Data structures cover …

- General issues in data structure design;
- One-dimensional and multi-dimensional arrays;
- Linked lists, doubly-linked lists and operations on these data structures;
- Stacks and operations on stacks;
- Queues and operations on queues;
- Maps and operations on maps;
- Trees & Graphs.

Data Structures and Algorithms Lecture 1

Overview and Guidelines on Quality Software Design

Data structures

- A data structure is a systematic way of organising a collection of data for efficient access
- Every data structure needs a variety of algorithms for processing the data in it
- Algorithms for insertion, deletion, retrieval, etc.
- So, it is natural to study data structures in terms of
  - **properties**
  - **organisation**
  - **operations**
- This can be done in a programming language independent way. (why?)
- Has been done in Pseudo code, Python, C, C++, Java, etc.

Motivation

- Why do you write programs?
- Why would you write quality software?
- How to write quality software?
- Why data structure is related to quality software design?

Why data structures?

Aren't primitive types, like boolean, integers and strings, and simple arrays enough?

- Yes, since the memory model of a computer is simply an array of integers
- But, this model . . .
  - is conceptually inadequate & low level, since information is usually expressed in the form of highly structured data
  - makes it difficult to describe complex algorithms, since the basic operations are too primitive

Menu

- Data structures
- Motivation of studying data structures
- Language to study data structures
- Abstraction
- Huffman coding & priority queues
- Information hiding
- Encapsulation
- Efficiency in space & time
- Static vs dynamic data structures

Topics

- In the forthcoming slides we first go though the main *principles that help developing good software*: **abstraction, information hiding, encapsulation.**
- We then talk briefly about how data structure design helps improving the quality of the final system.

- Finally we will address another important design issue related to data structures that help producing good software: the choice between **static** and **dynamic structures.**

Why not just in Java?

Why do we study data structures in a language independent way?

- Java is just one of many languages within the category of object-oriented languages
- The world's most favourite programming language changes about every five to ten years
- However, data structures have been around since the invention of high-level programming languages

Therefore,

- We must be able to realise data structures in other languages
- We also need the abstract context to study algorithms

Abstraction (1)

- We can talk about abstraction either as a process or as an entity.
- As a process, abstraction denotes the extracting of the essential details about an item, or a group of items, while ignoring the nonessential details.
- As an entity, abstraction denotes a model, a view, or some representation for an actual item which leaves out some of the details of the item.

- Abstraction dictates that some information is more important than other information, but does not provide a specific mechanism for handling the unimportant information.

Data structures that we will consider

We will study various data structures such as

- Arrays
- Lists
- Stacks
- Queues
- Maps
- Trees
- Graphs

In each case we will define the structure, give examples, and show how the structure is implemented in Java or pseudo code either directly or via other, previously defined, data structures.

Abstraction (2)

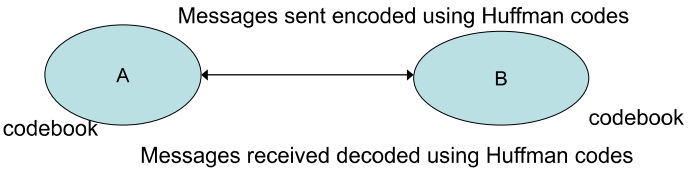
- In the context of software development, we can distinguish different kinds of abstractions:

- The aim of ***data* abstraction** is to identify which details of how data is stored and can be manipulated are important and which are not
- The aim of ***procedural* abstraction** is to identify which details of how a task is accomplished are important and which are not

Software quality

- Before digging in the various data structures that will be the main topic of this module it is important to ask the question:
- Why are we doing this?
- Whenever we develop a software system we should strive to create good software.
- To achieve this a number of design principles could be followed.
- Furthermore, the quality of the eventual software can be measured in several ways:
  - correctness,
  - efficiency.
- **A careful design of the data structures used in software system helps in designing good software.**

Communication based on Huffman coding



Key of abstraction…

- Extracting the **commonality** of components and hiding their details
- Abstraction typically focuses on the **outside view** of an object/concept

Huffman coding - Abstraction example

- Huffman coding is an effective way of encoding (and decoding) textual (or non-textual) data. It has been used in communication, e.g. source coding
- A large information system may need a piece of software that carries out the Huffman encoding of the data stored on a disk or generated from some data source.
- The Huffman encoding software uses priority queue to accomplish its tasks.
- The detailed description of such a module is given later.
- Important points:
  - The description abstracts from all details on how the priority queue and its operations are implemented.
  - Nonetheless the description enables a programmer to focus on the design of the particular module using the priority queue functions given.

Abstraction examples

- Looking at a map, we draw roads and highways, forests, not individual trees
- Looking at various bank accounts, what commonality can we extract?
  - Using an O-O approach
    - States
      - *AccountNo*
      - *CustomerName*
      - *Amount*
    - Behaviour
      - *Credit*, *Debit*, and *GetAmount*

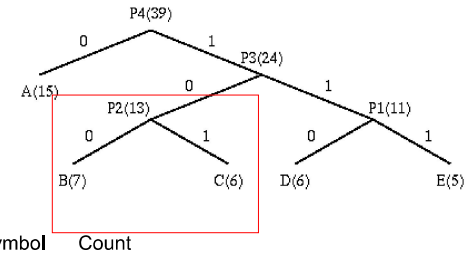
Huffman coding – Key ideas

- Based on the frequency of occurrence of a data item (pixel in images, alphabet in texts).
- The principle is to use a smaller number of bits to encode the data that occurs more frequently (why doing this?)
- Codes are stored in a ***Code Book*** which may be constructed for each image (alphabet) or a set of images
- In all cases the code book plus encoded data must be transmitted to enable decoding.

Use of abstraction in design

- Abstraction in design: break things into groups and figure out the details for each separately
- Abstraction leads to a top-down approach..
- Most projects can be improved with abstraction:
  - Think of the high-level. What do you want to accomplish? There should be one goal
  - Refine this goal into parts (components)
  - Think of multiple ways to implement each component

Constructing the en-/de-coding tree



Symbol	Count
A	15
B	7
C	6
D	6
E	5

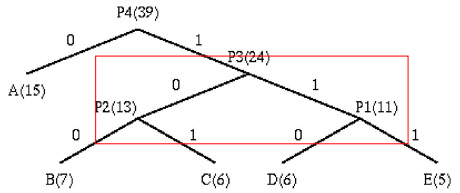
List: A(15),**B(7),C(6),**P1(11) → List: A(15),P1(11),*P2(13)*

More on Huffman coding – code book (code table)

Symbol	Count	Code	Symbol (Subtotal - # of bits)
A	15	0	A(15)
B	7	100	B(21)
C	6	101	C(18)
D	6	110	D(18)
E	5	111	E(15)
		TOTAL (# of bits): 87	

Q: How do we generate Huffman codes?

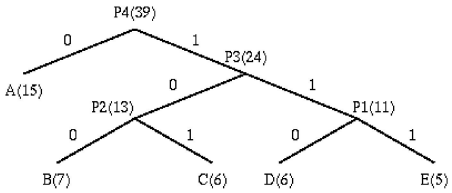
Constructing the en-/de-coding tree



Symbol	Count
A	15
B	7
C	6
D	6
E	5

List: A(15),**P1(11),P2(13)** → List: A(15),*P3(24)*

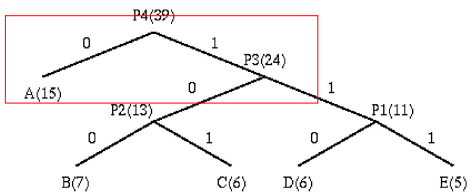
More on Huffman coding – en-/de-coding tree



Symbol	Count	Code	Subtotal (# of bits)
A	15	0	15
B	7	100	21
C	6	101	18
D	6	110	18
E	5	111	15
		TOTAL (# of bits): 87	

List: A(15),B(7),C(6),D(6),E(5)

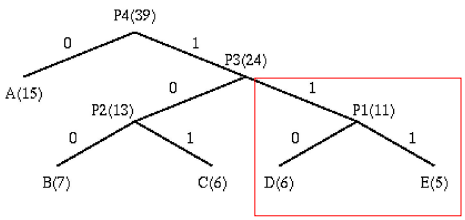
Constructing the en-/de-coding tree & table



Symbol	Count	Code	Subtotal (# of bits)
A	15	0	15
B	7	100	21
C	6	101	18
D	6	110	18
E	5	111	15
		TOTAL (# of bits): 87	

List: **A(15),P3(24)** → List: *P4(39)* END

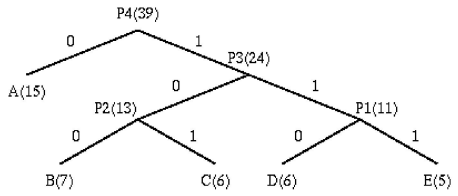
Constructing the en-/de-coding tree



Symbol	Count
A	15
B	7
C	6
D	6
E	5

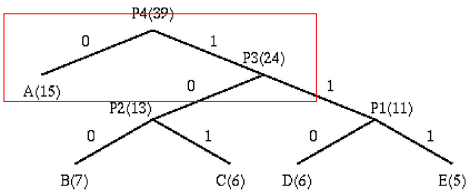
List: A(15),B(7),C(6),**D(6),E(5)** → List: A(15),B(7),C(6),*P1(11)*

Optimality of Huffman coding



Symbol	Count	Code	Subtotal (# of bits)
A	15	0	15
B	7	100	21
C	6	101	18
D	6	110	18
E	5	111	15
		TOTAL (# of bits):	87

Huffman decoding



Codebook

A	0
B	100
C	101
D	110
E	111

What are the input symbols if the following is received?

(1) 011011100
(2) 0110111001
(3) 1001110110

(1) 0,110,111,0,0 → ADEAA
(2) 0,110,111,0,0,1 → ADEAA + error
(3) 100,111,0,110 → BEAD

Use of abstraction in the design of Huffman coding

- Components
  - Encoding
    - Algorithms?
    - Data structures?
  - Decoding
    - Algorithms?
    - Data structures?
  - Codebook management
    - Algorithms?
    - Data structures?

Exercise

- Given the following symbol frequency table, design the corresponding Huffman coding scheme for it.

Symbol	Count	Code	Subtotal (# of bits)
A	1		
B	3		
C	5		
D	7		
E	11		
			TOTAL (# of bits):

- Using the codebook developed, decode:
  - 10011
  - 01101011
  - 0010001

Detecting commonality in Huffman coding – en-/decoding tree construction

Data structure used:
List A(15),B(7),C(6),D(6),E(5) →
List A(15),B(7),C(6),P1(11) →
List A(15),P1(11),P2(13) →
List A(15),P3(24) →
List P4(39)

Priority queue
Operations: EXTRACT-MIN, INSERT

Why is Huffman coding optimal?

S is a data structure containing pairs (a, $f[a]$) where
a is a character in the alphabet and $f[a]$ its frequency in the text.
Q is a priority queue, initially empty.

Priority Queue functions (methods) we need:

EXTRACT-MIN(Q)
INSERT(Q, z)

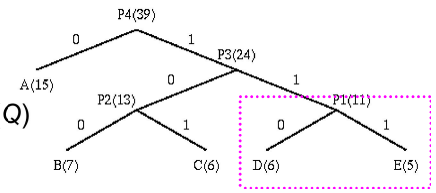
Binary Tree functions (methods) we need?

Use of abstraction in Huffman encoding – data structure & algorithm

- A *priority queue* is a data structure for maintaining a set Q of elements each with an associated value (and *key*).
- A priority queue supports the following operations:
  - INSERT(Q,x) inserts the element x into Q.
  - MIN(Q) returns the element of Q with minimal key.
  - EXTRACT-MIN(Q) removes and returns the element of Q with minimal key.
- Question: difference btn MIN & EXTRACT-MIN

S is a data structure containing pairs (a, $f[a]$) where
a is a character in the alphabet and $f[a]$ its frequency in the text.
Q is a priority queue, initially empty.

HUFFMAN ENCODING (building tree from leaves)
$n \leftarrow |S|$; $Q \leftarrow S$;
for $i \leftarrow 1$ **to** $n-1$
{
$z \leftarrow$ ALLOCATE-NODE()
right[z] $\leftarrow$ EXTRACT-MIN(Q)
$x \leftarrow$ right[z]
left[z] $\leftarrow$ EXTRACT-MIN(Q)
$y \leftarrow$ left[z]
$f[z] \leftarrow f[x] + f[y]$
INSERT(Q, z)
}
return EXTRACT-MIN(Q)



Use of abstraction in Huffman encoding – data structure & algorithm

- A *binary tree* is a data structure for maintaining a set Q of nodes each with an associated value and options of left and right child nodes.
- A *binary tree* supports the following operations:
  - ALLOCATE-NODE creates a new node, returning a reference to the node z created.
  - right(z) refers to the right child node of z.
  - left(z) refers to the left child node of z.

Implementation of Huffman coding using priority queues & binary trees

- Not only priority queue is used in the encoding algorithm shown above
- But also binary tree
- Review again these slides after we finish covering 'Binary trees'

Implementation of Huffman coding using priority queues & binary trees

Information hiding

- Information hiding hides the internal data or information from direct manipulation.
- Information hiding is related to *privacy, security* (Why?)

How does application of 'abstraction' principle simplify the task of Huffman coding?

Information hiding example

- Cars provide an example of this in how they interface with drivers.
- They present a ***standard interface***
  - pedals,
  - wheel,
  - shifter,
  - signals,
  - switches, etc.
- on which people are trained and licensed.
- Implications of this??
- Thus, people only have to learn to drive a car; they don't need to learn a completely different way of driving every time they drive a new model.

Exercise:

implement the algorithm for Huffman decoding

- Input: a pointer to the decoding tree root *z* & received Huffman encoded binary string *i*
- Output: Huffman decoded string
- Do this near the end of this term

Use cases of information hiding

- Hide the physical storage layout for data so that if it is changed, the change is restricted to a small subset of the total program.
- For example, if a three-dimensional point (*x*,*y*,*z*) is represented in a program with three floating point scalar variables and later, the representation is changed to a single array variable of size three…
- A module designed with information hiding in mind would protect the remainder of the program from such a change.

Information hiding

- Information hiding is the principle that users of a module need to know only the essential details of this module (as identified by abstraction)
- So, abstraction leads us to identify details of a module which are important for a user and which are unimportant
- Information hiding tells us that we should actively keep all unimportant details secret from the user and try to prevent him from making use any unimportant details
- The important details of a module that a user needs to know form the specification of a module
- So, information hiding means that modules are used via their specifications, not their implementations.

Encapsulation in an O-O world

- In object-oriented programming, encapsulation is the inclusion within an object of all the resources needed for the object to function – i.e., the methods and the data.
- The object is said to publish its interfaces.
- Other objects adhere to these interfaces to use the object without having to worry how the object accomplishes it.
- The idea is: don't tell me how you do it; just let me know what you can do!
- An object can be thought of as a self-contained atom. The object interface consists of public methods and instantiated data.

Information hiding in an O-O world

- In a well-designed object-oriented application, an object **publicizes *what*** it can do—that is, the services it is capable of providing, or its method headers—but
- ***hides*** the internal details both of ***how*** it performs these services and of the data (attributes & structures) that it maintains in order to support these services.

Encapsulation in communication

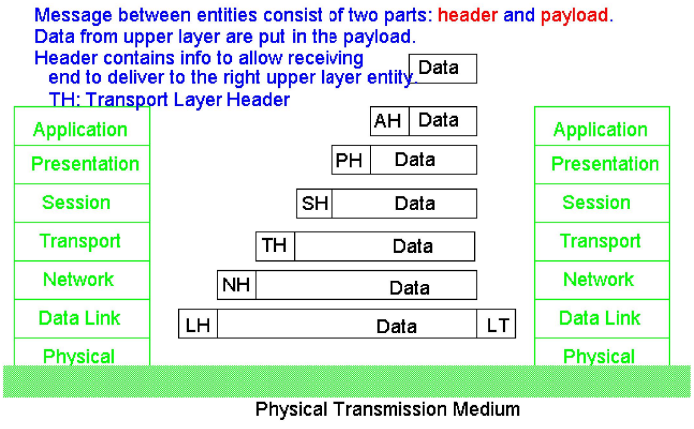
- In communication, encapsulation is the inclusion of one data structure within another structure so that the first data structure is hidden.
- For example, a TCP/IP-formatted packet can be encapsulated within an ATM frame.
- Within the context of sending and receiving the ATM frame, the encapsulated packet is simply a bit stream that describes the transfer.

Java method signature

- The **signature** of a method is the combination of
  - method's name along with
  - number and types of the parameters (and their order).

- *public void setMapReference(int xCoordinate, int yCoordinate)*
  {
  //method code – implementation details
  }
- The method signature is setMapReference(int, int)

Packet encapsulation



Encapsulation

- Like abstraction, we can consider encapsulation either as a process or as an entity:
- As a process, encapsulation means the act of enclosing one or more items (data/functions) within a (physical or logical) container.
- As an entity, encapsulation, refers to a ***package*** or an enclosure that holds (contains, encloses) one or more items (data/functions).

