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)

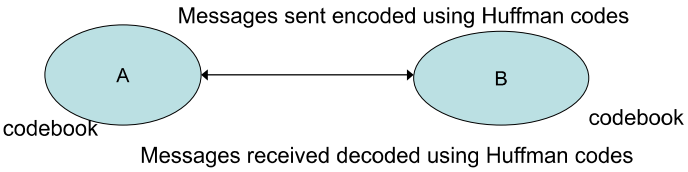
- In the context of software development, we can distinguish different kinds of abstractions:

- 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*

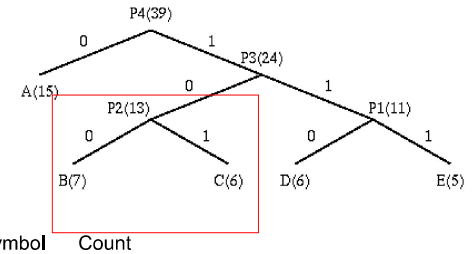
Huffman coding – Key ideas

- 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

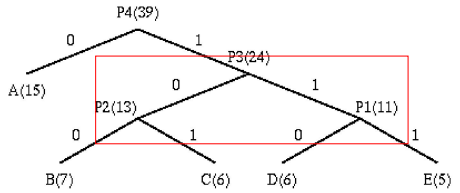
List: A(15),**B(7),C(6),**P1(11) → List: A(15),P1(11),*P2(13)*

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?

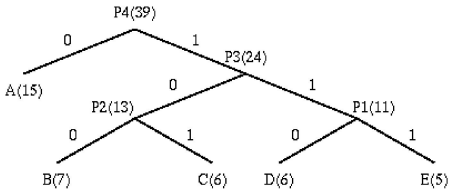
Constructing the en-/de-coding tree



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

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

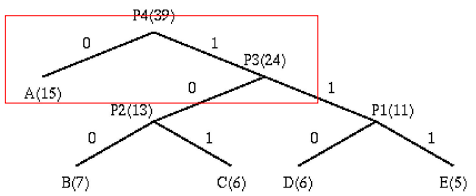
More on Huffman coding – en-/de-coding tree



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)

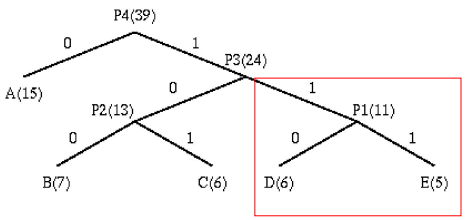
Constructing the en-/de-coding tree & table



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

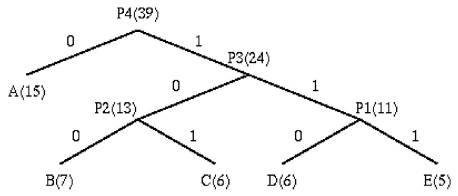
Constructing the en-/de-coding tree



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

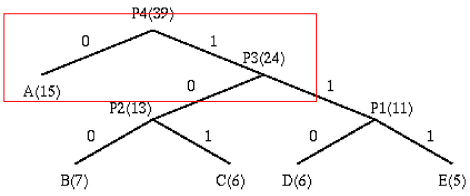
List: A(15),B(7),C(6),**D(6),E(5)** → List: A(15),B(7),C(6),*P1(11)*

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)

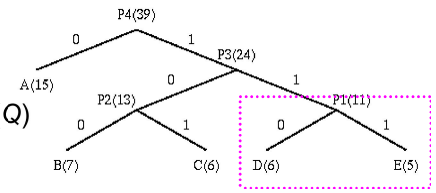
Binary Tree functions (methods) we need?

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

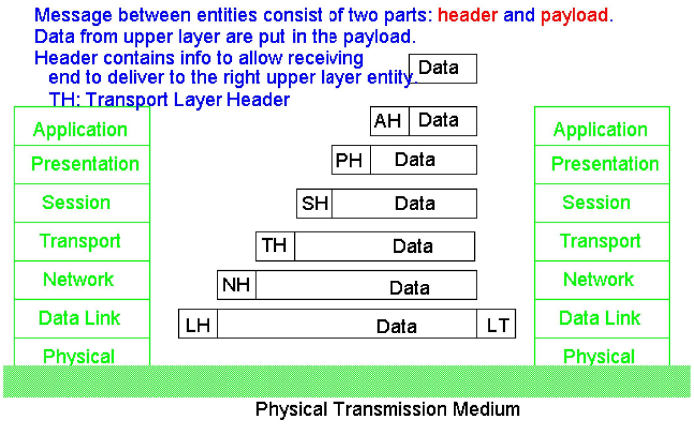
- In communication, encapsulation is the inclusion of one data structure within another structure so that the first data structure is hidden.
- For example, a TCP/IP-formatted packet can be encapsulated within an ATM frame.
- Within the context of sending and receiving the ATM frame, the encapsulated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

