CITS2200
Data Structures and Algorithms

Cara MacNish
School of Computer Science & Software Engineering
University of Western Australia

Topic 1

Introduction

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

Reading: Lambert & Osborne, Secs. 1.1–1.6

1.1 Why are we here?

100 students, 2 hours of lectures, 3 hours of labs/pracs,...

- 500 person hours
- 62 person days
- $> \frac{1}{4}$ person year each week!

- clarity
- correctness
- efficiency
- maintainability
- reusability

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
  — eg. torpedo's self-destruct mechanism
- clients are unpredictable!

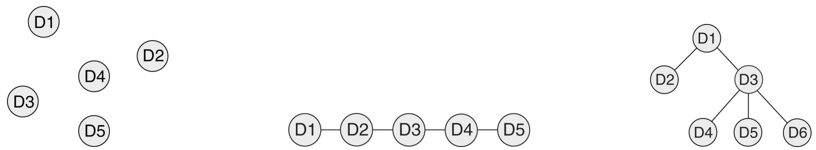
1.2 What will we study?

1.2.1 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)

1.2.2 Abstract Data Types

Allow user to abstract away from implementation detail

Example: Microwave Oven

• cook for 3 mins	vs	• turn on light • begin rotating plate • start timer • nuke everything ...

Some have a higher level of abstraction
— eg. "cook white rice"

Example: Picture Framing

Need to cut many pieces of wood at 45 degree angles

Program 1

1. `measure the distance of the mark where the cut is needed from the end of the piece of wood`
2. `measure the same distance on the other side of the wood`
3. `measure the distance between the two marks`
4. `add the same distance to the mark on the other side`
5. `draw a 45 degree line between the marks`
6. `align a guide block with the line`
7. `cut against the guide block`

Alternative solution...

Mitre Saw

Implementation:

eg. wooden box pre-cut with common angles, and saw

$\Rightarrow$ "knows" how to cut commonly used angles

Operations:

1. cut at 90 degrees
2. cut at 45 degrees

   ⋮

Program 2

1. `place wood in box aligning mark with 45 degree angle`
2. `cut wood`

We want to build and use "mitre saws" for our collections

— details of implementation abstracted away from user

We call these abstract data types (ADTs)

1.2.3 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

1.3 More on ADTs

1.3.1 History

The evolution of programming $\cdots \triangleright$

machine code

$\downarrow$

assembler

$\downarrow$

high-level languages (basic, fortran, ...)

$\swarrow$ $\searrow$

declarative programming (logic, functional, ...)

procedural programming (pascal, C, ...)

$\downarrow$

$\boxed{\text{object-oriented programming}}$

$\downarrow$

component-oriented programming ...?

agent-oriented programming ...?

$\downarrow$ abstraction away from machine “nuts and bolts” — towards humans?

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

— for a good introduction see the Eiffel book:

Object-Oriented Software Construction, by Bertrand Meyer

(a.k.a. the "OO bible")

1.3.2 ADTs and Java

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
  .
  .
```

1.3.3 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?

1.3.4 Advantages of ADTs

- modularity — independent development, re-use, portability, maintainability, upgrading, etc
- delay decisions about final implementation
- separate concerns of program 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

1.4 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 — bounded vs unbounded etc

We will cover a range of important, commonly used data structures. For example:

- stacks
- queues
- lists
- maps
- arrays
- trees
- sets, tables and dictionaries
- hash tables

1.4.1 Structure of the Course

- Introduction
- Java concepts
- Examples of abstraction
- ADT specification
- Review of recursion and recursive data structures
- Examples of ADTs — Queues and Stacks
- Performance analysis for data structures
- Widely used data structures $\Rightarrow$ all your favourites!

Topic 2

Java Primer

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

Reading: Lambert & Osborne, App. A & Sec. 1.2, 2.1–2.7

2.1 Review of Java Basics

2.1.1 Primitive Data Types

```
byte short int long
float double
char
boolean
```

2.1.2 Local Variables

Scope: block in which defined

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

Result?

2.1.3 Expressions

Built from variables, values, and operators.

arithmetic: +, −, ∗, /, %,...

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

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

2.1.4 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 and do-while

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

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

for

```
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());
```

?

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];
```

2.1.5 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?