- The separator between the inside and the outside of this enclosure is sometimes called ***wall*** or *barrier.*

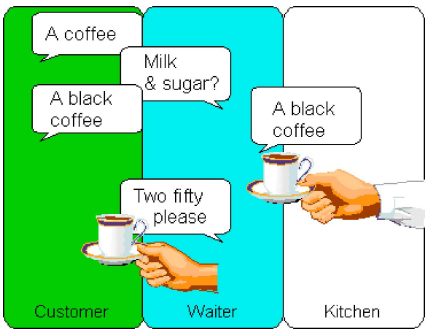
Advantages

Using the processes of abstraction and encapsulation under the guidinglines of information hiding, we enjoy the following advantages:

- Simpler, **modular programs** that are easier to design & understand
- **Side-effects** from direct manipulation of data are **eliminated** or minimised
- **Localisation of errors** (only methods defined on a class can operate on the class data), which allows localised testing
- Program modules are **easier to read, change, and maintain**…

Encapsulation example – 'cup of coffee'

- *Customer*, *waiter* and *kitchen* are three shielded objects in the 'cup of coffee' example.
- Customer does what?
- Waiter does what?
- Kitchen does?
- How do you encapsulate each?
- How do you interface them?



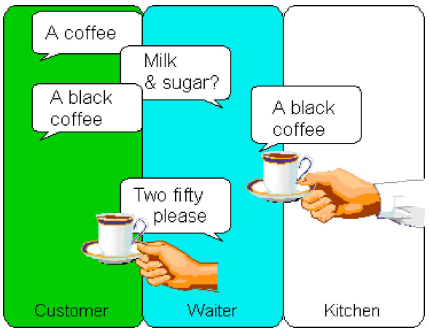
Data structures & abstraction, info hiding, encapsulation

- Data structures represent one big common factor across programs
- Specification of data structures requires the use of abstraction, info hiding, encapsulation

- Practice of abstraction, info hiding, encapsulation on modular programming leads to further development of data structures
- Further development of data structures leads to better modular programming

Encapsulation example

- The customer does not care about the coffee brewing process.
- Even the waiter does not care.
- Encapsulation keeps computer systems flexible. The business process can change easily. (for example?)



Exercise

- Using the info hiding & encapsulation concepts learnt, design a *courier service package* with the following roles inside.
  - Sender
  - Courier receptionist
  - Shipping agent (or delivery agent)
  - Receiver

- Identify the services/functions of each role.
- Clarify how encapsulation is achieved & highlight the service boundary by using a Java method signature like interface.

Comparisons

- Abstraction, information hiding, and encapsulation are different, but related, concepts
- **Abstraction** is a technique that helps us identify which specific information is important for the user of a module, and which information is unimportant

- **Information hiding** is the principle that all unimportant information should be hidden from a user
- **Encapsulation** is then the technique for packaging the information in such a way as to hide what should be hidden, and make visible what is intended to be visible.

Static versus dynamic data structures (1)

- Besides time and space efficiency another important criterion for choosing a data structure is whether the number of data items it is able to store can adjust to our needs or is bounded
- This distinction leads to the notions of dynamic data structures vs. static data structures
- **Dynamic data structures** grow or shrink during run-time to fit current requirements e.g. a structure used in modelling traffic flow
- **Static data structures** are fixed at the time of creation
- e.g. a structure used to store a postcode or credit card number (which has a fixed format)

Static versus dynamic data structures (2)

- Note that it is the *structure* that is static (or dynamic), not the data
- So, in a static data structure the stored data can change over time, only the structure is fixed
- Of course, the stored data could also stay constant in both static and dynamic data structures
- Note that in the definition of a static data structure we have placed no constraints on when it is created, only that ***once it is created it will be fixed***
- This will be important when we determine whether arrays in Java are static data structures or not!

Example

- An example of a static data structure is an array:
  int [ ] a = new int [50] ;
  which allocates memory space to hold 50 integers
- The language provides the construct '**new**' to indicate to the compiler/interpreter how much space of which data type needs to be allocated
- In Java, arrays are always dynamically allocated even when we write:
  inta [ ] = { 10 , 20 , 30 , 40 } ;
- However, the size of the array is fixed
- Q: Is a Java array a static or dynamic data structure?

Efficiency: Space

- A well-chosen data structure should try to minimise memory usage (avoid the allocation of unnecessary space)

- Examples:
- Storing a drawing/map via vectors vs. bitmaps vs. compressed bitmaps
- Tradeoffs of space efficiency vs convenience

Efficiency: Time (1)

- A well-chosen data structure will include operations that are efficient in terms of speed of execution (based on some well-chosen algorithm)
- For our purposes the most important measure for the speed of execution will be the number of accesses to data items stored in the data structure

Efficiency: Time (2)

Example:

- Consider a list of the names of n students and the operation we want to perform is searching for a specific name
- Suppose we use a data structure for implementing lists that allows direct access to an arbitrary element of the list.
- If the list is not sorted in any way, then in the worst case we need to look at each of the n names on the list (n accesses)
- (Q: what is a worst case?)(best case?)(average case?)
- If the list is sorted alphabetically, then we can use **binary search** and in the worst case we need to look at $\log_2(n)$ names on the list ($\log_2(n)$ accesses)

Dynamic data structures

Disadvantages

- Memory allocation/de-allocation overhead
- Whenever a dynamic data structure grows or shrinks, then memory space allocated to the data structure has to be added or removed (which requires time)

Static data structures

Advantages

ease of specification

- Programming languages usually provide an easy way to create static data structures of almost arbitrary size

no memory allocation overhead

- Since static data structures are fixed in size,
  - there are no operations that can be used to extend static structures;
  - such operations would need to allocate additional memory for the structure (which takes time)

Q&A

- Both space efficiency & time efficiency are metrics used to evaluate the performance of an algorithm (and a data structure). (T or F?)
- Dynamic data structures are more space efficient in general. (T or F?)
- Static data structures are more time efficient in general. (T or F?)
- Information hiding is the principle that users of a software component need to know only the essential details of how to *initialize* and *access* the component, and do not need to know the details of the implementation (T or F?)

Static data structures

Disadvantages

- must make sure there is enough capacity

  Since the number of data items we can store in a static data structure is fixed, once it is created, we have to make sure that this number is large enough for all our needs

- more elements? (errors), fewer elements? (waste)
  - However, when our program tries to store more data items in a static data structure than it allows, this will result in an error (e.g. ArrayIndexOutOfBoundsException)
  - On the other hand, if fewer data items are stored, then parts of the static data structure remain empty, but the memory has been allocated and cannot be used to store other data

Summary

- Data structures
- Motivation of studying data structures
- Language to study data structures
- **Abstraction**
- **Huffman coding & dynamic queues**
- **Information hiding**
- **Encapsulation**
- **Efficiency in space & time**
- **Static vs dynamic data structures**

Dynamic data structures

- **Advantages**
- There is no requirement to know the exact number of data items since dynamic data structures can shrink and grow to fit exactly the right number of data items, there is no need to know how many data items we will need to store
- Efficient use of memory space
  - extend a dynamic data structure in size whenever we need to add data items which could otherwise not be stored in the structure and
  - shrink a dynamic data structure whenever there is unused space in it, then the structure will always have exactly the right size and no memory space is wasted

Application of ‘abstraction’ ..

- **Data Mining**, the process of extracting patterns from data, has been applied in many fields to obtain good economical results.
- **Big Data Analytics**
- **Knowledge Discovery in Databases** (KDD) is an emerging field under data mining.
- Just as many other forms of knowledge discovery, KDD creates **abstractions** of the input data.

Readings

- [Mar07] Read 3.1, 3.2, 6.1, 6.4
- [Mar13] Read 3.1, 3.2, 6.1, 6.4

Overview of Data Structure Programming in this Course: Part 1

CP T1 02: 4

Programming with **Linear** collections

- Kinds of collections:
  - Lists, Sets, Bags, Maps, Stacks, Queues, Priority Queues
- Using Linear collections
  - Programming with collections
  - Searching & Sorting Data
  - Implementing linear collections
  - Implementing sorting algorithms
  - Linked data structures and "pointers"

Using the Java Collection Libraries Lecture 2

Overview of Data Structure Programming in this Course: Part 2

CP T1 02: 5

Programming with **Hierarchical** collections

- Kinds of collections:
  - Trees, binary trees, general trees
- Using tree structured collections
  - Building tree structures
  - Searching tree structures
  - Traversing tree structures
- Implementing tree structured collections
- Implementing linear collections
  - with binary search trees

CP T1 02: 2

- What support is offered by Java for programming with data structure?
- What type of data structure can be created using Java?
- How do we create and access data structure under Java?
- Some real examples?

Programming with Libraries

CP T1 02: 6

- Modern programs (especially GUI and network) are too big to build from scratch.

  ⇒ Have to reuse code written by other people

- Libraries are collections of code designed for reuse.
  - Java has a huge collection of standard libraries…. Java API
    - Packages, which are collections of
      - Classes
  - There are lots of other libraries as well
- Learning to use libraries is essential.
- What are the benefits of reuse?

Menu

CP T1 02: 3

- Overview of Data Structure Programming Topics
- Programming with Libraries
- Collections
- Programming with Lists of Objects

Abstract Data Types

Set, Bag, Queue, List, Stack, Map, etc are

Abstract Data Types (outcome of abstraction / encapsulation)

- an ADT is a type of data, described at an abstract level:
  - Specifies the **operations** that can be done to an object of this type
  - Specifies how it will **behave**.
- eg Set: (simple version)
  - Operations: **add**(*value*), **remove**(*value*), **contains**(*value*)→*boolean*
  - Behaviour:
    - A new set contains no values.
    - A set will contain a value *iff*
      - the value has been added to the set and
      - it has not been removed since adding it.
    - A set will not contain a value *iff*
      - the value has never been added to the set, or
      - it has been removed from the set and has not been added since it was removed.

Libraries to use

- java.util — **Collection** classes; Other **utility** classes
- java.io — Classes for **input and output**
- javax.swing, java.awt — Large library of classes for **GUI** programs

We will use these libraries in almost every program

Java Collections library

Interfaces:

- *Collection*
  = Bag (most general)
- *List*
  = ordered collection
- *Set*
  = unordered, no duplicates
- *Queue*
  ordered collection, limited access
  (add at one end, remove from other)
- *Map*
  = key-value pairs (or mapping)

Classes

- List classes:
  ArrayList, LinkedList, vector…
- Set classes:
  HashSet, TreeSet,…
- Map classes:
  HashMap, TreeMap, …
- …

Using Libraries

- Read the documentation to pick useful library — Java API
- **import** the package or class into your program

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

- Read the documentation to identify how to use
  - **Constructors** for making instances
  - **Methods** to call
  - **Interfaces** to implement
- Use the classes as if they were part of your program

Java Interfaces and ADT's

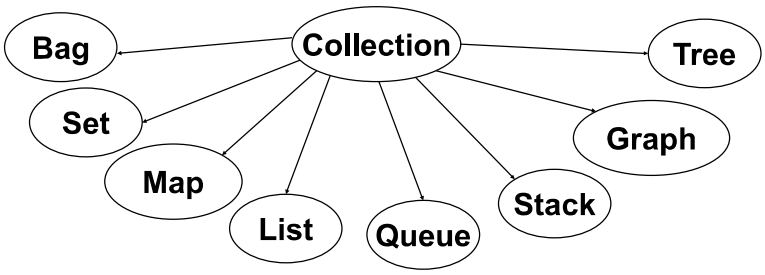
- A Java ***Interface*** corresponds to an Abstract Data Type
  - Specifies what methods can be called on objects of this type (specifies name, parameters and types, and type of return value)
  - Behaviour of methods is only given in comments (but cannot be enforced)

× No constructors - can't make an instance: **new** Set()
× No fields - doesn't say how to store the data
× No method bodies. - doesn't say how to perform the operations

```
public interface Set {
    public void add(??? item);          /*…description…*/
    public void remove(??? item);       /*…description…*/
    public boolean contains(??? item);  /*…description…*/
    …
      // (plus lots more methods in the Java Set interface)
```

"Standard" Collections

Common ways of organising a collection of values:



Each of these is a different **type** of collection
- values organised/structured differently
- different constraints on duplicates, and on access
- very abstract –
  - don't care what type the elements are.
  - don't care how they're stored or manipulated inside

Parameterised Types

- Interfaces may have type parameters (*eg*, type of the element):

It's a Set of something, as yet unspecified

```
public interface Set <T> {
    public void add(T item);          /*…description…*/
    public void remove(T item);       /*…description…*/
    public boolean contains(T item); /*…description…*/
    …   // (lots more methods in the Java Set interface)
```

- When declaring variable, specify the actual type of element

```
private Set <Person>  friends;
private List <Shape> drawing;
```

Collection Type

Type of value in Collection

List Interface in Java

The real **List** interface in Java 1.5 is defined as follows.

```
public interface List<E> extends Collection<E> {
...
boolean add(E o);
E get(int index);
...
boolean contains(Object o);
Iterator<E> iterator(); ...
}
```


Using the Java Collection Library

Problem:

- How do you create an instance of the interface?

  Interfaces don't have constructors!

  ```
  private List <Shape> drawing = new ???? ( );
  ```

- **Classes** in the Java Collection Library ***implement*** the interfaces
  - Define constructors to construct new instances
  - Define method bodies for performing the operations
  - Define fields to store the values

  ⇒ Your program can create an instance of a class.

  ```
  private List <Shape> drawing = new ArrayList <Shape> ( );
  Set <Person> friends = new HashSet <Person> ( );
  ```


Using Java Collection Interfaces

- Your program can
  - Declare a variable, parameter, or field of the interface type

    ```
    private List drawing;        // a list of Shapes
    ```

  - Call methods on that variable, parameter, or field

    ```
    drawing.add(new Rect(100, 100, 20, 30))
    ```

× Problem:

- How do we specify the type of the values?

ArrayList

- Part of the Java **Collections** framework.
  - predefined class
  - stores a list of items,
    - a collection of items kept in a particular order.
  - part of the java.util package

    ⇒ need to **import java.util.*;** at head of file

- You can make a new ArrayList object, and put items in it
  - Don't have to specify its size
  - Should specify the type of items.
    - new syntax: "type parameters"
  - Like an infinitely stretchable array
  - But, you can't use the […] notation
  - you have to call methods to access and assign

Parameterised Types

- The structure and access discipline of a collection is the same, regardless of the type of value in it:
  - A set of Strings, a set of Persons, a set of Shapes, a set of integers all behave the same way.

⇒ Only want one Interface for each kind of collection.
(there is only one *Set* interface)

- Need to specify kind of values in a particular collection

⇒ The collection Interfaces (and classes) are parameterised:

- Interface has a **type parameter**
- When declaring a variable collection, you specify
  - the type of the collection and
  - the type of the elements of the collection

Q&A

- Name 3 type of collections that can be implemented under Java Programming with Linear collections
- Name 3 operations that can be implemented under Java Programming with Linear collections
- Name 2 type of collections that can be implemented under Java Programming with Hierarchical collections
- Name 4 operations that can be implemented under Java Programming with Hierarchical collections
- What is a software "library"?
- Define Java "Package".
- Name Java's IO library.
- Name Java's GUI library.
- What is the Java statement to include package or class into your program?

Using ArrayList: declaring

List of students

- Array:

```
private static final int maxStudents = 1000;
private Student[ ] students = new Student[maxStudents];
private int count = 0;
```

- Alternatively, we can do the following…
- ArrayList:

```
private ArrayList <Student> students = new ArrayList <Student>();
```

  - The type of values in the list is between "<" and ">" after ArrayList.
  - No maximum; no initial size; no explicit count

Conclusions

- We can declare the type of a variable/field/parameter to be a collection of some element type
- We can construct a new object of an appropriate collection class.

What's next?

- What can we do with them?
  - What methods can we call on them?
  - How do we iterate down all the elements of a collection?
- How do we choose the right collection interface and class?

