancing* trees, but we will only examine the operations of AVL trees. (The other trees just involve more complicated versions of these operations).

An *AVL tree* is a binary search tree where, for every node, the height of the left and right subtrees differ by at most one. This means the depth of any external node is no more than twice the depth of any other internal node.

The picture below demonstrates some AVL-trees. Nodes are marked with the height of the right sub-tree minus the height of the left sub-tree.



AVL trees have $O(\log n)$ time peformance for searching, inserting, and deleting.

AVL Tree Operations

Since an AVL tree *is* a binary search tree, the searching algorithm is exactly the same as for a binary tree.

However, the insertion and deletion operations must be modified to maintain the balance of the tree.

AVL Tree Insertions

We first find the appropriate place (a leaf node) to add the new element. If the insertion makes the tree unbalanced, then we locally reorganize the tree (via a *rotation*) to restore balance.

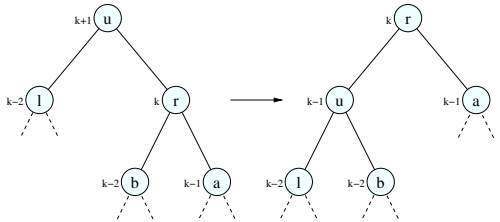
To insert an element, we:

1. Insert the element into an external node (as per usual for a binary search tree).
2. Starting with its parent, check each ancestor of the new node to ensure the left and right subtree heights differ by less than two.
3. If we find a node such that one child has height $k-2$ and the other has height k, then we perform a rotation to restore balance.

Rotation Case I

Suppose u is a node where its left child l has height two less than its right child r, and the *right* child of r has height one more than the *left* child of r.

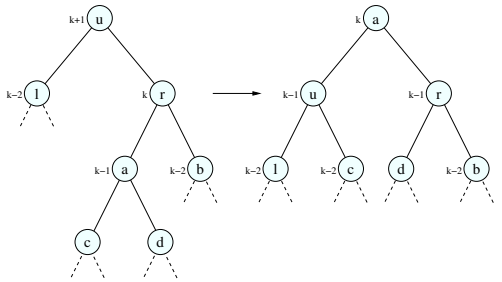
We rearrange the tree as follows:



Rotation Case II

Suppose u is a node where its left child l has height two less than its right child r, and the *left* child of r has height one more than the *right* child of r.

We rearrange the tree as follows:



AVL Tree Deletions

Note that both rotations do not increase the height of the sub-tree, so insertion only needs to be done at the lowest unbalanced ancestor.

To delete an element, we:

- Delete the element (as per usual for a binary search tree).
- Starting with its parent, check each ancestor of the new node to make sure it's balanced.
- If any node is not balanced, perform the necessary rotation (as above).
- Continue to check the ancestors of the deleted node up to the root.

Complexity

Rotations are constant time operations.

Insertions and deletions involve searching the tree for the element ($O(h)$, where h is the height of the tree) and then checking every ancestor of that element ($O(h)$ in the worst case).

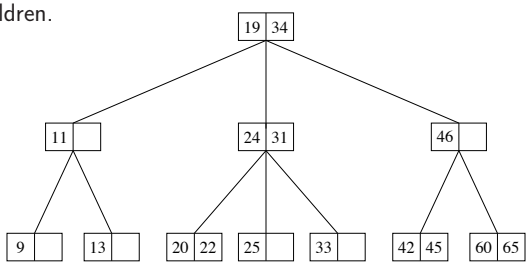
Complexity follows from the claim: *the height of an AVL tree is less than* $2 \log n$ *where* n *is the number of elements in the tree.*

7.4 B-trees

A *B-tree* is a tree data structure that allows searching, insertion, and deletion in amortized logarithmic time.

Each node of a B-tree can contain multiple items and multiple children (these numbers can be varied to tweak performance).

We will consider 2-3 B-trees, where each node can contain up to two items and up to three children.



If there is just one item in the node, then the B-Tree is organised as a binary search tree: all items in the left sub-tree must be less than the item in the node, and all items in the right sub-tree must be greater.