2.2 Primitive Types vs Reference Types

Primitive types

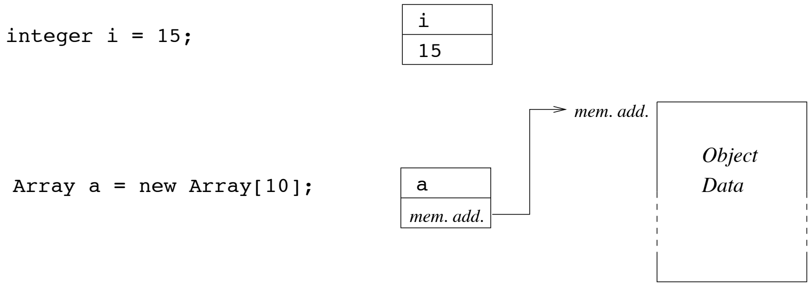
- fixed size
- size doesn't change with reassignment

⇒ 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



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

⇒ reference types

2.2.1 Assignment

Primitive Type

```
int i = 7;
int j = i;
j++;
System.out.println(i);
```

?

Reference Type

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

?

Parameter Passing

Primitive Type

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

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

?

Reference Type

```
// 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]);
```

?

2.2.2 Equality

Primitive Type

```
int i=7;
int j=7;
System.out.println(i==j);
```

?

Reference Type

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

?

2.3 Classes and Objects

2.3.1 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 w * l * h;}
}
```

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);
```

2.3.2 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

2.3.3 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.

2.3.4 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

2.3.5 The `static` keyword

Used for methods and variables in classes that don't create objects.

Example:

```
public class MatrixTest {

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

Called class variables and class methods.

Also used for "constants".

Example:

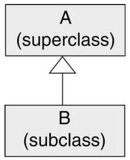
```
public class Matrix {

  static final int MAX_SIZE=100;

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

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

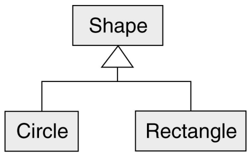
We will use `static` rarely in this unit.

2.4 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 & Osborne, Sec. 2.5.)

```
public class Circle extends Shape {

  private double radius;

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

We will not be building hierarchies extensively in this unit. However:

- You will see them in the text.
- You will see them in the Java API. Especially in the Java Collections classes.
- We will be using some very important features...

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();
```

2.4.1 Wrappers

There is one thing our `arrayOfAnythings` can't hold: primitives!

Since primitives are not classes, they aren't subclasses of `Object`.

Example:

```
Object holdsAnything;
holdsAnything = 42;
```

Compilation:

```
javac  Test.java
Test.java:11: incompatible types
found   : int
required: java.lang.Object
    holdsAnything = 42;
                    ^
1 error
```

Solution

"Wrap" primitives inside an object. . .

We could write our own "wrapper classes":

Example:

```
public class myInteger {
  private int theInt;
  public myInteger (int i) {theInt = i;}
  public int get () {return theInt;}
}
```

Now we can have:

```
Object holdsAnything;
holdsAnything = new myInteger(42);
```

But it is unnecessary: Java provides wrappers for all primitives:

⇒ `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.

2.4.2 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
```

2.5 Interfaces

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)
- no `public/private` necessary — they are implicitly `public`

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

More examples — see the Java API

eg. the Collection interface

2.6 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)

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

We can also create our own exception classes (by subclassing Java's exceptions).

However a full treatment of exceptions is not part of this unit.

The main use of exceptions in this unit will be for checking preconditions.

Reading: Lambert & Osborne, Sec. 1.12

Topic 3

Recursive Data Structures and Linked Lists

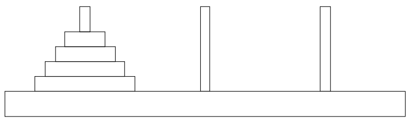
- Review of recursion: mathematical functions
- Recursive data structures: lists
- Implementing linked lists in Java
- Java and pointers
- Trees

Reading: L & O, Sections 10.1, 5.3–5.4

3.1 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 smaller disk on a larger 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!

3.1.1 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));
}
```

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?

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` $\Rightarrow$ see Exercises

Recursion is:

- powerful — can solve arbitrarily large problems
- concise — code doesn't increase in size with problem
- closely linked to very important proof technique called mathematical induction
- basis of logic programming and functional programming (logic program to solve 'Towers of Hanoi' takes just two lines!)