Using ArrayList: methods

- ArrayList has many methods! , including:
  - size(): returns the number of items in the list
  - add(*item*): adds an item to the *end* of the list
  - add(*index, item*): inserts an item at *index* (relocates later items)
  - set(*index, item*): replaces the item at *index* with *item*
  - contains(*item*): true if the list contains an item that equals *item*
  - get(*index*): returns the item at position *index*
  - remove(*item*): removes an occurrence of item

    (what if there are duplicates in the ArrayList?)
  - remove(*index*): removes the item at position *index* (both relocate later items)
  - You can use the "for each" loop on an array list, as well as a **for** loop

Readings

- [Mar07] Read 3.3
- [Mar13] Read 3.3

Using ArrayList

```
private ArrayList <Student> students = new ArrayList <Student>();
      :

Student s = new Student("Lindsay King", "300012345")
students.add(s);
students.add(0, new Student(fscanner));

for (int i = 0; i<students.size(); i++)
    System.out.println(students.get(i).toString());

for (Student  st : students)
    System.out.println(st.toString());

if (students.contains(current)){
    file.println(current);
    students.remove(current);
}
```

Comments on code style

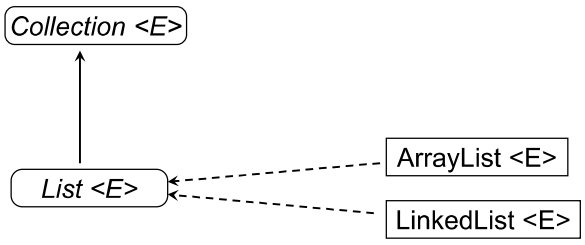
- We will drop "this." except when needed.
  - instead of `this.loadFromFile(fname)`
    just `loadFromFile(fname)`
- We will leave out { } when surrounding just one statement
  - instead of
```
while (i < name.length) {
  name[i] = null;
}
```
    just
```
while (i < name.length)
  name[i] = null;
```

More on Collections

Lecture 3

Collection Types

Interfaces Classes



Interfaces can **extend** other interfaces:

The **sub** interface has all the methods of the **super** interface plus its own methods (**sub** means? **super** means?)

Motivation for this study

- What type of 1-dimensional data structure is supported by Java?
- How do we create 1-dimensional data structure under Java?
- How do we use them? How to maintain them?
- Can we iterate through a 1-dimensional data structure under Java?
- More examples?

Methods on Collection and List

- Collection <E>
  - isEmpty() → boolean
  - size() → int
  - contains(E elem) → boolean
  - add(E elem) → boolean (whether it succeeded)
  - remove(E elem) → boolean (whether it removed an item)
  - iterator() → iterator <E>
  - …

Methods on all types of collections

- List <E>
  - add(int index, E elem)
  - remove(int index) → E (returns the item removed)
  - get(int index) → E
  - set(int index, E elem) → E (returns the item replaced)
  - indexOf(E elem) → int
  - subList(int from, int to) → List<E>
  - …

Additional methods on all Lists

Menu

- Collections and List
- Using List and ArrayList
- Iterators

Iterating through List:

```
public void displayTasks(){
    textArea.setText(tasks.size() +" tasks to be done:\n");
    for (Task task : tasks){
        textArea.append(task + "\n");
}
```

Automatically calls toString() method. What is being displayed?

Or

```
for (int i=0; i<tasks.size(); i++)
    textArea.append(tasks.get(i) + "\n");
```

Or

```
iterator <Task> iter = tasks.iterator();
while (iter.hasNext()){
    textArea.append(iter.next() +  "\n");
```

Using a collection type

- Variable or field declared to be of the interface type
  - Specify the type of the collection
  - Specify the type of the value

  private List <Task> tasks;
  - The type between "<" and ">" is the type of the elements
- Create an object of a class that implements the type:
  - Specify the class
  - Specify the type of the value

  tasks = new ArrayList <Task> ();
- Call methods on the object to access or modify

More of the TodoList example:

```
public void actionPerformed(ActionEvent e){
    String but = e.getActionCommand();
    if ( but .equals("Add") )              tasks.add(askTask());
    else if (but.equals("Remove") ) tasks.remove(askTask());
    else if (but.equals( "AddAt" ) )
        tasks.add(askIndex("add where?"), askTask());
    else if (but.equals("RemoveFrom") )
        tasks.remove(askIndex("from"));
    else if (but.equals("MoveTo") ){
        int from= askIndex("move from position: ");
        int to = askIndex("move to position: ")
                    Task task= tasks.get(from);
                    tasks.remove(from);
                    tasks.add(to, task);
        }
    displayTasks();
}
```

Example

- TodoList – collection of tasks, in order they should be done.
- Collection type: List of tasks
- Requirements of TodoList:
  - read list of tasks from a file,
  - display all the tasks
  - add task, at end, or at specified position
  - remove task,
  - move task to a different position.

Iterators

How does the "for each" work?

```
for (Task task : tasks){
    textArea.append(task + "\n");
```

An iterator object attached to tasks that will keep giving you the next element

- **Turns into**

```
Iterator <Task> iter = tasks.iterator();
while (iter.hasNext()){
    Task  task = iter.next();
    textArea.append(task +  "\n");
```

Iterator is an interface:

A Scanner is a fancy iterator

```
public interface Iterator <E> {
    public boolean hasNext();
    public E next();
}
```

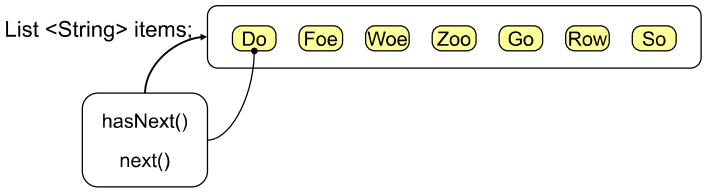
Example (TodoList program)

```
public class TodoList implements ActionListener{
    :
  private List<Task> tasks;
    :
  /* read list of tasks from a file, */
  public void readTasks(String fname){
    try {
        Scanner sc = new Scanner(new File(fname));
        tasks = new ArrayList<Task>();
        while ( sc.hasNext() )
            tasks.add(new Task(sc.next()));
        sc.close();
    } catch(IOException e){…}
    displayTasks();
  }
```

Q&A: How does ArrayList work?

- What does it store inside?
- How does it keep track of the size?
- How does it grow when necessary?
- How does its iterator work?

Iterators




```
for ( String str : items)   System.out.print(str + ", ");

Iterator<String> iter = items.iterator();
while (iter.hasNext()){
  String str = iter.next();
  System.out.print(str + ", ");
}
```

Summary

- Collections and List
- Using List and ArrayList
- Iterators

Q&A: Compare List against Array in the following aspects

- Lists are nicer than arrays:
  - … …
  - … …

List	Array
jobList.set(ind, value)	?
jobList.get(ind)	?
jobList.size()	?
jobList.add(value)	?
jobList.add(ind, value)	?
jobList.remove(ind)	?
jobList.remove(value)	?

- **for** (Task t : tasks) *vs* **for**(…) { … …}

Readings

- [Mar07] Read 3.4
- [Mar13] Read 3.4

List *vs* Array

- Lists are nicer than arrays:
  - No size limit!!! They grow bigger as necessary
  - Lots of code written for you:

jobList.set(ind, value)	jobArray[ind] = value
jobList.get(ind)	jobArray[ind]
jobList.size()	? *(Not the length!!!)*
jobList.add(value)	? *(Where is the last value? What happens if it's full?)*
jobList.add(ind, value)	? *(Have to shift everything up!!!)*
jobList.remove(ind)	? *(Have to shift everything down!!!)*
jobList.remove(value)	? *(Have to find value, then shift things down!!!)*

- **for** (Task t : tasks) *vs* **for**(int i = 0; i< ???; i++){
  Task t = taskArray[i];

More Collections: Bags, Sets, Stacks, Maps

Lecture 4

Bags

- A Bag is a collection with
  - no structure or order maintained
  - no access constraints (access any item any time)
  - duplicates allowed
- Minimal Operations:
  - **add**(value) → returns true *iff* a collection was changed
  - **remove**(value) → returns true *iff* a collection was changed
  - **contains**(value) → returns true *iff* value is in bag
    uses equal to test.
  - **findElement**(value) → returns a matching item, *iff* in bag
- Plus
  - **size**(), **isEmpty**(), **iterator**(), **clear**(), **addAll**(collection), **removeAll**(collection), **containsAll**(collection), …

Bag Applications

- When to use a Bag?
  - When there is no need to order a collection, and duplicates are possible:
    - A collection of current logged-on users (can there be dups?)
    - The books in a book collection (can there be dups?)
    - …
- There are no standard implementations of Bag!!

Menu

- More Collections
- Bags and Sets
- Stacks and Applications
- Maps and Applications

Set ADT

- Set is a collection with:
  - no structure or order maintained
  - no access constraints (access any item any time)
  - Only property is that duplicates are excluded
- Operations:

  (Same as Bag, but different behaviour)
  - add(value) → true *iff* value *was added (ie, no duplicate)*
  - remove(value) → *true iff value removed (was in set)*
  - contains(value) → *true iff value is in the set*
  - findElement(value) → *matching item, iff value is in the set*
  - …
- Sets are as common as Lists

Collections library

Interfaces:

- *Collection <E >*
  = Bag (most general)
- *List <E >*
  = ordered collection
- *Set <E >*
  = unordered, no duplicates
- *Stack<E>*
  ordered collection, limited access
  (add/remove at end)
- *Map <K, V >*
  = key-value pairs (or mapping)
- *Queue <E >*
  ordered collection, limited access
  (add at end, remove from front)

Classes

- List classes:
  ArrayList, LinkedList,
- Set classes:
  HashSet, TreeSet, …
- Stack classes:
  ArrayStack, LinkedStack
- Map classes:
  HashMap, TreeMap, …
- …

Applications of Stacks

- Processing files of structured (nested) data.
  - E.g. reading files with structured markup (HTML, XML,…)
- Program execution, e.g. working on subtasks, then returning to previous task.
- Undo in editors.
- Expression evaluation,
  - (6 + 4) * ((12.1 *sin(15)) – (cos(20) / 38))

Stack

- Organizes entries according to the order in which added
- Additions are made to one end, the top
- The item most recently added is always on the top

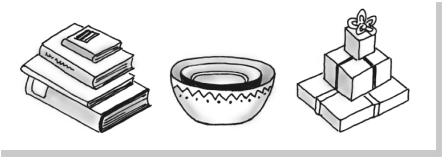


Fig. Some familiar stacks

HTML & XML examples

- **HTML example**

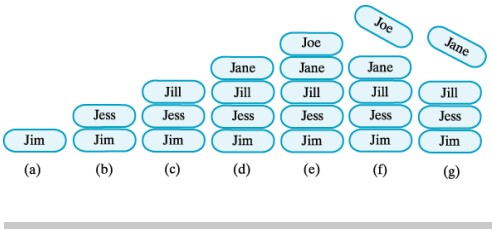
```
<html>
<body>
The content of the body element is displayed in your browser.
</body>
</html>
```

- **XML examples**

```
<Person>
        <name>Henry Ford</name>
</Person>

<Book>
        <title>My Life and Work</title>
        <author>Henry Ford</author>
</Book>
```

Stack example

Fig. A stack of strings after
(a) push adds *Jim*;
(b) push adds *Jess*;
(c) push adds *Jill*;
(d) push adds *Jane*;
(e) push adds *Joe*;
(f) pop retrieves and removes *Joe*;
(g) pop retrieves and removes *Jane*



How do we make sure XML/HTML tags in a web document is properly nested?

Stacks

- Stacks are a special kind of List:
  - Sequence of values, ('sequence' means?)
  - Constrained access: add, get, and remove only from one end.
  - There exists a Stack interface and different implementations of it (ArrayStack, LinkedStack, etc)
  - In Java Collections library:
    - Stack is a class that implements List
    - Has extra operations: **push**(value), **pop**(), **peek**()
- push(value): Put value on top of stack
- pop(): Removes and returns top of stack
- peek(): Returns top of stack, *without* removing
- plus the other List operations

Stack for evaluating expressions

- (6 + 4) * ((12.1 *sin(15)) – (cos(20) / 38))
- How does it work?

The Program Stack for Program Execution

- When a method is called
  - Runtime environment creates activation record
  - Shows method's state during execution
- Activation record pushed onto the program stack (Java stack)
  - Top of stack belongs to currently executing method
  - Next record down the stack belongs to the one that called current method

Using a Stack to Process Algebraic Expressions

- Checking for Balanced Parentheses, Brackets, and Braces in an Infix Algebraic Expression
- Transforming an Infix Expression to a Postfix Expression
- Evaluating Postfix Expressions
- Evaluating Infix Expressions

The Program Stack

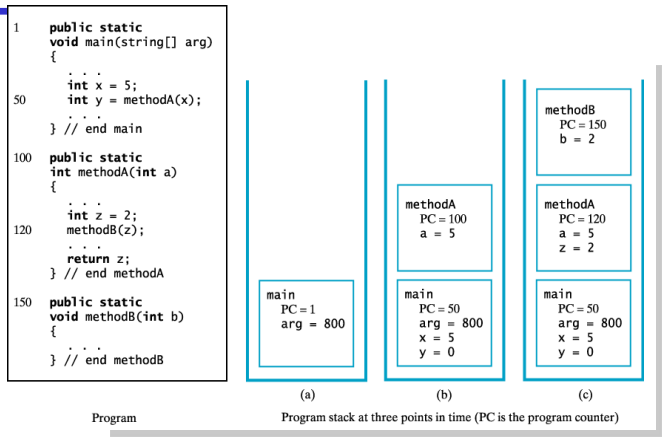


Fig. The program stack at 3 points in time; (a) when `main` begins execution; (b) when `methodA` begins execution, (c) when `methodB` begins execution.

Using a Stack to Process Algebraic Expressions

- **Infix expressions**
  - Binary operators appear between operands
  - `a + b`
- **Prefix expressions**
  - Binary operators appear before operands
  - `+ a b`
- **Postfix expressions**
  - Binary operators appear after operands
  - `a b +`
  - Easier to process – no need for parentheses nor precedence (why?)

Recursive Methods

- A recursive method making many recursive calls
  - Places many activation records in the program stack
  - Explains why recursive methods can use much memory
- Possible to replace recursion with iteration by using a stack

Checking for Balanced (), [], {}

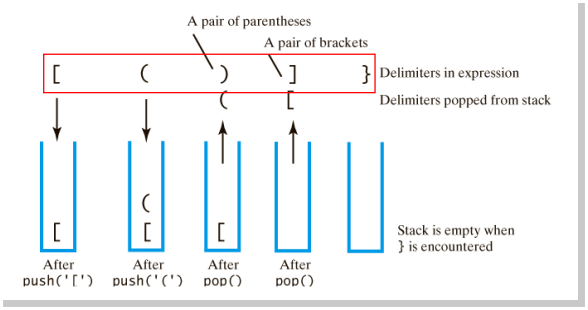


Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters `[()]}`

Checking for Balanced (), [], {}

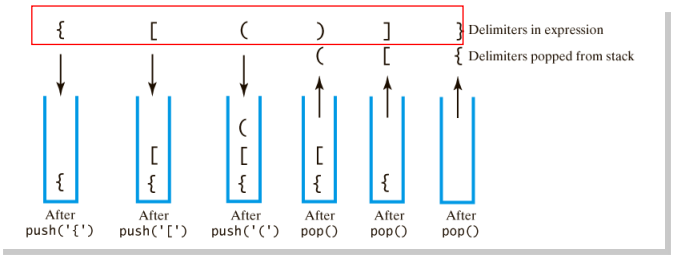


Fig. The contents of a stack during the scan of an expression that contains the balanced delimiters `{[()]}`

Q&A: Checking for Balanced (), [], {}

Show stack contents

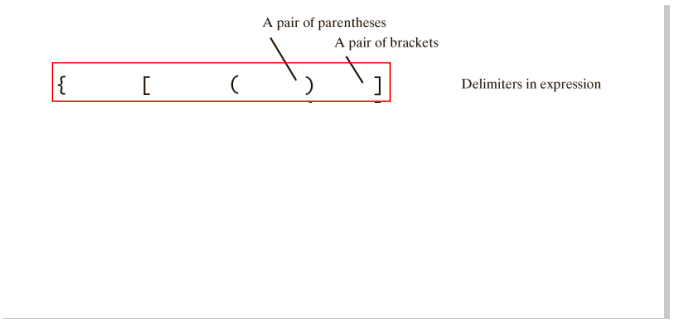


Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters `{[()]`

Checking for Balanced (), [], {}

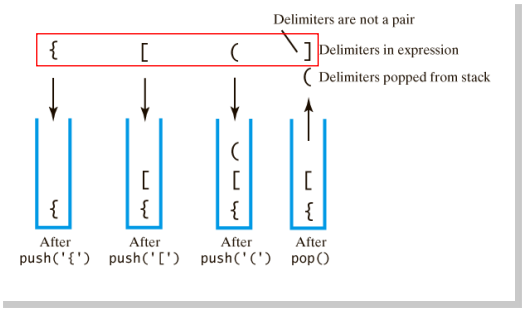


Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters `{[(])}`

Checking for Balanced (), [], {}

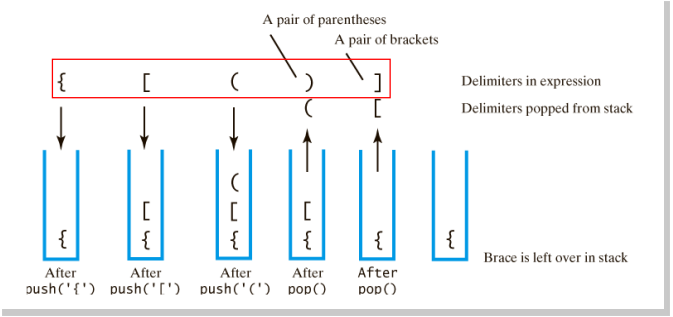


Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters `{[()]`

Q&A: Checking for Balanced (), [], {}

Show stack contents

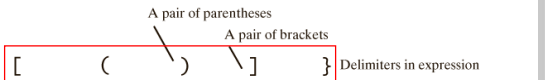


Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters `[()]}`

Transforming Infix to Postfix

Next Character	Postfix	Operator Stack (bottom to top)
a	*a*	
−	*a*	−
b	*a b*	−
+	*a b* −	
	a b −	+
c	*a b* − *c*	+
	a b − *c* +	

Fig. Converting infix expression **a – b + c** to postfix form: **a b – c +**

Checking for Balanced `()`, `[]`, `{}`

```
Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets, and braces in an expression are paired correctly.
isBalanced = true
while ( (isBalanced == true) and not at end of expression)
{   nextCharacter = next character in expression
    switch (nextCharacter)
    {   case '(': case '[': case '{':
            Push nextCharacter onto stack
            break
        case ')': case ']': case '}':
            if (stack is empty)  isBalanced = false      else
            {   openDelimiter = top of stack
                Pop stack
                isBalanced = true or false according to whether openDelimiter and
                     nextCharacter are a pair of delimiters
            }
            break
    }
}
if (stack is not empty)  isBalanced = false
return isBalanced
```

Transforming Infix to Postfix

Next Character	Postfix	Operator Stack (bottom to top)
a	*a*	
^	*a*	^
b	*a b*	^
^	*a b*	^ ^
c	*a b c*	^ ^
	a b c ^	^
	a b c ^ ^	

Fig. Converting infix expression **a ^ b ^ c** to postfix form: **a b c ^ ^**

Exercise: rewrite this algorithm in pseudo code

Infix-to-Postfix Algorithm

Symbol in Infix	Action
Operand	Append to end of output expression
Operator ^	Push ^ onto stack
Operator +,-, *, or /	Pop operators from stack, append to output expression until stack empty or top has lower precedence than new operator. Then push new operator onto stack
Open parenthesis	Push (onto stack
Close parenthesis	Pop operators from stack, append to output expression until we pop a matching open parenthesis. Discard both parentheses.