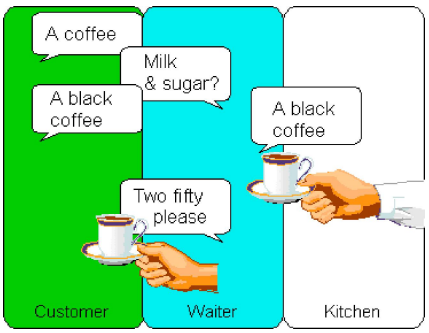
Advantages

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?



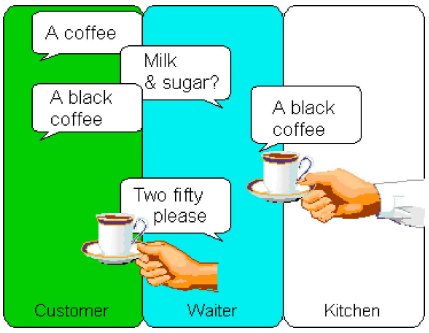
Data structures & abstraction, info hiding, encapsulation

- 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

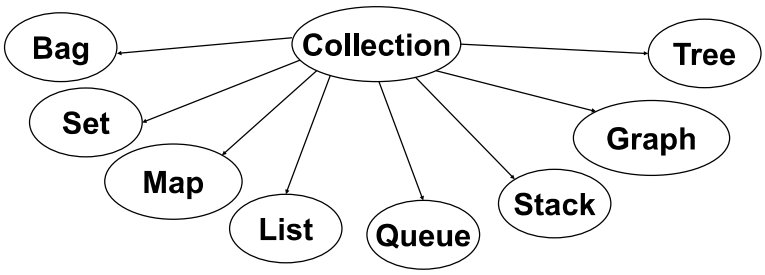
- A Java ***Interface*** corresponds to an Abstract Data Type
  - Specifies what methods can be called on objects of this type (specifies name, parameters and types, and type of return value)
  - Behaviour of methods is only given in comments (but cannot be enforced)

× 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

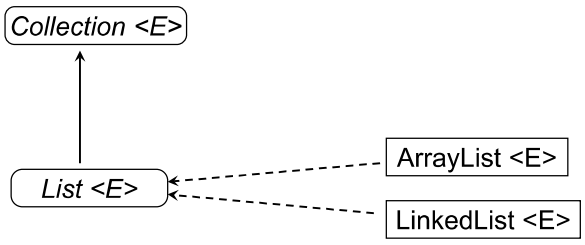
- We will drop "this." except when needed.
  - instead of `this.loadFromFile(fname)`
    just `loadFromFile(fname)`
- We will leave out { } when surrounding just one statement
  - instead of
```
while (i < name.length) {
  name[i] = null;
}
```
    just
```
while (i < name.length)
  name[i] = null;
```

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();
}
```

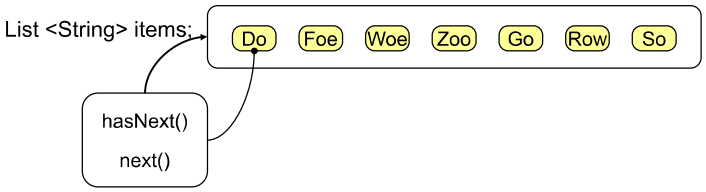
Example (TodoList program)

```
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

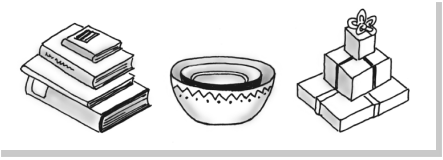


Fig. Some familiar stacks

HTML & XML examples

- **HTML example**

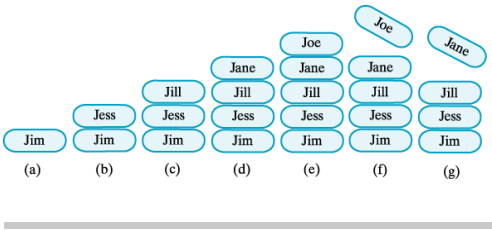
```
<html>
<body>
The content of the body element is displayed in your browser.
</body>
</html>
```