- 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

3.2 Recursive Data Structures

Recursive programs usually operate on recursive data structures

⇒ data structure defined in terms of itself

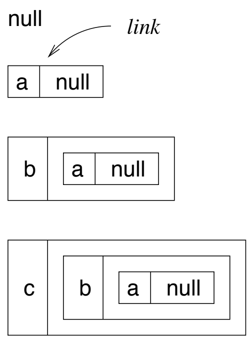
3.2.1 Lists

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

Building a list...



3.3 A LinkedList Class in Java

3.3.1 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 constructor makes a new link from an item and an existing link.

3.3.2 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...

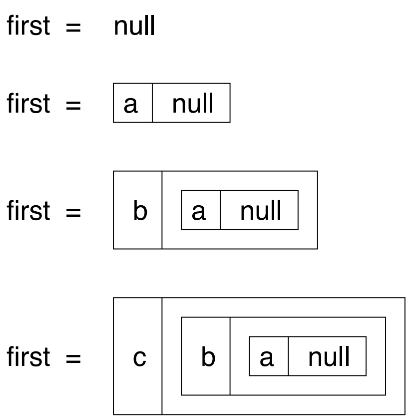
- **Constructing an empty list**

```
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;                    // Its part of my 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.
                  // In this case it occurs (or is ''thrown'')
                  // if the user tries to examine an empty list.

  /**
   * 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;
  }
}
```

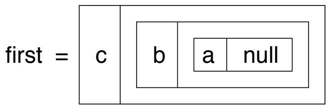
3.4 Java and Pointers

Conceptually, the successor of a list is a list.

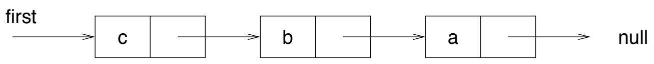
One of the great things about Java (and other suitable object oriented languages) is that the program closely reflects this "theoretical" concept — from a programmer's point-of-view the successor of a `LinkChar` is a `LinkChar`.

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.

3.4.1 Freedom from Pointers

While Java uses references or pointers internally, the programmer is freed from the task of having to manipulate them. This is in contrast to many traditional languages (eg Pascal, C, C++) where pointers must be explicitly handled by the programmer.

Example: Pascal

```
type linktype = ^celltype;
     celltype = record
                  item: char;
                  successor: linktype
                end;

     First = linktype;
```

A procedure to insert an item looks like:

```
procedure insert(c: char; var l: First);
var p: linktype;
begin
  new(p);
  p^.item := c;
  p^.successor := l;
  l := p;
end;
```

Compare this to:

```
void insert (char c) {first = new Link(c, first);}
```

Java allows us to abstract away from the details.

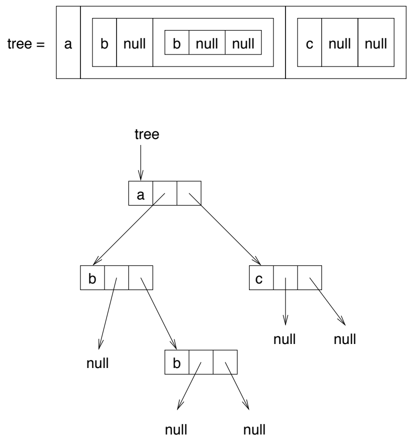
3.5 Trees

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

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

[Some examples of trees — see Wood p142]

Graphical representations...



More on trees later.

3.6 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 courses

⇒ You need to understand them. If not, seek help!

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

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

4.1 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

4.2 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 thats 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!

4.3 Data abstraction

The above program integrates:

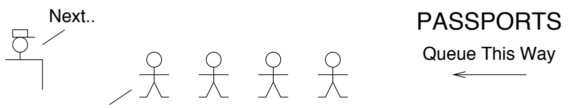
- data, and instructions to access it
- “higher-level” role of the program

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:

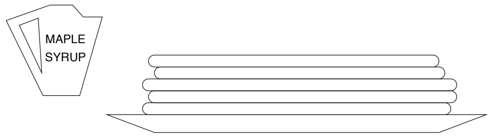
1. Create an empty queue
2. Test whether the queue is empty
3. Add a new latest element
4. Examine the earliest element
5. Delete the earliest element

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!