Transforming Infix to Postfix

Next Character	Postfix	Operator Stack (bottom to top)
a	*a*	
+	*a*	+
b	*a b*	+
*	*a b*	+ *
c	*a b c*	+ *
	a b c *	+
	a b c * +	

Fig. Converting the infix expression `a + b * c` to postfix form `a b c * +`

Evaluating Postfix Expression

Infix expression **(a+b)/c** is converted into the postfix expression **a b + c /**

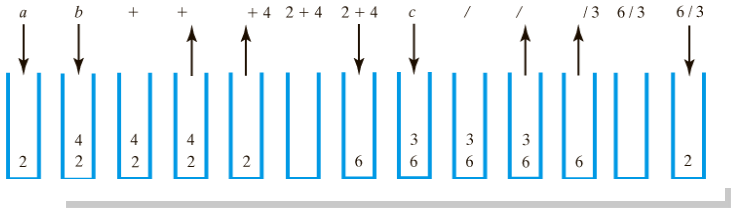


Fig. The stack during the evaluation of the postfix expression `a b + c /` when `a` is 2, `b` is 4 and `c` is 3

Exercise

- Convert infix to postfix & show stack contents:

- (a+n)*(b-8*m)
- b/v^7
- {3+[d-7*(g+5)]/w}
- [(4+b]-2)

Evaluating Postfix Expression

```
Algorithm evaluatePostfix(postfix)    // Evaluates a postfix expression.
valueStack = a new empty stack
while (postfix has characters left to parse)
{    nextCharacter = next nonblank character of postfix
     switch (nextCharacter)
     {    case variable:
               valueStack.push(value of the variable nextCharacter)
               break
          case '+': case '-': case '*': case '/': case '^':
               operandTwo = valueStack.pop()
               operandOne = valueStack.pop()
               result = the result of the operation in nextCharacter and its
               operands   operandOne and operandTwo
               valueStack.push(result)
               break
          default: break
     }
}
return valueStack.peek()                    /* does not check errors in postfix */
```

Evaluating Postfix Expression

Infix expression: **a/b** is converted into postfix expression: **ab/**

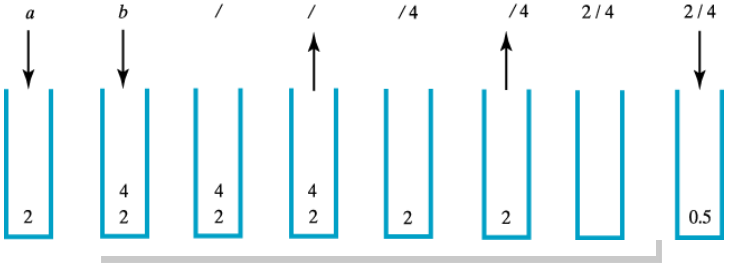


Fig. The stack during the evaluation of the postfix expression `a b /` when `a` is 2 and `b` is 4

Exercise:

add error processing to the pseudo code of Postfix Expression Evaluation

Q&A: Evaluating Postfix Expression: show stack contents

Infix expression **(a+b)/c** is converted into the postfix expression **a b + c /**

Fig. The stack during the evaluation of the postfix expression `a b + c /` when `a` is 2, `b` is 4 and `c` is 3

Java Class Library: The Class `Stack`

- Methods in class `Stack` in `java.util`

```
public void push(Object item);
public Object pop();
public Object peek();
public boolean isEmpty();
public void clear();
```

Evaluating Infix Expressions using Two Stacks

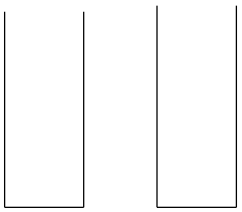


Fig. Two stacks during evaluation of **a + b * c** when **a = 2, b = 3, c = 4;** (a) after reaching end of expression; (b) while performing multiplication; (c) while performing the addition

Q&A: What should be included in a Java Stack Interface Spec ?

Evaluating Infix Expressions using Two Stacks

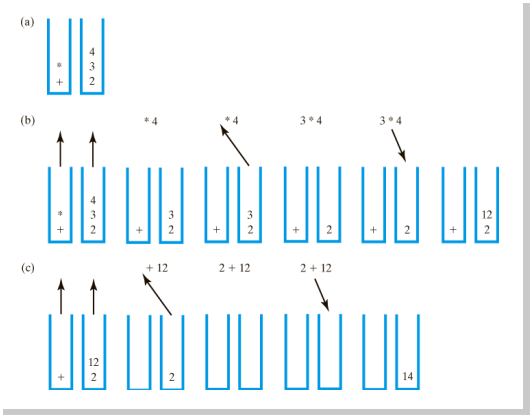


Fig. Two stacks during evaluation of **a + b * c** when **a = 2, b = 3, c = 4;** (a) after reaching end of expression; (b) while performing multiplication; (c) while performing the addition

Spec of the ADT Stack

- Specification of a stack of objects

```
public interface StackInterface
{    /** Task: Adds a new entry to the top of the stack.
     * @param newEntry an object to be added to the stack */
     public void push(Object newEntry);

     /** Task: Removes and returns the top of the stack.
     * @return either the object at the top of the stack or null if  the stack was
empty */
     public object pop();

     /** Task: Retrieves the top of the stack.
     * @return either the object at the top of the stack or null if the stack is
empty */
     public object peek();

     /** Task: Determines whether the stack is empty.
     * @return true if the stack is empty */
     public boolean isEmpty();

     /** Task: Removes all entries from the stack */
     public void clear();
} // end StackInterface
```

Ex. Try infix evaluation with dual stacks on the following: a * b + c when a = 2, b = 3, c = 4

Example of using Map

- Find the highest frequency word in a file

  ⇒ must count frequency of every word.

  ie, need to associate a count (int) with each word (String)

  ⇒ use a Map of word–count pairs:

- Two Steps:
  - construct the counts of each word: countWords(file) → map
  - find the highest count findMaxCount(map) → word

```
System.out.println( findMaxCount( countWords(file) ) );
```

Stacks Example

- Reversing the items from a file:
  - Read and push onto a stack
  - Pop them off the stack

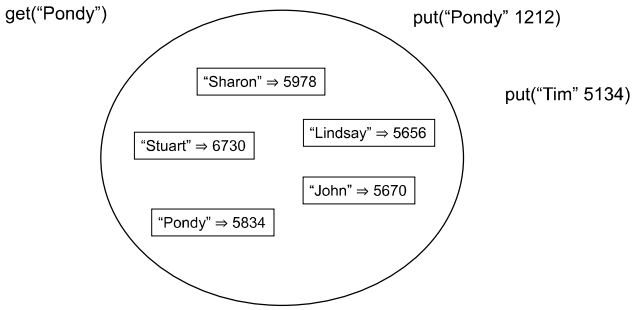
```
public void reverseNums(Scanner sc){
  Stack<Integer> myNums = new ArrayStack<Integer>();
  while (sc.hasNext())
     myNums.push(sc.Next())
  while (! myNums.isEmpty())
     textArea.append(myNums.pop() + "\n");
}
```

Example of using Map – in pseudocode

```
/** Construct histogram of counts of all words in a file */
public Map<String, Integer> countWords(Scanner sc){
    // construct a new map
    // for each word in the file
    //     if word is in the map, increment its count
    //     else, add it to the map with a count of 1
    // return map
}

/** Find word in histogram with highest count */
public String findMaxCount(Map<String, Integer> counts){
    // for each word in map
    //    if has higher count than current max, record it
    // return current max word
}
```

Maps

- Collection of data, but not of single values:
  - Map = **Set** of pairs of keys to values
  - Constrained access: get values via keys.
  - **No duplicate keys**
  - Lots of implementations, most common is **HashMap.**



Example of using Map

```
/** Construct histogram of counts of all words in a file */
public Map<String, Integer> countWords(Scanner scan){
    Map<String, Integer>  counts = new HashMap<String, Integer> ();
    for (String word : scan){
      if ( counts.containsKey(word) )
          counts.put(word, counts.get(word)+1);
      else
          counts.put(word, 1);
    return counts;
}

/** Find word in histogram with highest count */
public String findMaxCount(Map<String, Integer> counts){
    // for each word in map
    //    if has higher count than current max, record it
    // return current max word
}
```

Maps

- When declaring and constructing, must specify two types:
  - Type of the key, and type of the value

```
private Map<String, Integer> phoneBook;
  :
phoneBook = new HashMap<String, Integer>();
```

- Central operations:
  - get(key), → returns value associated with key (or null)
  - put(key, value), → sets the value associated with key (and returns the old value, if any)
  - remove(key), → removes the key *and* associated value (and returns the old value, if any)
  - containsKey(key), → boolean
  - size()

Summary

- More Collections
- Bags and Sets
- Stacks and Applications
- Maps and Applications

Readings

- [Mar07] Read 3.6, 4.8
- [Mar13] Read 3.6, 4.8

Iterating through a Map

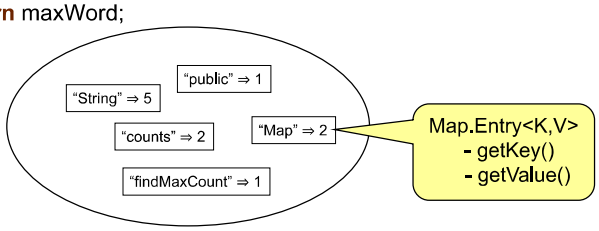
- How do you iterate through a Map? (eg, to print it out)
  - A Map isn't just a collection of items!
    ⇒ could iterate through the collection of keys
    ⇒ could iterate through the collection of values
    ⇒ could iterate through the collection of pairs
- Java Map allows all three!
  - keySet() → Set of all keys
    - **for** (String name : phonebook.keySet()){….
  - values() → Collection of all values
    - **for** (Integer num : phonebook.values()){….
  - entrySet() → Set of all Map.Entry's
    - **for** (Map.Entry<String, Integer> entry : phonebook.entrySet()){….
      … entry.getKey() …
      … entry.getValue()…

Iterating through Map: keySet

```
/** Find word in histogram with highest count */
public String findMaxCount(Map<String, Integer> counts){
    String maxWord = null;
    int maxCount = -1;
    for (String word : counts.keySet() ){
      int count = counts.get(word);
      if (count > maxCount){
          maxCount = count;
          maxWord = word;
      }
    }
    return maxWord;
}
```

Iterating through Map: entrySet

```
public String findMaxCount(Map<String, Integer> counts){
    String maxWord = null;
    int maxCount = -1;
    for (Map.Entry<String, Integer> entry : counts.entrySet() ){
      if (entry.getValue() > maxCount){
          maxCount = entry.getValue();
          maxWord = entry.getKey();
      }
    }
    return maxWord;
}
```



Example of using Map

```
/** Construct histogram of counts of all words in a file */
public Map<String, Integer> countWords(Scanner sc){
    // construct new map
    // for each word in file
    //    if word is in the map, increment its count
    //    else, put it in map with a count of 1
    // return map
}

/** Find word in histogram with highest count */
public String findMaxCount(Map<String, Integer> counts){
    // for each word in map
    //   if has higher count than current max, record it
    // return current max word
}
```

Queues and Iterators

Lecture 5

Example of using Map

```
/** Construct histogram of counts of all words in a file */
public Map<String, Integer> countWords(Scanner scan){
    Map<String, Integer>  counts = new HashMap<String, Integer> ();
    while (scan.hasNext()){
      String word = scan.next();
      if ( counts.containsKey(word) )
          counts.put(word, counts.get(word)+1);
      else
          counts.put(word, 1);
    }
    return counts;
}
/** Find word in histogram with highest count */
public String findMaxCount(Map<String, Integer> counts){
    // for each word in map
    //   if has higher count than current max, record it
    // return current max word
}
```

Menu

- Examples of using Map
- Queues and Priority Queues
- Classes/Interfaces that accompany collections
  - Iterator
  - Iterable

Iterating through Map: keySet

```
/** Find word in histogram with highest count */
public String findMaxCount(Map<String, Integer> counts){
    String maxWord = null;
    int maxCount = -1;
    for (String word : counts.keySet() ){
      int count = counts.get(word);
      if (count > maxCount){
          maxCount = count;
          maxWord = word;
      }
    }
    return maxWord;
}
```

Example of using Map

- Find the highest frequency word in a file

  ⇒ must count frequency of every word.

  ie, need to associate a count (int) with each word (String)

  ⇒ use a Map of word–count pairs:

- Two Steps:
  - construct the counts of each word: countWords(file) → map
  - find the highest count findMaxCount(map) → word

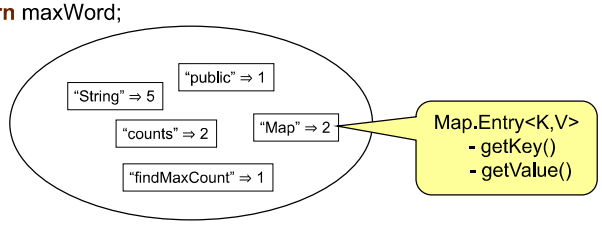
```
System.out.println( findMaxCount( countWords(file) ) );
```

Queue Operations

- **offer**(value) ⇒ boolean
  - add a value to the queue
  - (sometimes called “enqueue”)
- **poll**() ⇒ *value*
  - remove and return value at front/head of queue or null if the queue is empty
  - (sometimes called “dequeue”, like “pop”)
- **peek**() ⇒ *value*
  - return value at head of queue, or null if queue is empty (doesn’t remove from queue)
- **remove**() and **element**()
  - like poll() and peek(), but throw exception if queue is empty.

Iterating through Map: entrySet

```
public String findMaxCount(Map<String, Integer> counts){
    String maxWord = null;
    int maxCount = -1;
    for (Map.Entry<String, Integer> entry : counts.entrySet() ){
      if (entry.getValue() > maxCount){
          maxCount = entry.getValue();
          maxWord = entry.getKey();
      }
    }
    return maxWord;
}
```

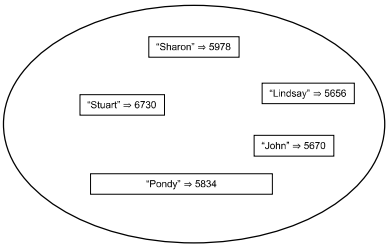


Iteration and “for each” loop

- Standard “for each” loop with collections:

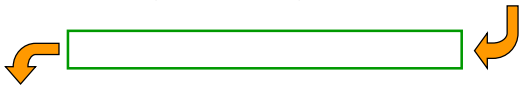
```
for (Face face : crowd) {
    face.render(canvas);
}
for (Map.Entry<String,Integer> entry : phonebook.entrySet()){
    textArea.append(entry.getKey()+" : "+entry.getValue());
}
```

- Uses Iterators.