It there are two elements in the node, then:

- all items in the left sub-tree must be less than the smallest item in the node
- all items in the middle sub-tree must be between the two items in the node
- all elements in the right sub-tree must be greater than the largest item in the node

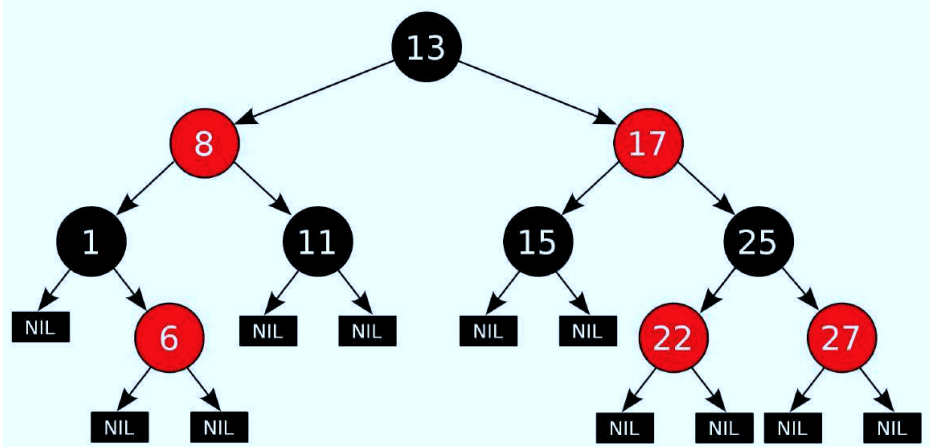
Also, every non-leaf node must have at least two successors and all leaf nodes must be at the same level.

7.5 Red-black Trees

A *red-black tree* is another variation of a binary search tree that is slightly more efficient (and complicated) than B-trees and AVL trees.

A red-black tree is a binary tree where each node is coloured either red or black such that the colouring satisfies:

- *the root property* — the root is black
- *the external property* — every external node is black
- *the internal property* — the children of a red node are black
- *the depth property* — every external node has the same number of black ancestors.



Picture courtesy of Wikimedia Commons

8. Summary

We have outlined 3 ADTs for use with collections of unique elements or records:

- Set — includes set-theoretic operations, elements may or may not be ordered
- Table — restriction of Set with fewer operations, elements assumed not ordered
- Dictionary — extension of Table, assumes ordering and contains "order-related" operations

We have covered a number of representations:

- List — can be used for unordered sets and tables
- ordered list (block) — can improve efficiency for ordered sets and can be used for bounded dictionaries
- characteristic function — can be very efficient for ordered sets and dictionaries in certain cases, bounded
- binary search tree — unbounded representation for dictionaries, efficiency better than List but depends on tree shape

Next — efficient representations for Tables...

CITS2200 Data Structures and Algorithms

Topic 15

Hash Tables

- Introduction to hashing — basic ideas
- Hash functions
  - properties, 2-universal functions, hashing non-integers
- Collision resolution
  - bucketing and separate chaining
  - open addressing
  - dynamic tables — linear hashing

Reading: Lambert and Osbourne, Section 13.2

1. Introduction

The Table is one of the most commonly used data structures — central to databases and related information systems.

In the previous section, we briefly examined a List representation for tables, but this had linear time access.

Can we do better?

We have seen a number of situations where constant time access to data can be achieved by indexing directly into a block...

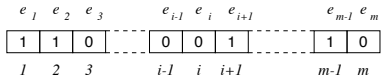
eg. Array uses an addressing function

$$\alpha(i,j) = (i-1) \times n + j \qquad 1 \le i \le m,\ 1 \le j \le n$$

	a	b	c	d	e
a	1	2	3	4	5
b	6	7	8	9	10
c	11	12	13	14	15
d	16	17	18	19	20

eg. Set uses a characteristic function...

$$f(e) = \begin{cases} true \ \ (\text{or } 1) & e \in A \\ false \ \ (\text{or } 0) & \text{otherwise} \end{cases}$$



These approaches:

- often sacrifice space for time — space wasted by all the "holes"
- rely on the ordering of elements, which translates to an ordering of the memory block.

Can we improve on these?

- more compact use of space
- applicable to unordered information (eg Table)

⇒ *hashing...*

2. Basic Hashing

The direct indexing approaches above work by:

- setting aside a big enough block for all possible data items
- spacing these so that the address of any item can be found by a simple calculation from its ordinality

What if we use a block which is not big enough for all possible items?

- Addressing function must map all items into this space.
- Some items may get mapped to the same position $\Rightarrow$ called a *collision*.

Example

Suppose we fill out our Lotto coupons as follows. Each time we notice a positive integer in our travels, we calculate its remainder modulo 45 and add that to our coupon...

The first thing we see is a pizza brochure containing the numbers 165, 93898500, 2, 13, 1690. These map to positions 30, 15, 2, 13, 25.

This fills 5 positions in our data store...