4.4 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

Note: No variable in the argument list corresponds to the object itself (the queue). This is because the methods are instance methods — whenever they are called they will "belong" to a particular object.

eg.

```
Queue q = new Queue();
System.out.println(q.isEmpty());
```

In data structure texts for non-object-oriented languages such as Pascal, you will find an extra argument in the specification of operations.

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.

4.5 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 $\cdots \triangleright$

```
// 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 (`char`s). This can be seen in the argument to `enqueue` and the return types of `examine` and `dequeue`.

In this course (particularly in the Labs) we will specify data structures using interfaces, and in most cases consider a number of alternative implementations.

(We'll look at different implementations of the QueueChar interface later.)

4.6 `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 course 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.

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;
}
```

Notes:

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

4.7 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 thats 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

4.8 Summary

- When programming we should look for abstractions of the data — could we use a generic data structure (ADT) rather than "reimplement 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...

Topic 5

Queues

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

Reading: Lambert & Osborne, Sect. 8.1–8.4.

5.1 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.

5.2 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 of `char`s.

5.2.1 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

5.3 Interface

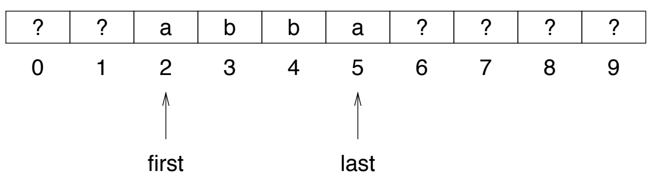
```
import Exceptions.*;
                                // Character queue interface.
public interface QueueChar {    // some javadoc comments omitted

  /**
   * 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;
```

5.4 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

5.4.1 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.

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

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

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

5.4.2 Modifiers

enqueue, examine and dequeue are straightforward. . .

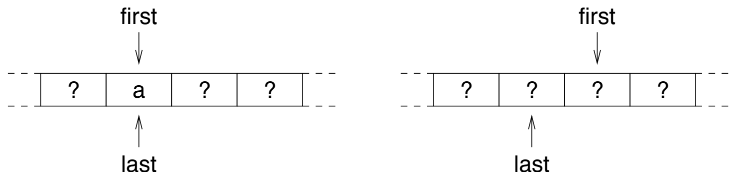
```
/**
 * add a new item to the queue
 * @param a the item to add
 * @exception Overflow if queue is full
 */
public void enqueue (char a) throws Overflow {
  if (!isFull()) {
    last++;
    items[last] = a;
  }
  else throw new Overflow("enqueuing to full queue");
}
```

```
/**
 * 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");
}
```

5.4.3 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;}
```

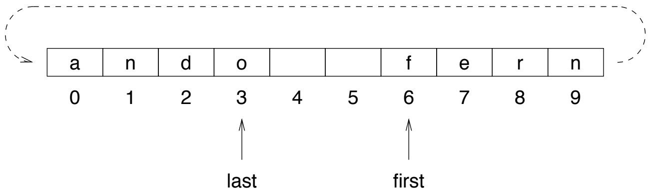
Notes

- `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 $n - 1$.

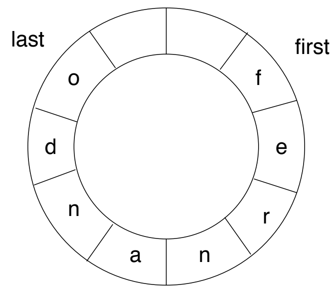
5.4.4 Alternative block implementations

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");
}
```

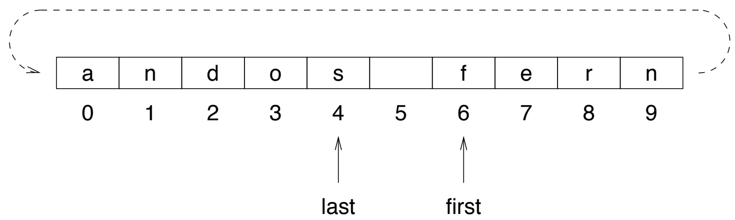
- A queue is now empty when:

  `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.

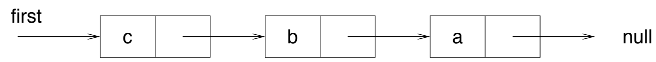
$\Rightarrow$ Exercises!