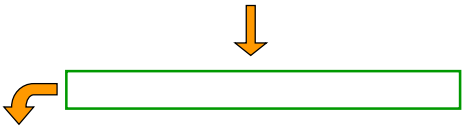


Queues

- Queues are like/unlike Stacks
  - Collection of values with an order
  - Constrained access:
    - Only remove from the front
  - Two varieties:
    - **Ordinary queues**: only add at the back

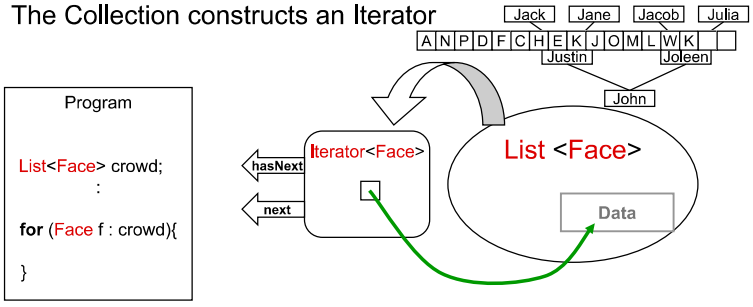


- **Priority queues**: add or remove with a given priority



Why Iterators?

- Program cannot get inside the Collection object
- The Collection constructs an Iterator



- Iterator may access inside of collection
- Iterator provides elements one at a time.
- Each Collection class needs an associated Iterator class

Queues

- Used for
  - Operating Systems, Network Applications, Multi-user Systems
    - Handling requests/events/jobs that must be done in order
    - (memory pool holding such requests are often called a “buffer” in this context)
  - Simulation programs
    - Representing queues in the real world (traffic, customers, deliveries, ….)
    - Managing events that must happen in the future
  - Search Algorithms
    - Computer Games
    - Artificial Intelligence
- Java provides
  - a Queue interface
  - several classes: **LinkedList, PriorityQueue**

Creating an Iterable

- Class that provides an Iterator:
  - eg: A NumberSequence representing an infinite arithmetic sequence of numbers, with a starting number and a step size,
    eg 5, 8, 11, 14, 17,….

```
public class NumberSequence implements Iterable<Integer>{
    private int start;
    private int step;
    public NumberSequence(int start, int step){
      this.start = start;
      this.step = step;
    }
    public Iterator<Integer> iterator(){
      return new NumberSequenceIterator(this);
    }
```

Iterator Interface

- Operations on Iterators:
  - **hasNext()** : *returns* true *iff there is another value to get*
  - **next()** : *returns the next value*
- Standard pattern of use:

```
Iterator<type > itr  = construct iterator
while (itr.hasNext() ){
    type var = itr.next();
    … var …
}
```

- Almost same as the "for each" loop:

```
for (type var :  collection ){
    … var …
```

Creating an Iterator for an Iterable

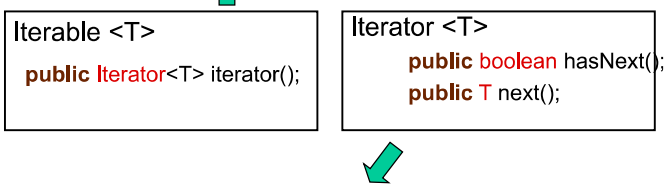
```
private class NumberSequenceIterator implements Iterator<Integer>{
    private int nextNum;
    private NumberSequence source;
    public NumberSequenceIterator(NumberSequence ns){
        source = ns;
        nextNum = ns.start;
    }
    public boolean hasNext(){
        return true;
    }
    public Integer next(){
        int ans = nextNum;
        nextNum += ns.step;
        return ans;
    }
}  // end of NumberSequenceIterator class
}  // end of Number Sequence class
```

Iterators and Iterable

- But, **the "for each" loop requires an Iterable**:

```
for (type var :  Iterable <type> ){
  … var …
}
```

eg, all Collections




```
Iterator<type > itr  = construct iterator
while (itr.hasNext() ){
  type var = itr.next();
  … var …
}
```

Using the Iterable

- Can use the iterable object in the for each loop:

```
for (int n : new NumberSequence(15, 8)){
    System.out.printf("next number is %d \n", n);
}
```

- Can use the iterator of the iterable object directly.

```
Iterator<Integer> iter = new NumberSequence(15, 8).iterator();
processFirstPage(iter);
for (int p=2; p<maxPages; p++)
   processNextPage(p, iter);
```

(passing iterator to different methods to deal with)

Creating Iterators

- Iterators are not just for Collection objects:
  - Anything that generates a sequence of values
  - Scanner
  - Pseudo Random Number generator :

```
public class NumCreator implements Iterator<Integer>{
    private int num  = 1,
    public boolean hasNext(){
      return true;
    }
    public Integer next(){
      num = (num * 92863) % 104729 + 1;
      return num;
    }
  :
Iterator<Integer> lottery = new NumCreator();
for (int i = 1; i<1000; i++)
    textArea.append(lottery.next()+ "\n");
```

Q&A

- Java has specified a “Queue” interface. (T or F)
- Java does not have any class support for “Priority Queue”. (T or F)
- peek() operation under the Queue interface will throw an exception if the queue is empty. (T or F)
- poll() operation under the Queue interface will throw an exception if the queue is empty. (T or F)
- There is an element() method under the Queue interface. (T or F)
- Iterable is an interface specification for a class that is equipped with an Iterator.
- Iterator is an interface specification for a class that can generate iterative elements.

Summary

- Queues and Priority Queues
- Classes/Interfaces that accompany collections
  - Iterator
  - Iterable

Readings

- [Mar07] Read 3.7, 3.4
- [Mar13] Read 3.7, 3.4

Iterators & Comparators

Lecture 6

Menu

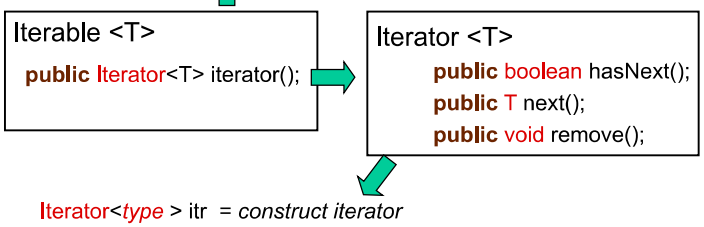
- Iterators and Iterables
- Sorting collections
- Comparators and Comparables

Iterators and Iterable

- The foreach loop requires an Iterable:

```
for (type var :  Iterable <type> ){
   … var …
}
```

eg, all Collections




```
Iterator<type > itr  = construct iterator
while (itr.hasNext() ){
   type var = itr.next();
   … var …
}
```


Creating Iterators

- Iterators are not just for Collection objects:
  - Anything that can generate a sequence of values
  - Scanner
  - Pseudo Random Number generator :

```
public class RandNumIter implements Iterator<Integer>{
    private int num  = 1,
    public boolean hasNext(){
       return true;
    }
    public Integer next(){
       num = (num * 92863) % 104729 + 1
       return num;
    }
    public void remove(){throw new
  UnsupportedOperationException();}
}
Iterator<Integer> lottery = new RandNumIter();
for (int i = 1; i<1000; i++)
```

remove(): must be defined, but doesn't need to do anything!

Creating an Iterable

- An Iterable<T> is an object that provides an Iterator<T>:
  - eg: An ArithSequence representing an infinite arithmetic sequence of numbers, with a starting number and a step size,
    eg 5, 8, 11, 14, 17,….

```
public class ArithSequence implements Iterable<Integer>{
    private int start;
    private int step;
    public ArithSequence(int start, int step){
      this.start = start;
      this.step = step;
    }
    public Iterator<Integer> iterator(){
      return new ArithSequenceIterator(this);
    }
      :
```


Creating an Iterable


```
    private class ArithSequenceIterator implements Iterator<Integer>{
        private int nextNum;
        private Ari…quence source;
        public Ar…equence as){
            sourc
            nextN
        }
        public boolean hasNext(){
            return true;
        }
        public Integer next(){
            int ans = nextNum;
            nextNum += source.step;
            return ans;
        }
        public void remove(){throw new UnsupportedOperationException();}
    }  // end of ArithSequenceIterator class
}  // end of Arithmetic Sequence class
```

Class is only accessible from inside ArithSequence

Sorting in "Natural order"

- But what order will it sort into ?
  - "natural order of the values"
- Fine for Strings, Integer, Double
  - Strings ordered alphabetically, as in a phonebook (actually a little more complicated….)
  - Integer, Double ordered by numerical value
- But what's the "natural order" of Faces in a crowd?
  - Answer:
    - Whatever you defined it to be, if you defined it.
    - There is no order if you didn't define it.
- How do you define the natural order?

Using the Iterable

- Can use the iterable object in the foreach loop:

```
for (int n : new ArithSequence(15, 8)){
    System.out.printf("next number is %d \n", n);
}
```

- Can use the iterator of the iterable object directly.

```
ArithSequence seq = new ArithSequence(15, 8));
Iterator<Integer> iter = seq.iterator();
processFirstPage(iter);
for (int p=2; p<maxPages; p++)
   processNextPage(p, iter);
```

Can pass iterator to different methods to deal with.

"Natural Ordering" & Comparable

- If a class implements the Comparable<*T* > interface
  - Objects from that class have a "natural ordering"
  - Objects can be compared **using** the compareTo() method
  - Collections.sort() can sort Lists of those objects automatically
- Comparable <*T* > is an Interface:
  - Requires
    - compareTo(*T* ob) → int
  - ob1.compareTo(ob2)
    - returns –ve if ob1 ordered before ob2
    - returns 0 if ob1 ordered with ob2
    - returns +ve if ob1 ordered after ob2

Working with Collections

- Done:
  - Declaring and Creating collections
  - Adding, removing, getting, setting, putting,….
  - Iterating through collections
    - [Iterators, Iterable, and the foreach loop]
- What else?
  - Sorting Collections
  - Implementing Collection classes

Making Face Comparable

```
public class Face implements Comparable<Face>{
        :
  public int size(){
     return (wd * ht);
   }
        :
/** Natural ordering is by size, small to large. */
   public int compareTo(Face other){
     if       ( this.size() < other.size() ) return  -1;
     else if ( this.size() > other.size() ) return  1;
     else  return 0;
   }
```

```
        :
   else if (button.equals("SmallToBig")){
     Collections.sort(crowd);
              for (Face f : crowd)
                 f.render(canvas);
    }
```


Sorting a collection

- What kinds of collections could you sort?
  - Set ?
  - Stack ?
  - Queue ?
  - List ?
  - Map ?
- How can you sort them?

Using Multiple Comparators


```
String button = event.getActionCommand();
if (button.equals("SmallToBig")){
    Collections.sort(crowd);    // use the "natural ordering" on Faces.
    render();
}
else if (button.equals("BigToSmall")){
    Collections.sort(crowd, new BigToSmallComparator());
    render();
}
else if (button.equals("LeftToRight")){
    Collections.sort(crowd, new LftToRtComparator());
    render();
}
else if (button.equals("TopToBottom")){
    Collections.sort(crowd, new TopToBotComparator());
    render();
}
```


Sorting with Comparators

- Suppose we need two different sorting orders at different times?
- Collections.sort(…) has two forms:
  - Sort by the natural order

    Collections.sort(todoList)
    - the values in todoList must be Comparable
  - Sort according to a specified order:

    Collections.sort(crowd, faceByArea)
    - faceByArea is a Comparator object for comparing the values in crowd.

Comparable	⟺	Comparator
interface for objects that can be compared		An object that can compare other objects

COMPARABLE VS COMPARATOR

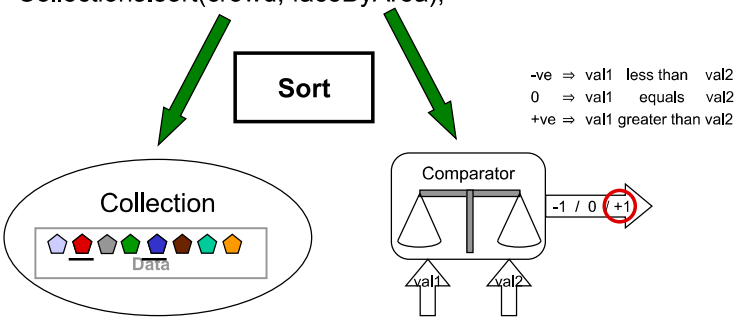
- Classes should implement the **Comparable interface** to control their ***natural ordering*.**
- Objects that implement Comparable can be sorted by **Collections.sort()** and **Arrays.sort()** and can be used as keys in a sorted map or elements in a sorted set without the need to specify a *Comparator*.

« Interface »
Comparable
+ compareTo(Object) : int

- **compareTo()** compares this object with another object and returns a *negative* integer, *zero*, or a *positive* integer as this object is *less* than, *equal* to, or *greater* than the other object.

Sorting with Comparators

- Collections.sort(crowd, faceByArea);



COMPARABLE VS COMPARATOR

- Use **Comparator** to sort objects in an order other than their natural ordering**.**

« Interface » Comparator
+ compare(Object, Object) : int

- **compare()** compares its two arguments for order, and returns a *negative* integer, *zero*, or a *positive* integer as the first argument is *less* than, *equal* to, or *greater* than the second**.**

Comparators

- Comparator <T> is an Interface
- Requires
  - **public** int compare(T o1, T o2);
    - → –ve if o1 ordered before o2
    - → 0 if o1 equals o2 [must be compatible with equals()!]
    - → +ve if o1 ordered after o2

```
/** Compares faces by the position of their top edge */
private class TopToBotComparator implements Comparator<Face>{
    public int compare(Face f1, Face f2){
        return (f1.getTop() - f2.getTop());
    }
}
```

Q&A

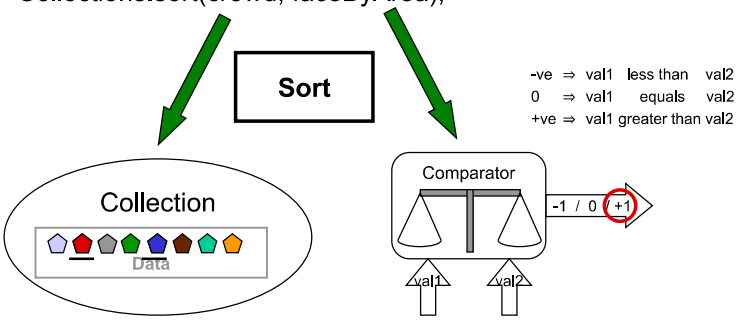
- An object defined under a comparable class will have a “natural ordering”. (T or F)
- Objects declared under a comparable class can be compared using which method?
- What is the signature of the *compareTo* method?
- Which method can be used to sort list of comparable objects?
- Comparator is an object that can compare other objects. (T or F)
- What is the signature for the *compare()* method?
- A comparable class can implement multiple comparators. (T or F)

Summary

- Iterators and Iterables
- Sorting collections
- Comparators and Comparables

Sorting with Comparators

- Collections.sort(crowd, faceByArea);



Implementing Collections

Lecture 7

Comparators

- Comparator <T> is an Interface
- Requires
  - **public** int compare(T o1, T o2);
    - → –ve if o1 ordered before o2
    - → 0 if o1 equals o2
    - → +ve if o1 ordered after o2

```
/** Compares faces by the position of their top edge */
private class TopToBotComparator implements Comparator<Face>{
   public int compare(Face f1, Face f2){
      return (f1.getTop() - f2.getTop());
   }
}
```


Menu

- Comparators
- Exceptions
- Implementing Collections:
  - Interfaces, Classes

Using Multiple Comparators

```
if (button.equals("SmallToBig")){
    Collections.sort(crowd);    // use the "natural ordering" on Faces.
    render();
}
else if (button.equals("BigToSmall")){
    Collections.sort(crowd, new BigToSmallComparator());
    render();
}
 else if (button.equals("LeftToRight")){
    Collections.sort(crowd, new LfToRtComparator());
    render();
}
 else if (button.equals("TopToBottom")){
    Collections.sort(crowd, new TopToBotComparator());
    render();
}
```


Sorting with Comparators

- Suppose we need two different sorting orders at different times?
- Collections.sort(…) has two forms:
  - Sort by the natural order

    Collections.sort(todoList)
    - the values in todoList must be comparable
  - Sort according to a specified order:

    Collections.sort(crowd, faceByArea)
    - faceByArea is a Comparator object for comparing the values in crowd.