0	1	X	3	4	5	6	7	8
9	10	11	12	X	14	X	16	17
18	19	20	21	22	23	24	X	26
27	28	29	X	31	32	33	34	35
36	37	38	39	40	41	42	43	44

Next we ring up to order our Greek Vegetarian, and we're told it'll be ready in 15!
$\Rightarrow$ *collision*

Collisions

When two entries "hash" to the same key, we have a *collision*.

We need a method for dealing with this. However...

Advantages:

- Once we allow collisions we have much more freedom in choosing an addressing function.
- It no longer matters whether we know an ordering over the items.

Exercise

Obtain an address from your name into the block 1...10 as follows:

- Count up the number of letters in your name.
- Add 1.
- Double it.
- Add 1.
- Double it.
- Subtract the number of letters in your name.
- Add the digits in your current number together.
- Square it.
- Add the digits in your number together.

What address did you get?

Such a function is called a *hash function*. It takes the item to be looked up or stored and "hashes it" into an address in the block, or *hash table*.

Hashing is used in both data management (via hash tables) and security and cryptography (for example MD5 checksums).

We will consider hash functions in more detail, and then consider methods for dealing with collisions.

3. Hash Functions

To begin with, we'll assume that the element (or key) to be hashed is an integer. We need a function

$$h : \mathcal{N} \to 0 \ldots m-1$$

that maps it into a hash table `block` of size m.

Thus, to store a table t, we would set

$$\texttt{block}[h(a)] = a$$

for all elements a in t, and fill the other elements of `block` with null references.

We call $h(a)$ the *home address* of a.

3.1 Properties of Hash Functions

A hash function that maps each item to a unique position is called *perfect*. Note that if we had an infinitely large storage block we could always design a perfect hash function.

Since our hash functions will generally not be perfect, we want a function that distributes evenly over the hash table — that is, one that is not *biased*.

Q: What is an example of a "worst case" hash function in terms of bias?

3.2 2-universal Functions

In the Lotto example earlier, we used the hash function

$$h(i) = \ \ i \bmod 45.$$

A commonly used class of hash functions, called *2-universal* functions, extends this idea...

A *2-universal* hash function has the form

$$h(i) = ((c_1 i + c_2) \bmod p) \bmod m$$

where m is the size of the hash table, $p > m$ is a large prime ($p > 2^{20}$), $c_1 < p$ is a positive integer, and $c_2 < p$ is a non-negative integer.

— $(c_1 i + c_2) \bmod p$: "scrambles" i

— $\bmod m$: maps into the block

Small changes (eg to c_1) lead to completely different hash functions.

Another 2-universal hash function that can be used in languages with bit-string operations...

Assume items are b-bit strings for some $b > 0$, and $m = 2^l$ for some $l > 0$. The *multiplicative hash function* has the form:

$$h(i) = (a\, i \bmod 2^b) \operatorname{div} 2^{b-1}$$

for odd a such that $0 < a \leq 2^b - 1$.

Advantage — *mod* and *div* operations can be evaluated by shifting rather than integer division $\Rightarrow$ very quick

3.3 Hashing Non-integers

Non-integers are generally mapped (hashed!) to integers before applying the hash functions mentioned earlier.

Example

Assume we have a program with 3 variables

```
float abc, abd, bad;
```

and we wish to hash to a location to store their values. We could obtain an integer from:

- the length of each word (as in the earlier example) — will lead to a lot of collisions
  $\Rightarrow$ in this case all variables will hash to the same location
- the ordinality of the first character of each word — fewer collisions, but still poor
  $\Rightarrow$ the first two will hash to the same location

- summing the ordinality of all characters — likely to be even fewer collisions
  $\Rightarrow$ but here, the last two will collide
- "weight" the characters differently, eg 3 times the first plus two times the second plus one times the third — collisions will be much rarer (but may still occur)
  $\Rightarrow$ no collisions in this set

Extending the weighting idea, a typical hash function for strings is to treat the characters as digits of an integer to some base b.

Assume we have a character string $s_1 s_2 \ldots s_k$. Then, we calculate

$$[ord(s_1).b^{k-1} + ord(s_2).b^{k-2} + \ldots + ord(s_k)b^0] \bmod 2^B$$

Here:

- b is a small odd number, such as 37...
- $\bmod 2^B$ gives the least significant B bits of the result — eg 16 or 32.

Q: Why the *least* significant bits?

4. Collision Resolution Techniques

There are many variations on collision resolution techniques. We consider examples of three common types.

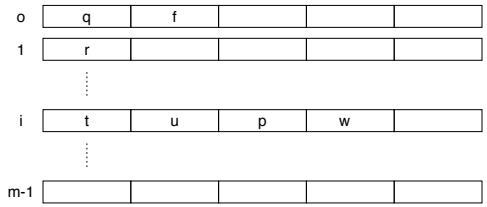
4.1 Bucketing and Separate Chaining

The simplest solution to the collision problem is to allow more than one item to be stored at each position in the hash table

$\Rightarrow$ associate a List with each hash table cell...