5.5 Recursive (Linked) Representation

Biggest problem with block representation — predefined queue length

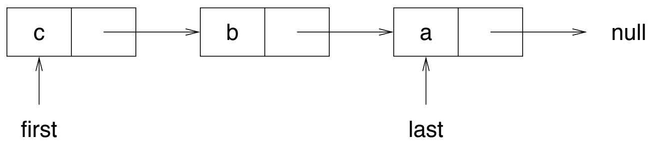
Solution: use a recursive structure!

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...



5.5.1 Class Declaration

```
import Exceptions.*;

/**
 * Linked list representation of a queue of characters.
 * The queue is unbounded.
 */
public class QueueCharLinked implements QueueChar {

  /**
   * reference to the front of the queue, or null if
   * the queue is empty
   */
  private LinkChar first;
```

```
  /**
   * reference to the back of the queue, or null if
   * the queue is empty
   */
  private LinkChar last;

}
```

5.5.2 Constructors and Checkers

Empty queue:

first	$\longrightarrow$	null
last	$\longrightarrow$	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;}
```

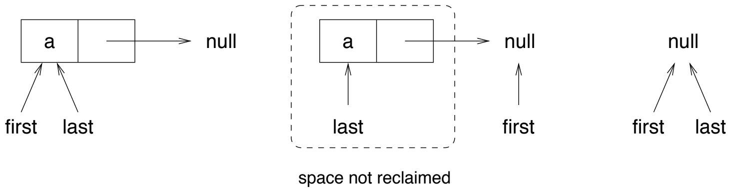
5.5.3 Examining and Dequeueing

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");
}
```

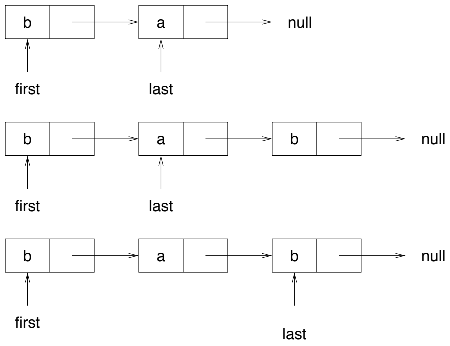
Dequeueing is the same as deleting in the linked list, except that when the last item is dequeued, `last` must be assigned null. . .




```
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");
}
```