Comparable	⟺	Comparator
interface for objects that can be compared		An object that can compare other objects

Types of Exceptions

- Lots of kinds of exceptions

Exceptions

AclNotFoundException, ActivationException, AlreadyBoundException, ApplicationException, AWTException, BackingStoreException, BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException, BrokenBarrierException, CertificateException, ClassNotFoundException, CloneNotSupportedException, DataFormatException, DatatypeConfigurationException, DestroyFailedException, ExecutionException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSException, IllegalAccessException, IllegalClassFormatException, InstantiationException, InterruptedException, IntrospectionException, InvalidApplicationException, InvalidMidiDataException, InvalidPreferencesFormatException, InvalidTargetObjectTypeException, InvocationTargetException, **IOException**, JMException, LastOwnerException, LineUnavailableException, MidiUnavailableException, MimeTypeParseException, NamingException, NoninvertibleTransformException, NoSuchFieldException, **NoSuchMethodException**, NotBoundException, NotOwnerException, ParseException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, RefreshFailedException, RemarshalException, **RuntimeException**, SAXException, ServerNotActiveException, SQLException, TimeoutExc… Unsupported… Unsupported… XPathExcep…

RunTimeExceptions:

AnnotationTypeMismatchException, ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMMException, ConcurrentModificationException, DOMException, EmptyStackException, EnumConstantNotPresentException, EventException, IllegalArgumentException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, ImagingOpException, IncompleteAnnotationException, **IndexOutOfBoundsException**, JMRuntimeException, LSException, MalformedParameterizedTypeException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, **NullPointerException**, ProfileDataException, ProviderException, RasterFormatException, RejectedExecutionException, SecurityException, SystemException, TypeNotPresentException, UndeclaredThrowableException, UnmodifiableSetException, **UnsupportedOperationException**

Exceptions

- What should a method do when something goes wrong?
  - Throw an exception!
  - An exception is an event that
    - occurs during the execution of a program
    - disrupts the normal flow of instructions
- An Exception is a "non-local exit" (what does that mean?)
  - Error of some kind, eg
    - file doesn't exist,
    - dividing by zero
    - calling a method on the null object
    - calling an undefined method
  - An exceptional situation, eg
    - text in file doesn't have the expected format
    - user enters an unexpected answer
- How do you deal with Exceptions?
  - **throwing exceptions**
  - **handling (catching) exceptions**

May be thrown by the JVM when executing code.

May be thrown by code from the java library

May be deliberately thrown by your code.

Types of Exceptions

- Lots of exceptions
- **RuntimeExceptions don't have to be handled**:
  - An uncaught RuntimeException will result in an error message
  - You can catch them if you want.
- **Other Exceptions must be handled**:
  - eg IOException

    (which is why we always used a ***try…catch*** when opening files).
- An exception object contains information:

  the state of the execution stack

  any additional information specific to the exception

Catching exceptions

- exceptions thrown in a **try ... catch** can be "caught":

```
try {
    … code that might throw an exception
}
catch (⟨ExceptionType1⟩ e1) { …actions to do in this case….}
catch (⟨ExceptionType2⟩ e2) { …actions to do in this case….}
catch (⟨ExceptionType3⟩ e3) {    System.out.println(e.getMessage());
                                 e.printStackTrace();}
```

Exception object, containing information about the state

- Meaning:
  - **If an exception is thrown, jump to catch claus**
  - **Match exception type against each catch clau**
  - **If caught, perform the actions, and jump to code after all the catch clauses**

    The actions may use information in the exception object.

Informative action for an error condition that the program can't deal with itself

Throwing exceptions

- The ***throw*** statement causes an exception.
  - eg, a drawing file has a line with an unknown shape:

```
if (name.equals("oval"))                shapes.add(new Oval(file));
else if (name.equals("rectangle"))      shapes.add(new Rectangle(file));
else if (name.equals("line"))           shapes.add(new Line(file));
        :
else if (name.equals("polyline"))       shapes.add(new PolyLine(file));
else
   throw new RuntimeException("Unknown shape in drawing file");
render();
```

General RuntimeException object.
- Could use a more specific one
- Could define our own type of exception

  - eg, defining a method that shouldn't be… such as adding an element to an immutable List

```
public void add(E element){
   throws new UnsupportedOperationException("Immutable List");
}
```


Catching exceptions

```
String filename = "/nosuchdir/myfilename";
try {
    // Create the file
    new File(filename).createNewFile();
} catch (IOException e) {
    // Print out the exception that occurred
    System.out.println("Unable to create "+filename+": "+e.getMessage());
}
// Execution continues here after the IOException handler is executed
```

Here's the output:

Unable to create /nosuchdir/myfilename: The system cannot find the path specified

Defining List: Type Variables


```
public interface List <E> extends Collection<E>{
    public int size ();
    public E get (int index);
    public E set (int index, E elem);
    public boolean add (E elem);
    public boolean add (int index, E elem);
    public E remove (int index);
    public void remove (Object ob);
    public Iterator<E> iterator();
    public void clear();
    public boolean contains (Object ob);
    public int indexOf (Object ob);
    public boolean addAll (Collection<E> c);
      :
```

E is a type variable

E will be bound to an actual type when a list is declared:

```
private List <String> items;
```

Throwing exceptions


```
public Object pop() {
        Object obj;
        if (size == 0) {
                throw new EmptyStackException();
        }
… …
}
```

Defining ArrayList

- Design the data structures to store the values
  - array of items
  - count
- Define the fields and constructors

```
public class ArrayList <E> implements List<E> {
    private E [ ] data;
    private int count;
```

- Define all the methods specified in the List interface

Implementing Collections

- Part 1: using collections
  - ✔ Different types of collections
  - ✔ Alternative data structures and algorithms
    - Cost and efficiency
    - Testing
    - Binary search algorithm
    - Sorting algorithms
- Part 2: implementing collections
  - ✔ What it means to implement a Collection Interface
  - ✔ The Java collections library
  - ✔ Interfaces for List, Set
  - ✔ Using "for each" loop, Iterable, and Iterators
  - ✔ Using Sort, Comparable, and Comparators
  - ✔ Throwing exceptions

Q&A

- A method will do what when something goes wrong?
- An exception provides a local exit when something goes wrong. (T or F)
- Name 3 cases when an exception can be thrown.
- What do we do with exceptions?
- Exceptions can be caught using what Java statement?
- Does exception have a type?
- After the exception handler finishes its work, what will the program do next?
- Can exception handler use information in the exception object?
- What does "**e.getMessage**()" do where **e** is an exception object?

Interfaces and Classes

- Interface
  - specifies **type**
  - defines **method headers** only
- List <*E* >
  - Specifies sequence of *E*
  - size, add_1, get, set, $remove_1$
  - add_2, addAll, clear, contains, containsAll, equals, hashCode, isEmpty, indexOf, lastIndexOf, iterator, listIterator, $remove_2$, removeAll,retainAll, subList, toArray.
- ArrayList <*E* >
  - implements List <*E* >
  - defines fields with array of <*E* > and count
  - defines size(), add(), …
  - defines …
  - defines constructors
- Class
  - implements Interface
  - defines **fields** for the data structures to hold the data.
  - defines **method bodies**
  - defines **constructors**

Q&A

- Name 3 common Java exceptions.
- RuntimeExceptions doesn't have to be handled. (T or F)
- IOException doesn't have to be handled. (T or F)
- Which Java statement can cause an exception?
- What happens when we try adding an element to an immutable List?
- Does a Java Interface provide a 'constructor'?
- ArrayList implements which Interface?
- **public interface** List <E> extends which Java Interface?

Summary

- Comparators
- Exceptions
- Implementing Collections:
  - Interfaces, Classes

Implementing Collections II

Lecture 8

Defining ArrayList: too many methods

- Problem: There are a lot of methods in List that need to be defined in ArrayList, and many are complicated.
- But, many could be defined in terms of a few basic methods: (size, add, get, set, remove)
  - eg,

```
public boolean addAll(Collection<E> other){
  for (E item : other)
      add(item);
  }
```

- Solution: **an Abstract class**
  - Defines the complex methods in terms of the basic methods
  - Leaves the basic methods "abstract" (no body)
  - classes implementing List can extend the abstract class.

Interfaces and Classes

• Interface • specifies type • defines method headers	• List <E> • Specifies sequence of E *type*
• Abstract Class • implements Interface • defines some methods • leaves other methods "abstract"	• AbstractList <E> • implements List <E> • defines array of <E> • defines addAll, subList, … • add, set, get, … are left **abstract**
• Class • extends Abstract Class • defines data structures • defines basic methods • defines constructors	• ArrayList <E> • extends AbstractList • implements fields & constructor • implements add, get, …

Summary

- Implementing Collections:
  - Interfaces, Abstract Classes, Classes

AbstractList

```
public abstract class AbstractList <E> implements List<E>{
```

No constructor or fields

```
  public abstract int size();
```

declared abstract - must be defined in a real class

```
  public boolean isEmpty(){
    return (size() == 0);
  }
```

defined in terms of size()

```
  public abstract E get(int index);
```

declared abstract - must be defined in a real class

```
  public void add(int index, E element){
    throws new UnsupportedOperationException();
  }
```

defined to throw exception should be defined in a real class

```
  public boolean add(E element){
    add(size(), element);
```

defined in terms of other add

Defining ArrayList

- Design the data structures to store the values
  - array of items
  - count
- Define the fields and constructors

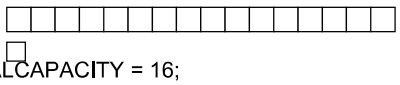
```
public class ArrayList <E> implements List<E> {
    private E [ ] data;
    private int count;
```

- Define all the methods specified in the List interface

size()	add(E o)	add(int index, E element)	contains(Object o)	get(int index)
isEmpty()	clear()	set(int index, E element)	indexOf(Object o)	remove(int index)
remove(Object o)		lastIndexOf(Object o)	iterator()	
equals(Object o)		hashCode()	listIterator()	… … …

ArrayList: fields and constructor


```
public class ArrayList <E> extends AbstractList <E> {
    private E[ ] data;
    private int count=0;
    private static int INITIALCAPACITY = 16;

 public ArrayList(){
    data =      (E[ ]) new Object[INITIALCAPACITY];
  }
```



- Can't use type variables as array constructors!!!!
- Must Create as Object[ ] and cast to E[ ]
- The compiler will return a warning!
  - " …. uses unchecked or unsafe operations" (why it is 'unchecked'?)

AbstractList continued


```
public boolean contains(Object ob){
  for (int i = 0; i<size(); i++)
    if (get(i).equals(ob) ) return true;
  return false;
}

public void clear(){
  while (size() > 0)
    remove(0);
}
   :
   :
```

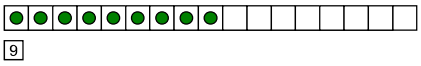
defined in terms of size and get

defined in terms of size and remove

- AbstractList cannot be instantiated.

ArrayList: size, isEmpty


```
public class ArrayList <E> extends AbstractList <E> {
    private E[ ] data;
    private int count=0;
        :
/** Returns number of elements in collection as integer */
    public int size () {
       return count;
    }

/** Returns true if this set contains no elements. */
    public boolean isEmpty(){
        return count==0;
    }
```



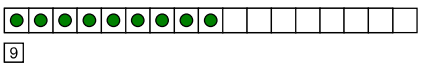
ArrayList extends AbstractList


```
public class ArrayList <E> extends AbstractList<E>{
  // fields to hold the data:
     need an array for the items and an int for the count.
  // constructor(s)
     Initialise the array
  // definitions of basic methods not defined in AbstractList
     size()
     get(index)
     set(index, value)
     remove(index)
     add(index, value)
     iterator()
  //  (other methods are inherited from AbstractList )
  // definition of the Iterator class
```

Can give other methods to override the inherited methods, if it would be more efficient

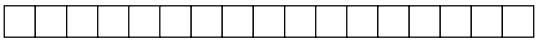
ArrayList: get


```
public class ArrayList <E> extends AbstractList<E> {
    private E[ ] data;
    private int count=0;
        :
 /**Returns the value at the specified index.
  * Throws an IndexOutOfBoundsException is index is out of
  bounds */
  public E get(int index){
    if (index < 0 || index >= count)
    throw new IndexOutOfBoundsException();
    return data[index];
  }
```



Implementing ArrayList

- Data structure:

  data 

  count 

- size:
  - returns the value of count
- get and set:
  - check if within bounds, and
  - access the appropriate value in the array
- add(index, elem):
  - check if within bounds, (0..size)
  - move other items up, and insert
  - as long as there is room in the array !

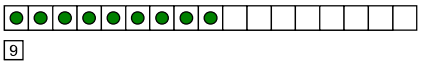
ArrayList: add


```
public class ArrayList <E> extends AbstractList <E> {
   private E[] data;
   private int count=0;
       :

   /** Adds the specified element at the specified index */
   public void add(int index, E item){
      if (index < 0 || index >= count)
         throw new IndexOutOfBoundsException();
      for (int i=count; i > index; i--)      ←move items up
         data[i]=data[i-1];
      data[index]=item;                      ←insert
      count++;                               ←increment
   }
```

- What's wrong???

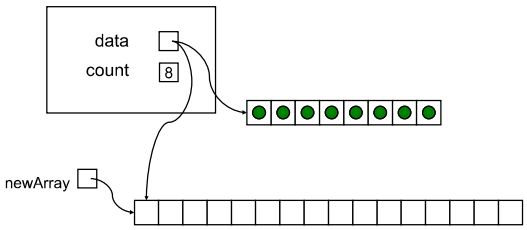
ArrayList: add (fixed)


```
public class ArrayList <E> extends AbstractList <E> {
   private E[] data;
   private int count=0;
       :

   /** Adds the specified element at the specified index.*/
   public void add(int index, E item){
      if (index < 0 || index > count)      ←can add at end?
         throw new IndexOutOfBoundsException();
      ensureCapacity();                    ←make room
      for (int i=count; i > index; i--)    ←move items up
         data[i]=data[i-1];
      data[index]=item;                    ←insert
      count++;                             ←increment
   }
```



Increasing Capacity

- ensureCapacity():



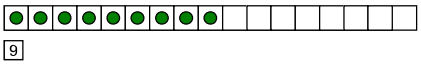
- How big should the new array be?

ArrayList: set


```
public class ArrayList <E> extends AbstractList<E> {
   private E[ ] data;
   private int count=0;
       :

   /**Replaces the value at the specified index by the specified value
    * Returns the old value.
    * Throws an IndexOutOfBoundsException if index is out of
    bounds */
   public E set(int index, E value){
      if (index < 0 || index >= count)
         throw new IndexOutOfBoundsException();
      E ans = data[index];
      data[index] = value;
      return ans;
   }
```

ArrayList: remove


```
public class ArrayList <E> extends AbstractList <E> {
   private E[] data;
   private int count=0;
       :

   /** Removes the element at the specified index, and returns it.
    * Throws an IndexOutOfBoundsException if index is out of
    bounds */
   public E remove (int index){
      if (index < 0 || index >= count)
        throw new IndexOutOfBoundsException();
      E ans = data[index];                 ←remember
      for (int i=index; i< count; i++)     ←move items down
        data[i]=data[i+1];
      count--;                             ←decrement
      return ans;                          ←return
   }
```



problem?

ArrayList: remove (fixed)


```
public class ArrayList <E> extends AbstractList <E> {
   private E[] data;
   private int count=0;
       :

   /** Removes the element at the specified index, and returns it.
    * Throws an IndexOutOfBoundsException if index is out of
    bounds */ public E remove (int index){
      if (index < 0 || index >= count)
        throw new IndexOutOfBoundsException();
      E ans = data[index];                 ←remember
      for (int i=index+1; i< count; i++)   ←move items down
        data[i-1]=data[i];
      count--;                             ←decrement
      data[count] = null;                  ←delete previous last element
      return ans;                          ←return
   }
```

Menu

- Implementing Collections:
  - Interfaces, Abstract Classes, Classes

ArrayList: ensureCapacity

```
/**Ensure data array has sufficient number of elements
   * to add a new element */
private void ensureCapacity () {
  if (count < data.length) return;        ← room already
  E [ ] newArray = (E[]) (new Object[data.length+INITIALCAPACITY]);
    for (int i = 0; i < count; i++)       ← copy to new array
        newArray[i] = data[i];
    data = newArray;                      ← replace (replace what?)
}
```

OR

```
private void ensureCapacity () {
    if (count < data.length) return;      ← room already
    E [ ] newArray = (E[ ]) (new Object[data.length * 2]);
    for (int i = 0; i < count; i++)       ← copy to new array
        newArray[i] = data[i];
    data = newArray;                      ← replace
}
```

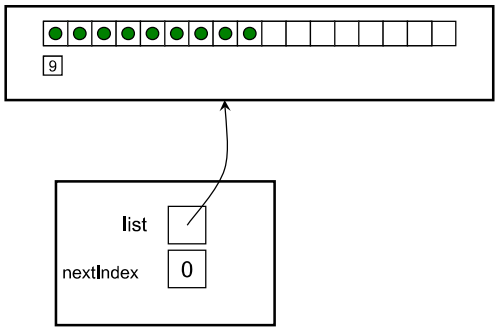

ArrayList: What else?

- iterator():
  - defining an iterator for ArrayList.
- Cost:
  - What is the cost (time) of adding or removing an item?
  - How expensive is it to increase the size?
  - How should we increase the size?

Q&A

- What are the key features of an abstract class?
- Can an abstract class be instantiated?
- Abstract methods can be defined within a class to save implementation efforts. (T or F)
- What are the key issues of implementation when we remove an element from an ArrayList?
- What are the key issues of implementation when we add an element from an ArrayList?