Bucketing

— each list is represented by a (fixed size) block.



"Advantage"

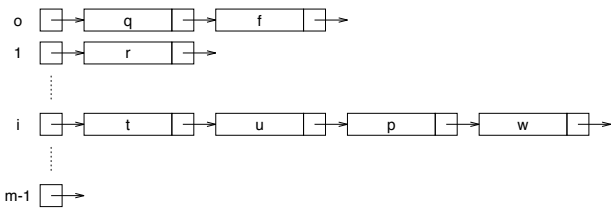
- Simple to implement — hash to address then search list.

Disadvantages

- Searching the List slows down Table access.
- Fixed size $\Rightarrow$ may waste a lot of space (both in hash table and buckets).
- Buckets may *overflow!* $\Rightarrow$ back where we started (a collision is just an overflow with a bucket size of 1).

Separate Chaining (Variable Size Bucketing)

— each List is represented by linked list or *chain*.



Advantages

- Simple to implement.
- No overflow.

Disadvantages

- Searching the List slows down Table access.
- Extra space for pointers (if we are storing records of information the space used by pointers will generally be small compared to the total space used).
- Performance deteriorates as chain lengths increase.

Performance

Worst case for separate chaining $\Rightarrow$ all items stored in a single chain. Worst case performance same as List: $O(n)$ — nothing gained!

But *expected case* performance is much better...

The *load factor* λ of a hash table is the number of items in the table divided by the size m of the table.

> *Assume that each entry in a hash table is equally likely to be accessed, and that each sequence of n insertions is equally likely to occur. Then a hash table that uses separate chaining and has load factor λ has the following expected case performance:*
>
> - $s(\lambda) = 2 + \lambda/2$ *probes (read accesses) for successful search*
> - $u(\lambda) = 2 + \lambda$ *probes for unsuccessful search*

See Wood, Section 9.2.

4.2 Open Addressing

Separate chaining (and bucketing) require additional space. Yet there will normally be space in the table that is wasted.

Alternative $\Rightarrow$ *open addressing* methods

- store all items in the hash table
- deal with collisions by incrementing hash table index, with wrap-around

Linear probing — increment hash index by one (with wrap-around) until the item, or *null*, is found.

Problem — items tend to "cluster".

Double hashing — increment hash index using an "increment hash function"! $\Rightarrow$ may jump to anywhere in table.

Advantages

- All space in the hash table can be used.

Disadvantages

- Insertions limited by size of table.
- Deletions are problematic...

Deleting items means others may not be able to be reached — requires reorganizing table, or marking (flagging) items as deleted.

The latter is most common, but means erosion of space in the hash table.

4.3 Dynamic Tables — Linear Hashing

Finally, there are methods that consider the hash table to be dynamic rather than static!

Linear hashing is an extension of separate chaining — rather than allowing the variable-length buckets (chains) to grow indefinitely, we limit the average size of the buckets.

- Insertions: if the average chain size exceeds a predefined upper bound, split the "next" unsplit bucket, and hash the items in the bucket by a function with double the previous base (ie m, $2m$, $4m$,...).
- Deletions: if the average bucket size drops below a predefined minimum and the table is no smaller than the original table, shrink the table.

Effectively, we use two hash functions parameterised by $split$ (the split point) and M (roughly the table size). Assume h is the hash function that maps keys to large integers.

Then, the hashed address is:

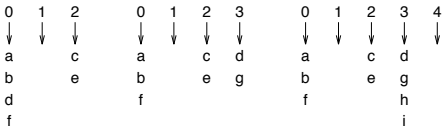
$$H(k) = \left\{ \begin{array}{ll} h(k) \bmod 2M, & \text{if } h(k) \bmod M < split \\ h(k) \bmod M, & \text{otherwise} \end{array} \right\}$$

If the loading factor becomes too high (eg more than $0.8M$ elements in the table), then:

1. $split = split + 1$
2. rehash all items in bucket $split - 1$
3. if $split = M$, set $split = 0$ and $M = 2M$.

Example

Assume table is initially of size 3, and maximum loading is 2...



Disadvantages

- (More complicated to code.)
- Requires movement of items.

Advantages

- Maximum load is maintained — improves expected efficiency.
- Insertions — no overflow, not bounded by size of table.
- Deletions — no erosion.

This approach is used by `java.util.Hashtable` — have a look at its API documentation.