5.5.4 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;
  }
}
```

5.6 Summary

We have seen a number of alternative representations for the Queue ADT

- 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

Next — efficiency comparisons...

Topic 6

Performance Analysis 1: Introduction

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

Reading: Lambert and Osborne, Section 4.1.

6.1 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

6.2 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

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

⇒ 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

6.3 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.

Simulational Measurement

Construct a (computer) model of system and evaluate performance with simulated data.

eg. US nuclear weapons defence system

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 (?)

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.

6.4 Example: A Basic Analysis of the Queue ADTs

As an example of comparison of ADT performance we consider two representations of queues — block (without wraparound) and recursive — 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

6.4.1 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

6.4.2 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

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

8 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 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 units vs 3 units) — given the assumptions made these are not very informative.

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

6.5 Growth Rates

Q: If it takes 2 hours to roast a turkey, how long does it take to cook a mammoth?

A: Need to make some assumptions...

Chef 1: Turkey 10kg, Mammoth 1000kg

Temporal-calorific-multiplier $c = \frac{2}{10} = 0.2$ hrs/kg

Cooking time $t = 0.2 \times 1000 = 200$ hrs

Chef 2: Turkey 10kg, Mammoth 1000kg

A little more distance for the heat to penetrate takes a lot more heat. In fact each centimetre of radius takes 6 times as long as the previous one. Radius increases with approx cube root of volume, so cook time increases with square of volume. Volume is proportional to mass. Therefore cook time increases with square of mass.

Temporal-calorific-multiplier $c = \frac{2}{10^2}$ hrs/kg^2

Cooking time $t = \frac{2}{100} \times 1000^2 = 20\,000$ hrs

Chef 3: Turkey 10kg, Mammoth 1000kg

Heat is hanging around in the rest of the unused oven space anyway, although for bigger animals you need a bigger oven, which is slower to heat up. Doubling the mass only adds a constant amount to the cooking time.

Temporal-calorific-multiplier $c = \frac{2}{\log 10}$

Cooking time $t = \frac{2}{\log 10} \times \log(1000) = 6$ hrs

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 abstract measure of time
- ignore actual storage space occupied by data elements and consider number of items stored as abstract measure of space

6.6 Summary

Three types of performance measurement — empirical, simulational, 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, comparing small differences, we will concentrate on a coarser (but more robust) analysis in terms of rates of growth.

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.

7.1 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.

7.2 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...

7.2.1 Worst Case Analysis

Choose data which have the largest time/space requirements.

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 course.

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

7.2.2 Expected Case Analysis

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

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

Advantages

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

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...

7.2.3 Amortized Case Analysis (or "Encouraging Long-termism in Forestry")

Suppose that each day my company can perform one of two operations:

1. plant a tree
2. cut down n trees

Greenpeace will give me \$1 for each tree we plant.

Chop-n-Mulchit Woodchippers will give me \$1 for each tree we cut down.

Clearly we can make n times as much money (for the same number of days) by chopping down trees as we can by growing them!

— if d is the number of days, we make nd chopping, and only d growing.

Whereas the return from "growing day" operations for a fixed period of days is constant, the return from "chopping day" ops appears to be linear in n

— bigger n, more money!

But what if trees are a finite resource — say we start with an empty paddock?

Over time we can't cut down more trees than we grow. For each "cutting day" operation we need n "growing day" operations. Averaged out over these $n+1$ operations, our return per day is

$$\frac{(n+n)\,dollars}{(n+1)\,days} = \frac{(2n)\,dollars}{(n+1)\,days} \approx \$2/day$$

That is, the "average" return per operation (day) is constant!

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

In this case the "big picture" shows we can't make as much money as the "small picture" suggests.

Moral: Companies relying on natural resources need to look at the amortized analysis!

In terms of more familiar data structures, a similar example for a Multidelete Stack (adapted from Wood)...

Create a new ADT MStack (multidelete stack) from Stack by replacing the operation pop() with mPop(i) which removes i elements from the top of the stack.

What is the performance of mPop on:

1. a block implementation?
2. a linked list implementation?

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!

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

7.3 Asymptotic Growth Rates

We have talked about comparing data structure implementations — can use any of empirical, simulational or analytical.

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)

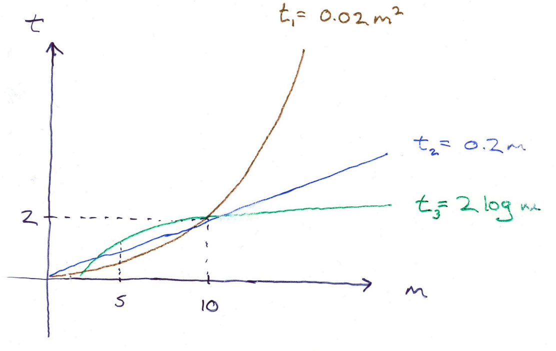
In this section we formalise the ideas of analytical comparison and growth rates.

7.3.1 Why Asymptopia

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

- Evaluation may be misleading.

  Recall cooking a woolly mamoth...



Assume $t_1 = 0.002m^2$, $t_2 = 0.2m$, $t_3 = 2\log m$.

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.

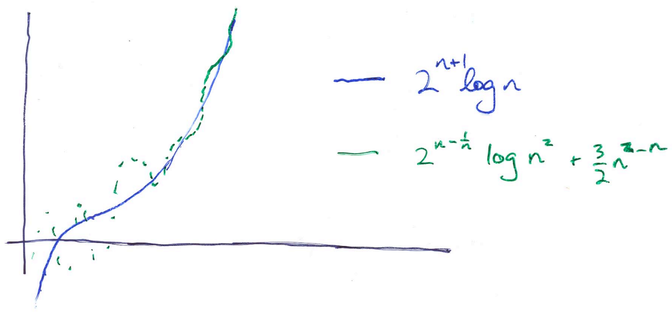
- Want a closed form.

  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

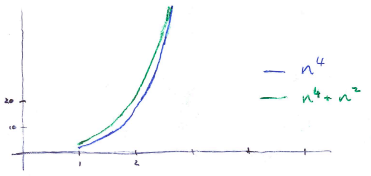
Investigate what simple function the more complex one tends to or asymptotically approaches as the argument approaches infinity, ie in the limit.

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	$\frac{n^2}{n^4+n^2}$
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%



7.3.2 Comparison "in the Limit"

How well does one function approximate another?

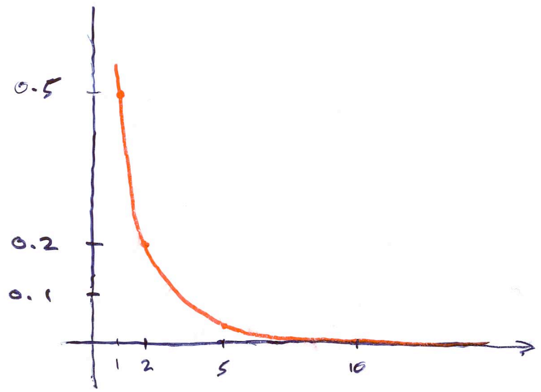
Compare growth rates. Two basic comparisons...

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

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

eg.

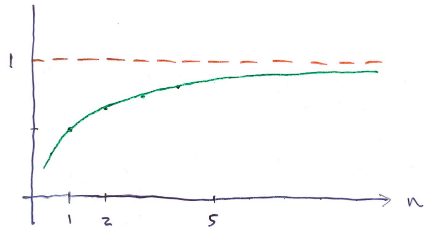
$$\frac{n^2}{n^4+n^2}$$



2.
$$\frac{f(n)}{g(n)} \longrightarrow 1 \quad \text{as} \quad n \longrightarrow \infty$$

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

eg.
$$\frac{n}{n+1}$$



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

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

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

7.3.3 '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. $\square$

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 $m > \dfrac{c}{n_0}$.

Thus the initial assumption is incorrect, and n^3 is not $O(n^2)$.

$\square$

From these examples we can start to see that big O analysis focusses 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.

7.3.4 '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 $O(f(n))$.

7.4 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 the DSA course. (See Discrete Structures.)

However some can be solved by common sense!

Example:

What is the time complexity of the recursive addition program from Section 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

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

For any value of c such that $c > b$ there exists $n_0 > 0$ such that $T(n) \leq cn$ for any $n > n_0$.

ie $T(n)$ is $O(n)$ $\Rightarrow$ linear in size of the input `y`

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

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

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.

7.5 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

Topic 8

Objects and Iterators

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

Reading: Lambert & Osborne, Sections 6.3–6.5; 2.3.5

8.1 Generalising ADTs to use Objects

Our ADTs so far have stored primitive types.

eg. block implementation of a queue from Section 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 Section 2.4

— inheritance, casting, and wrappers.

8.1.1 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");
  }
```