Iterator



More on Implementing Collections III

Lecture 9

ArrayList: iterator


```
/** Returns an iterator over the elements in the List */
public Iterator <E> iterator(){
    return new ArrayListIterator<E>(this);
}

/** Definition of the iterator for an ArrayList
  * Defined inside the ArrayList class, and can therefore  access
    *   the private fields of an ArrayList object.  */
   private class ArrayListIterator <E> implements Iterator <E>{
    // fields to store state
    // constructor
    // hasNext(),
    // next(),
    // remove()     (an optional operation for Iterators)
  }
```

Menu

- Implementing ArrayList:
  - Iterators
  - Cost of adding and removing

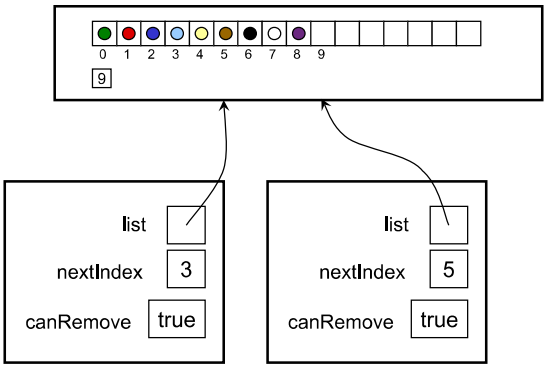
Iterator


```
private class ArrayListIterator <E> implements Iterator <E>{

    private ArrayList<E> list;// reference to the list it is iterating down
    private int nextIndex = 0;  // the index of the next value to return
    private boolean canRemove = false;
                // to disallow the remove operation initially

    /** Constructor  */
    private ArrayListIterator (ArrayList <E> list) {
      this.list = list;
     }

    /** Return true if iterator has at least one more element  */
    public boolean hasNext () {
      return (nextIndex < list.count);
    }
```

ArrayList: What else?

- iterator():
  - defining an iterator for ArrayList.
- Cost:
  - What is the cost (time) of adding or removing an item?
  - How expensive is it to increase the size?
  - How do we increase the size?

Multiple Iterators



Multiple Iterators: Summary

- Each iterator keeps track of its own position in the List
- Removing the last item returned is possible, but
- The implementation is not smart, and may be corrupted if any changes are made to the ArrayList that it is iterating down.
- Note that because it is an inner class, it has access to the ArrayList's private fields.

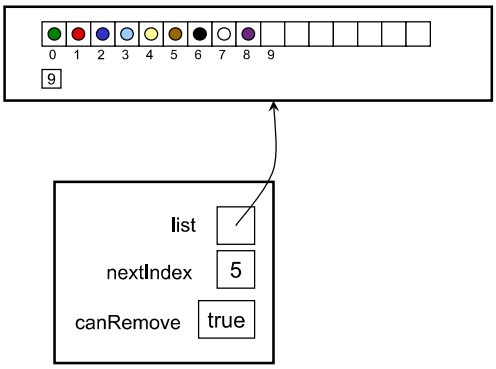
ArrayList: Cost

- What's the cost of get, set, remove, add?
- How should we implement ensureCapacity() ?
- How do you measure the cost of operations on collections?
- What is the "cost" of an algorithm or a program?
- Number of steps required if the list contains *n* items:
  - get:
  - set:
  - remove:
  - add:

Iterator: next, remove

```
/** Return next element in the List */
public E next () {
    if (nextIndex >= list.count)  throw new NoSuchElementException();
          return list.get(nextIndex++); ← increment and return
        }


/** Remove from the list the last element returned by the iterator.
  *  Can only be called once per call to next.  */
public void remove(){
     throw new UnsupportedOperationException();
  }
```


Iterator, with remove



Iterator: next, remove

```
/** Return next element in the List */
  public E next () {
      if (nextIndex >= list.count)  throw new
  NoSuchElementException();
      canRemove = true;              ← for the remove method
      return list.get(nextIndex++);  ← increment and return
    }

/** Remove from the list the last element returned by the iterator.
  Can only be called once per call to next.  */
  public void remove(){
      if ( ! canRemove )   throw new IllegalStateException();
      canRemove = false;             ← can only remove once
      nextIndex--;          ← put counter back to last item
      list.remove(nextIndex);        ← remove last item
    }
```

what if we don't put counter back to last item?

After removal, nextIndex will be pointing at which item?

Q&A

- remove() is compulsory in Iterator implementation. (T or F)
- How does ArrayList make use of the 'type parameter' in its implementation?
- Which element will be removed by ArrayList.remove()?
- What does ArrayList.next() check before returning the next element in the list?
- How does ArrayList.remove() ensure only 1 element can be removed after each call to next()?
- What can happen if 2 or more Iterators running concurrently under the same ArrayList? Name 2 scenarios.

Summary

- Implementing ArrayList:
  - Iterators
  - Costs of adding and removing

Readings

- [Mar07] Read 3.4
- [Mar13] Read 3.4

Analysing Costs

Lecture 10

Menu

- Cost of operations and measuring efficiency
- ArrayList: retrieve, remove, add
- ArraySet: contains, remove, add

Analysing Costs

How can we determine the costs of a program?

- **Time**:
  - Run the **program** and count the milliseconds/minutes/days.
  - Count the number of steps/operations the **algorithm** will take.
- **Space**:
  - Measure the amount of memory the **program** occupies
  - Count the number of elementary data items the **algorithm** stores.
- Question:
  - Programs or Algorithms?
- Answer:
  - Both
    - programs: benchmarking
    - algorithms: analysis

Benchmarking: program cost

- Measure:
  - actual programs
  - on real machines
  - on specific input
  - measure elapsed time
    - **System.currentTimeMillis**()
      → time from system clock in milliseconds (long)
  - measure real memory usage
- Problems:
  - what input ⇒ choose test sets carefully
    use large data sets
    don't include user input
  - other users/processes ⇒ minimise
    average over many runs
  - which computer? ⇒ specify details

Analysis: Algorithm complexity

- Abstract away from the details of
  - the hardware
  - the operating system
  - the programming language
  - the compiler
  - the program
  - the specific input
- Measure number of "steps" as a function of the data size.
  - worst case (easier)
  - average case (harder)
  - best case (easy, but useless)
- Construct an expression for the number of steps:
  - cost = 3.47 n^2 - 67n + 53 steps
  - cost = 3n log(n) - 5n + 6 steps

  simplified into terms of different powers/functions of n

Analysis: Asymptotic Notation

- We only care about the cost when it is large
  - ⇒ drop the lower order terms
    (the ones that will be insignificant with large n)

    cost = 3.47 n^2 + ... steps

    cost = 3n log(n) + … steps
- We don't care about the constant factors
  - Actual constant will depend on the hardware

  ⇒ Drop the constant factors

  cost ∝ n^2 + … steps

  cost ∝ n log(n) + … steps
- "Asymptotic cost", or "big-O" cost.
  - describes how cost grows with input size
  - cost is O(1) : fixed cost
  - cost is O(n): grows with n

Which one is the fastest?

Usually we are only interested in the ***asymptotic*** time complexity, i.e., when n is large

$O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

Big Oh Notation

- A notation for describing efficiency of computer algorithms
- assuming a 100-MHz clock, N = 1024k = 2^{20}

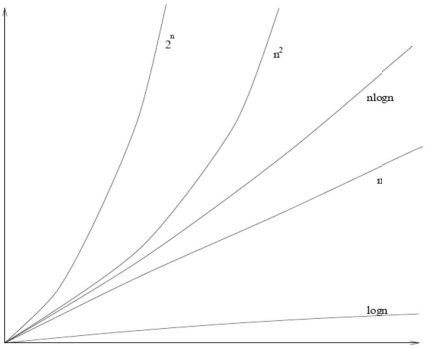
- O(1) - constant time, 10 ns

- O(log N) - logarithmic time, 200 ns
- O(N) - linear time, 10.5ms
- O(N log N) - n log n time, 210 ms

- O(N ^ 2) - quadratic time, 3.05 hours
- O(N ^ 3) - cubic time, 365 years

- O(2 ^ N) - exponential, 10 ^ (10 ^ 5) years

Typical Costs



Typical Costs in Big 'O'

If data/input is size n		How does it grow.
O(1)	*"constant"*	cost is independent of *n*
O(log(n))	*"logarithmic"*	size x 10 → add a little (log(10)) to the cost
O(n)	*"linear"*	size x 10 → 10 x the cost
O(n log(n))	*"en-log-en"*	size x 10 → bit more than 10 x
$O(n^2)$	*"quadratic"*	size x 10 → 100 * the cost
$O(n^3)$	*"cubic"*	size x 10 → 1000 * the cost
:		
:		
$O(2^n)$	*"exponential"*	adding one to size -> doubles the cost => You don't want to run this algorithm!
O(n!)	*"factorial"*	adding one to size -> n the cost => You **definitely** don't want this algorithm!

Problem: What is a "step"?

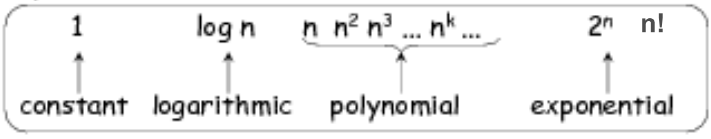
- Count
  - actions in the innermost loop (happen the most times)
  - actions that happen every time round (not inside an "if")
  - actions involving "data values" (rather than indexes)
  - representative actions (don't need every action)

```
public E remove (int index){
    if (index < 0 || index >= count) throw new ….Exception();
    E ans = data[index];
    for (int i=index+1; i< count; i++)      ← in the innermost loop
      data[i-1]=data[i];                    ←Key Step
     count--;
    data[count] = null;
    return ans;
  }
```

Each for loop:
1 comparison: i<count
1 addition: i++
1 data retrieval: data[i]
1 subtraction: i-1
1 memory store: data[i-1]=data[i]

Hierarchy of functions

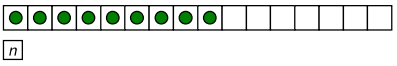
- We can define a hierarchy of functions each having a **greater** order of magnitude than its predecessor:



- We can further refine the hierarchy by inserting **n log n** between **n** and **n²**, **n² log n** between **n²** and **n³**, and so on.

ArrayList: add at end

- Cost of add(value):
  - what's the key step?
  - worst case:



  - average case:

```
public void add (E item){
   ensureCapacity();
   data[count++] = item;
}
private void ensureCapacity () {
   if (count < data.length) return;
   E [ ] newArray = (E[ ]) (new Object[data.length * 2]);
   for (int i = 0; i < count; i++)
      newArray[i] = data[i];
   data = newArray;
 }
```

What's a step: Pragmatics

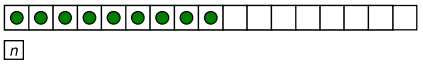
- Count the most expensive actions:
  - Adding 2 numbers is cheap
  - Raising to a power is not so cheap
  - Comparing 2 strings *may* be expensive
  - Reading a line from a file *may* be very expensive
  - Waiting for input from a user or another program may take forever…
- Sometimes we need to know how the underlying operations are implemented in the computer to analyse well

ArrayList: add at end

- Average case:
  - average over all possible states and input. OR
  - amortised cost over time.
- Amortised cost: total cost of adding n items $\div$ n :
  - first 10: cost = 1 each total = 10
  - 11th: cost = 10+1 total = 21
  - 12-20: cost = 1 each total = 30
  - 21st: cost = 20+1 total = 51
  - 22-40: cost = 1 each total = 70
  - 41st: cost = 40+1 total = 111
  - 42-80: cost = 1 each total = 150
  - :
  - - n total =
- Amortised cost (per item) =

ArrayList: get, set, remove

- Assume List contains n items.
- Cost of get and set:
  - best, worst, average: O(1)
  - ⇒ constant number of steps, regardless of n
- Cost of Remove:



  - worst case:
    - what is the worst case?
    - how many steps?
  - average case:
    - what is the average case?
    - how many steps?

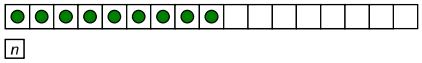
ArrayList costs: Summary

• get	O(1)	
• set	O(1)	
• remove	O(n)	
• add (at i)	O(n)	(worst and average)
• add (at end)	O(1)	(average)
	O(n)	(worst)
	O(1)	(amortised average)

- Question:
  - what would the amortised cost be if the array size is increased by 10 each time?

ArrayList: add

- Cost of add(index, value):
  - what's the key step?
  - worst case:



  - average case:

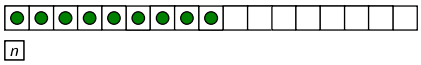
```
public void add(int index, E item){
  if (index<0 || index>count) throw new IndexOutOfBoundsException();
  ensureCapacity();
  for (int i=count; i > index; i--)
       data[i]=data[i-1];
  data[index]=item;
  count++;
}
```

Readings

- [Mar07] Read 2.2, 2.3, 2.4
- [Mar13] Read 2.2, 2.3, 2.4

Cost of ArraySet

- ArraySet uses same data structure as ArrayList
  - does not need to keep items in order
- Operations are:



  - contains(item)
  - add(item) ← always add at the end
  - remove(item) ← don't need to shift down – just move last item down
- What are the costs?
  - contains: O(n)
  - remove:
  - add: O(1) O(n)

Q&A

- O(log(n)) < O(sqrt(n)) (T or F)
- O(n^n) < O(n!) (T or F)
- O(2^n) < O(n^n) (T or F)
- When analysing the cost of an algorithm, loop usually is the focus. (T or F)
- Which of the following operations is more expensive?
  - Reading a line from a file
  - Reading a line from a user
- Worst case cost analysis is usually more difficult than average cost analysis. (T or F)

Summary

- Cost of operations and measuring efficiency
- ArrayList: retrieve, remove, add
- ArraySet: contains, remove, add

Recursion

Lecture 11

Menu

- recursive functions
- factorial function
- fibonacci function
- recursion vs iteration

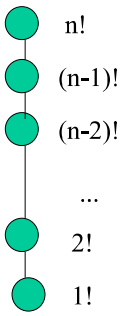
recursive functions

- A recursive function is a function that calls itself directly or indirectly
- Related to recursive problem solving: binary tree traversal (**divide & conquer**)
- The function knows how to solve a base case (**stopping rule**)
- A recursion step is needed to divide the problem into sub-problems (**key step**)
- Need to check for **termination**

factorial

- 1! = 1
- 5! = 5 * 4 * 3 * 2 * 1 or
- 5! = 5 * 4!

- A recursive definition:
- n! = n * (n-1)!



recursion tree for n!

factorial function fac

- base case:
  if (number <= 1) return 1;

- recursive step:
  return (number * fac (number - 1));


```
#include <stdio.h>
long fac(long);

main()
{
 int i;
 for (i=1; i <=5; i++)
     printf("%d! = %ld\n", i, fac(i));
 return 0;
}
```

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

```
long fac(long number)
{
if (number <= 1)
   return 1;
else
 return ( number * fac (number - 1));
}
```

```
/* Recursive definition of function fibonacci */

long fibonacci(long n)
{
  if (n == 0 || n == 1)
    return n;
  else
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```


How does fac work?