5. Summary

Hash tables can be used to:

- improve the space requirements of some ADTs for which bounded representations are suitable
- improve the time efficiency of some ADTs, such as Table, which require unbounded representations

We have seen a number of methods for collision resolution in hash tables:

- bucketing and separate chaining
- open addressing, including linear probing and double hashing
- dynamic methods, such as linear hashing

6. Caution!

Note that while performance can be very good, *this is not a panacea!* For many applications, such as those naturally represented by trees, hashing would lose the structure.

ie. *don't "hash first, ask questions later"*

(Hash can be addictive!)

CITS2200 Data Structures and Algorithms

Topic 16

Revision

Dr. Luigi Barone
Room 2.12
luigi@csse.uwa.edu.au

Dr. Rowan Davies
Room 2.16
rowan@csse.uwa.edu.au

A unit about *space*, *time*, and *integrity*.

Handbook Description

At the core of most computer applications is the storage and retrieval of information. The way that the stored data is structured has a strong impact on *what can be retrieved*, *how quickly it can be retrieved*, and *how much space it occupies*. The use of generic structures, or abstract data types (ADTs), to encapsulate the data also allows *software engineering principles* of independent modification, extension and reuse.

This unit studies the *specification, implementations and time and space performance of a range of commonly-used ADTs* and corresponding algorithms in an object-oriented setting. The aim is to provide students with the background needed both to implement their own ADTs where necessary, and to select and use appropriate ADTs from object-oriented libraries where suitable.

1. What We Studied

This unit presented a dearth of information about commonly used data structures and a framework for analysing and comparing them.

1.1 Preliminaries

- What is a data structure?
- What is an abstract data type?
- What is the difference between an abstract data type and a data structure?
- Why study ADTs and data structures.

1.2 Recursion

- Recursive algorithms (mathematical functions).
- Recursive data structures (linked lists, linked trees).
- Implementing recursion in Java.
- Recursion in binary searches and tree traversals.

1.3 Data Abstraction And Specification Of ADTs

- Be aware of the difference between an ADT (a specification) and a data structure (an implementation of an ADT).
- A programmer should identify the required specification, and then search for the most appropriate implementation.
- How does Java support the separation of specifications and implementations?

1.4 Complexity Analysis

- What makes a good implementation, and how do we compare different implementations?
- Complexity analysis is used to measure time and space performance of data structures via mathematical functions parameterised by the size of input.
- Expected case analysis makes assumptions about the type of input and calculates the average time/space taken.
- Worst case analysis makes the most pessimistic assumptions to obtain a guaranteed lower bound of performance.
- Asymptotic analysis abstracts away all minor terms from the functions and focuses on the *rates of growth*.
- Amortized analysis is used in data structures to amortize the cost of expensive operations over the cheaper operations that must accompany them.

1.5 Programming

- As an outcome of this unit, you are expected to be a very competent programmer.
- You should be aware of how Java supports ADTs through encapsulation, abstraction, interfaces, and generics.
- You should be aware of common methods to protect the integrity of data structures using access modifiers, inner classes, and iterators.

If you have successfully completed the requirements of this unit, you should consider yourself a competent programmer. You may now wish to consider:

- Participating in the ACM-ICPC programming competition.
- Looking for work experience to fulfill Professional Practicum requirements.

2. The ADTs And Data Structures

You should be aware of the implementations, complexity, and applications for:

ADT	**Implementation**
Stack	Block, Linked
Queue	Block, Linked, Cyclic
Deque	Block, Cyclic
List	Singly linked, Doubly linked, Cyclic, Block, Simplist
Maps	Linked, Block
Binary Trees	Singly linked, Block, Doubly linked, SBinTree
Trees	Singly linked, Doubly Linked
Priority Queue	Linked, Heap
Sets	Block with characteristic function, Linked (ordered and unordered)
Table	Block, Linked
Dictionary	List, Binary search tree, B-Tree, AVL tree, Red-black tree
Hash Table	Bucketing, Separate chaining, Open addressing, Linear hashing

3. The Exam

The structure of the exam:

- 2 hours 10 minutes.
- 15 multiple choice, each worth 2 marks $\rightarrow$ 30 marks.
- 3 short answer questions, each worth 20 marks $\rightarrow$ 60 marks.
- 90 marks total $\rightarrow$ 1.5 minutes per mark.

More information is available from the CITS2200 web-site.

All the material covered in the lectures (with the exception of any material marked as not examinable), tutorials, labsheets, and the project is examinable, whether it was for assessment or not.

Email us if you would like to arrange a consultation before the exam.

Good luck!