8.1.2 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 Section 2.4 that Java provides wrapper classes for all primitive types.

Autoboxing — Note for Java 1.5

Java 1.5 provides autoboxing and auto-unboxing. Effectively acts as automatic wrapping and unwrapping.

```
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/

8.1.3 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
```

Generics — Note for Java 1.5

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.

Generics in Java are complex and are the subject of considerable debate.

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
```

8.2 Iterators

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

Example:

In Lab Exercise 4 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 Chapter 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`.

These are:

- `boolean hasNext()` — return `true` if the iterator has more items
- `Object 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).

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`.

8.2.1 Using an Iterator

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

8.2.2 Implemention — 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);
  }
  ...
```

8.2.3 Implemention — iterator

```
import java.util.Iterator;

class BasicQueueIterator implements Iterator {
  private Queue backingQ;
  private int current;

  BasicQueueIterator(Queue 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.");
  }
}
```

8.2.4 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 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 iterator...

```
class BasicQueueIterator implements Iterator {
  private Queue 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];
      currentIndex = (current+1)%backingQ.items.length;
      return temp;
    }
  }
```

8.3 Inner Classes

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

Later versions of Java provide a tidier way... inner classes.

Inner classes are declared within a class:

```
public class MyClass {

  // fields

  // methods

  private class MyInnerClass {

    // fields

    // methods
  }

  ...
}
```

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() {   // only constructed in outer class
      current = first;               // variable accessed directly
    }                                // no passing of backing queue

    public boolean hasNext () {
      return !isEmpty() &&                           // methods & variables
        ((last >= first && current <= last) ||       // accessed directly
         (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?