```
long fac(long number)
{
if (number <= 1)
   return 1;
else
 return ( number * fac (number - 1));
}
```

fac(1) --> if (1 <= 1) return 1; --> 1
fac(2) --> else return (2 * fac (2 - 1));
--> return (2 * fac(1));
--> return (2* 1);


```
#include <stdio.h>

long fibonacci(long);

main()
{
  long result, number;

  printf("Enter an integer: ");
  scanf("%ld", &number);
  result = fibonacci(number);
  printf("Fibonacci(%ld) = %ld\n", number, result);
  return 0;
}
```

```
Enter an integer: 0
Fibonacci(0) = 0
Enter an integer: 1
Fibonacci(1) = 1
Enter an integer: 2
Fibonacci(2) = 1
Enter an integer: 3
Fibonacci(3) = 2
Enter an integer: 4
Fibonacci(4) = 3
```



```
long fac(long number)
{
if (number <= 1)
   return 1;
else
 return ( number * fac (number - 1));
}
```

fac(3) --> else return (3 * fac (3 - 1));
--> return (3 * fac (2));
--> return (3* (2 * fac(1)));
--> return (3* 2*1);

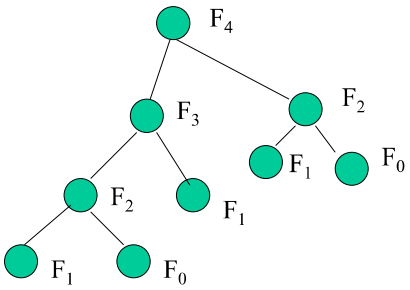
How does fibonacci work?

fibonacci (0) --> if (0 == 0 || 0 == 1) return 0;
fibonacci (1) --> if (1 == 0 || 1 == 1) return 1;
fibonacci (2) --> else return fibonacci(2 - 1) + fibonacci(2 - 2);
--> fibonacci(1) + fibonacci(0);
--> 1 + 0
fibonacci (3) --> else return fibonacci(3 - 1) + fibonacci(3 - 2);
--> return fibonacci(2) + fibonacci(1);
fibonacci (4) --> else return fibonacci(4 - 1) + fibonacci(4 - 2);
--> return fibonacci(3) + fibonacci(2);
-->{ fibonacci(3 - 1) + fibonacci(3 - 2) }+
{fibonacci(2 - 1) + fibonacci(2 - 2) }+
--> ...

fibonacci function

- fibonacci series: 0,1,1,2,3,5,8,13,21, **...**
- **base case:**
- fibonacci (0) = 0
- fibonacci (1) = 1
- **recursive step:**
- fibonacci (n) = fibonacci (n-1) + fibonacci (n-2)

Recursion tree for fibonacci(4)



Q&A

- What is the key step in designing a recursive function?
- Every recursive function can be rewritten as an iterative function. (T or F)

Recursion vs iteration

- iteration: while, for
- recursion uses a selection structure & function calls;
- iteration uses an iterative structure
- recursion & iteration each involves a termination test:
  - recursion ends when a base case recognized
  - iteration ends when the continuation condition fails
- every recursive function can be rewritten as an iterative function
- iteration: efficient but not easy to design
- recursion: slow (cost memory & processor power) but elegant

Summary

- recursive functions
- factorial function
- fibonacci function
- recursion vs iteration

EXERCISE

- Design a recursive function to solve the following problem: sum up a given array a[0]..a[m-1] & return the sum to the caller
- int sum_up(a, n) // n: number of array elements to sum up

Readings

- [Mar07] Read 1.3
- [Mar13] Read 1.3

Linked Structures

Lecture 12

Testing Collection Implementations

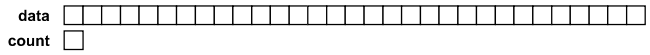
- Write a **test method**
  - As part of the class or as a separate testing class
  - Should test all the operations
  - Should test normal and extreme cases
- Good practice:
  - write it first (**black box testing**)
  - implement the collection
  - extend the test method to cover the special cases of the implementation (**white box testing**)
- Nicer design uses **tests/assertions**:
  - check that the code does the right thing
  - only report when there is a problem or error
- May take longer to write than the collection code!

Menu

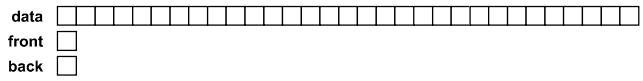
- Testing collection implementations
- Queues
- Motivation for linked lists
- Linked structures for implementing Collections

Queues

- We haven't talked about implementing queues
- Simplest array implementations are slow:



- Efficient array implementation
  - "wrapping around"
  - O(1) for add ("offer") and remove ("poll")



  - Have to be careful in ensureCapacity()
    - How do we know array is full?

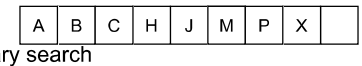
Where have we been?

Implementing Collections with arrays:

- ArrayList: O(n) to add/remove, except at end
- Stack: O(1)
- ArraySet: O(n) (add/find/remove) (⇐ cost of searching)

How can we insert fast?

- Fast *access* in array

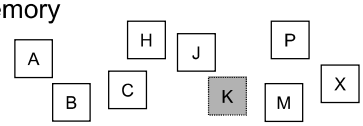


  ⇒ items must be sorted, to use binary search

- Arrays stored in contiguous chunks of memory.

  ⇒ inserting new items will be slow

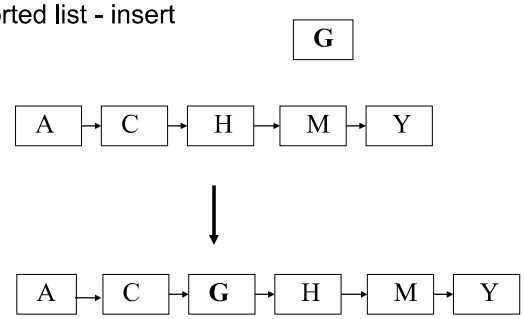
- Can't insert fast with an array!
- To insert fast, we need each item to be in its own chunk of memory



- But, how do we keep track of the order?

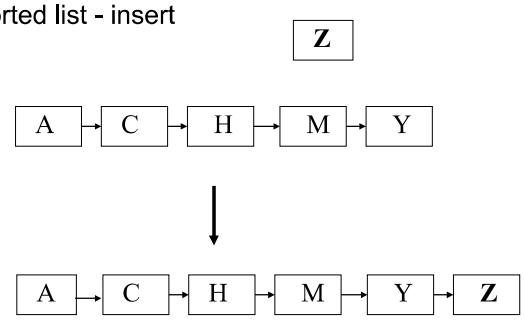
linked lists - insertion

- sorted list - insert

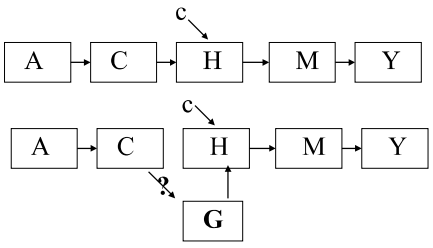


linked lists - insertion

- sorted list - insert



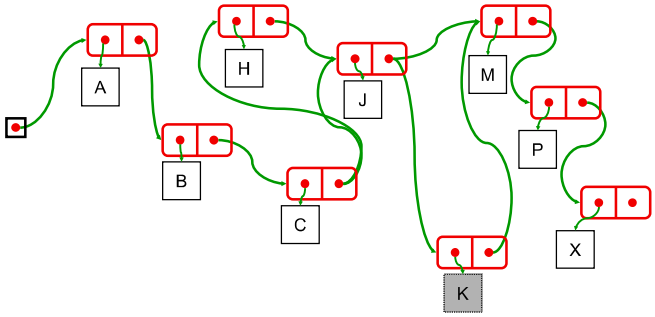
Insert in action



Problem: lost track of C when 'H' visited, cannot redirect nextPtr in C to point to 'G' Solution ?

Linked Structures

- Put each value in an object with a field for a link to the next



- Traverse the list by following the links
- Insert by changing links
- Remove by changing links

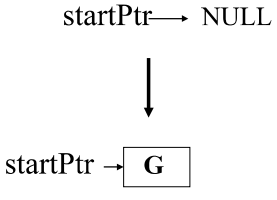
Linked lists

- collections of data in a row

- insertions & deletions anywhere in the list
- other operations: search for an element in the list, print the list, test if list empty, etc.

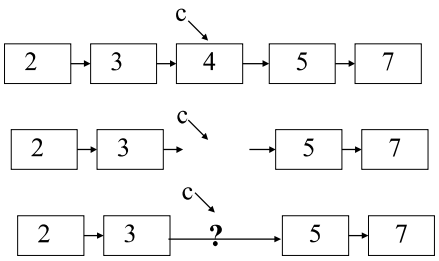
linked lists - insertion

- sorted list (empty) - insert



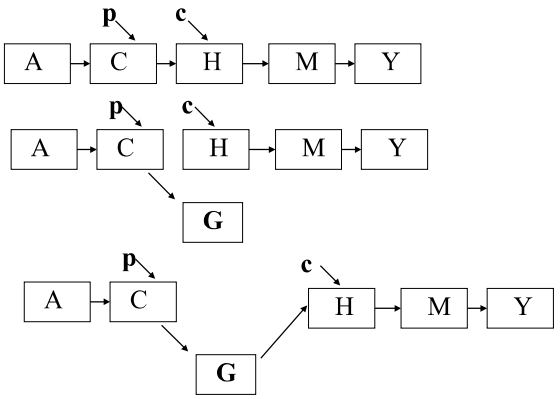
Delete in action

4



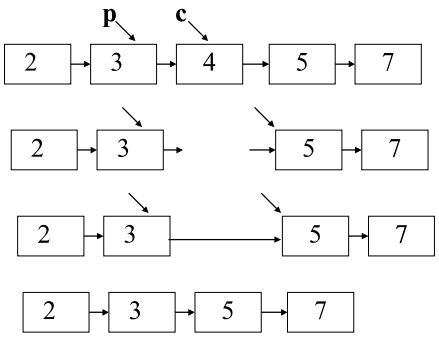
Problem: need to redirect nextPtr of 3 to 5, already lose track of 3 when c moved to 4
Solution ?

Insert in action



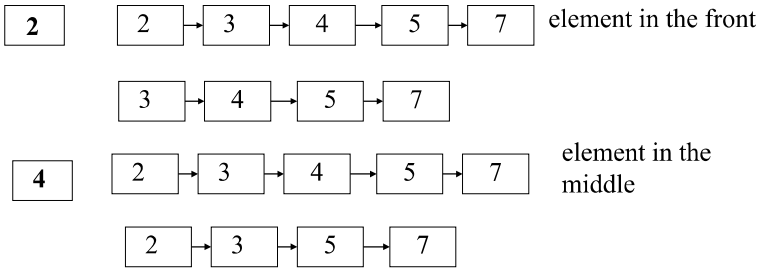
Delete in action

4



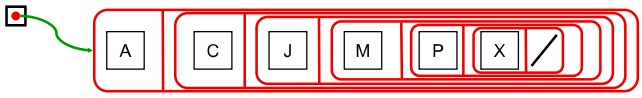
linked lists - search for deletion

- search & then delete

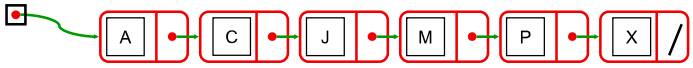


Linked List Structures – Alternative Views

- Can be drawn with Nodes inside Nodes:



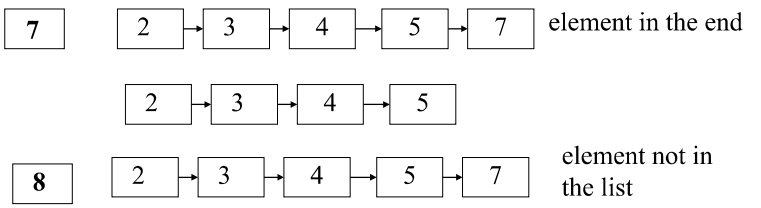
- "Pointers" are better:
  - Each node contains a reference to the next node
  - reference = memory location of / pointer to object



- Can view a node in two ways:
  - an object containing two fields
  - the head of a linked list of values

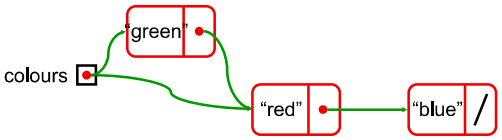
linked lists - search for deletion

- search & then delete



Using Linked Nodes

```
LinkedNode<String> colours = new LinkedNode<String> ("red", null);

colours.setNext(new LinkedNode<String>("blue", null));

colours = new LinkedNode<String>("green", colours);
```




```
System.out.format("1st: %s\n",  colours.get() );
System.out.format("2nd: %s\n", colours.next().get() );
System.out.format("3rd: %s\n",  colours.next().next().get() );
```

green
red
blue

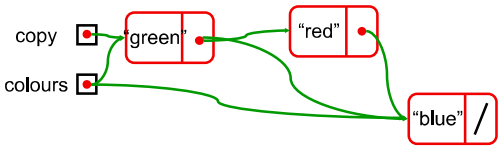
Aside: Memory allocation

- What are references/pointers?
  - Pointers/references are an address or a chunk of memory, where data can be stored.
- How do you get this memory allocated?
  - You've been doing it using **new**:
    - creating an object will allocate some heap memory for the object.
    - **new** returns the address of the chunk of memory
    - copying the address does not copy the chunk of memory.
- Memory from the heap must be recycled after use:
  - The **garbage collector** automatically frees up any memory chunks that no longer have anything pointing/referring to them.
  - This frees you from having to worry about explicitly freeing memory.

Using Linked Nodes

- Remove the second node:

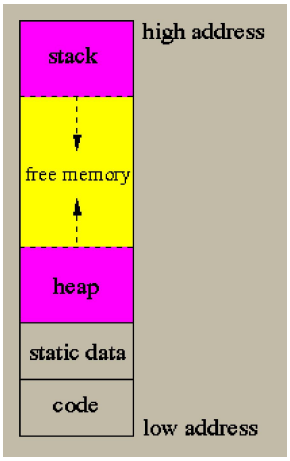
```
colours.setNext(colours.next().next());
```



- Copy colours, then remove first node

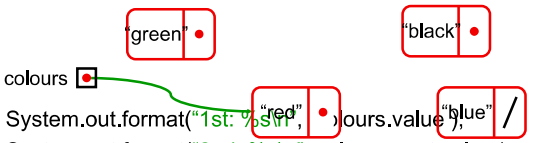
```
LinkedNode<String> copy = colours;
colours = colours.next();
```


Heap & memory allocation



Using Simpler Linked Nodes

```
LinkedNode<String> colours = new LinkedNode<String> ("red", null);
colours.next = new LinkedNode<String>("blue", null);
colours = new LinkedNode<String>("green", colours);
colours.next.next = new LinkedNode<String>("black", colors.next.next);
```




```
System.out.format("1st: %s\n",  colours.value );
System.out.format("2nd: %s\n", colours.next.value );
System.out.format("3rd: %s\n",  colours.next.next.value );
System.out.format("4th: %s\n",  colours.next.next.next.value );
colours.next.next.next.next = colours;
```

green
red
black
blue

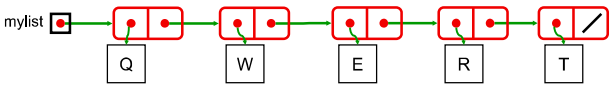
A circular list: g-r-b-b-g

A Linked Node class:

```
public class LinkedNode <E>{
    private E value;
    private LinkedNode<E> next;
    public LinkedNode(E item, LinkedNode<E> nextNode){
        value = item;
        next = nextNode;
    }
    public E get() { return value; }
    public LinkedNode<E> next() { return next; }
    public void set(E item) {
        value = item;
    }
    public void setNext(LinkedNode<E> nextNode) {
        next = nextNode;
    }
}
```

Inserting:

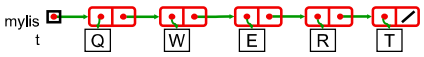

```
/** Insert the value at position n in the list (counting from 0)
   Assumes list is not empty,  n>0, and n <= length of list */
public void insert (E item, int n, LinkedNode<E>list){ ….
```



Insert X at position 2 in mylist
Insert Y at position 4 in mylist

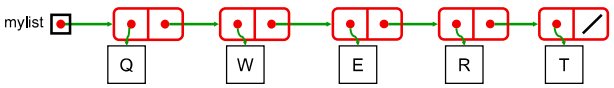
Inserting:


```
/** Insert the value at position n in the list (counting from 0)
   Assumes list is not empty,  n>0, and n <= length of list */
public void insert (E item, int n, LinkedNode<E>list){
   if (n == 1 )
      list.next = new LinkedNode<E>(item, list.next);
   else
      insert(item, n-1, list.next);
}
            or
public void insert (E item, int n, LinkedNode<E>list){
   int pos =0;
   LinkedNode<E> rest=list;   // rest is the pos'th node
   while (pos <n-1){
      pos++;
      rest=rest.next;
   }
   rest.next = new LinkedNode<E>(item, rest.next);
}
```



Removing:


```
/** Remove the value from the list
   Assumes list is not empty, and value not in first node */
public void remove (E item, LinkedNode<E>list){
```

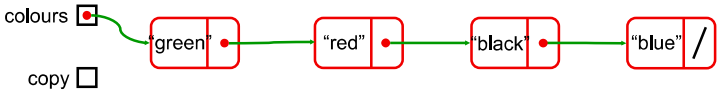


Remove R from mylist
Remove Y from mylist
Remove T from mylist

Using Simpler Linked Nodes

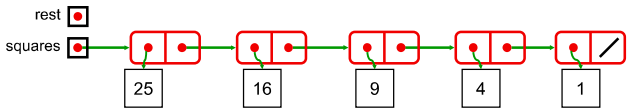
- Remove the third node:

```
colours.next.next = colours.next.next.next;
```



Creating & Iterating through a linked list


```
LinkedNode<Integer> squares = null;
for (int i = 1;  i < 6; i++)
   squares= new LinkedNode<Integer>( i*i, squares);

LinkedNode<Integer> rest = squares;
while (rest != null){
   System.out.format("%6d \n", rest.value);
   rest = rest.next;
}
  or
for (LinkedNode<Integer> rest=squares; rest!=null; rest=rest.next){
      System.out.format("%6d \n", rest.value);
}
```



Method to print a linked list


```
/** Prints the values in the list starting at a node */
public void printList(LinkedNode<E> list){
   if (list == null)   return;
   System.out.format("%d, ", list.value);
   printList(list.next);
}
```

Recursive methods are generally easier to design than iterative, for recursive data structures.

or

```
public void printList(LinkedNode<E> list){
   for (LinkedNode<Integer> rest=list;  rest!=null;   rest=rest.next )
      System.out.format("%d, ", rest.value);
}
```