- **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

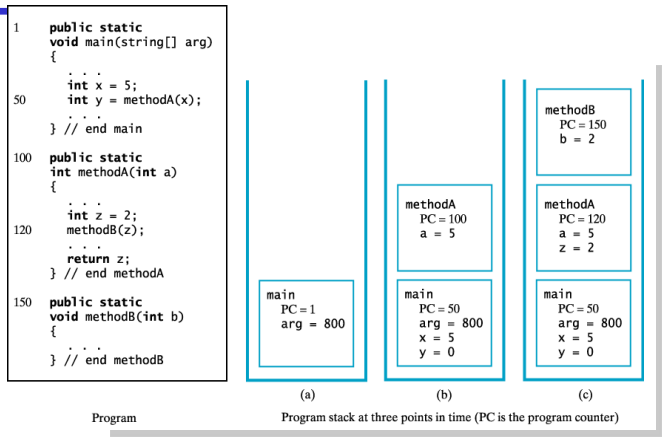


Fig. The program stack at 3 points in time; (a) when `main` begins execution; (b) when `methodA` begins execution, (c) when `methodB` begins execution.

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 (), [], {}

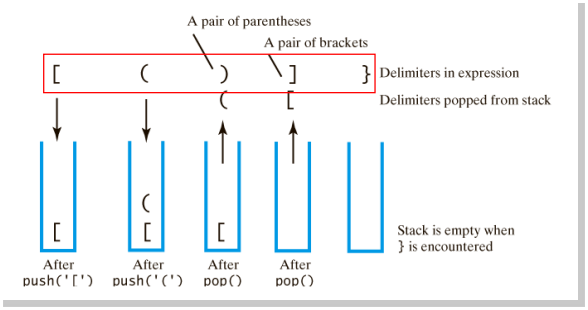


Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters `[()]}`

Checking for Balanced (), [], {}

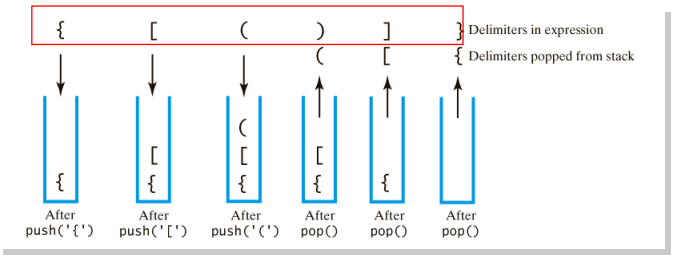


Fig. The contents of a stack during the scan of an expression that contains the balanced delimiters `{[()]}`

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

Show stack contents

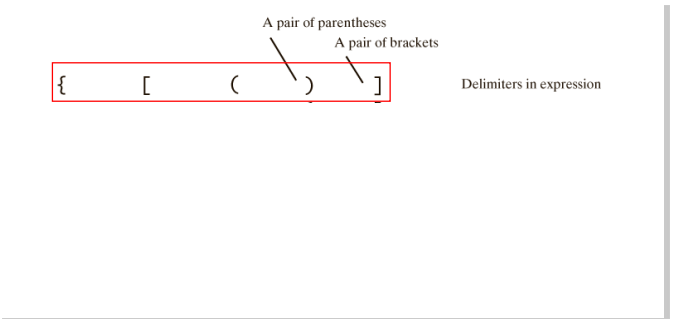


Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters `{[()]`

Checking for Balanced (), [], {}

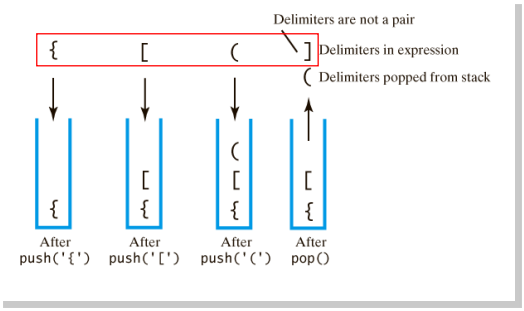


Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters `{[(])}`

Checking for Balanced (), [], {}

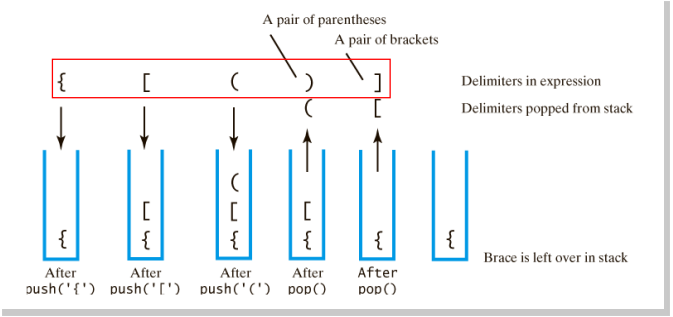


Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters `{[()]`

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

Show stack contents

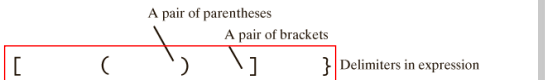


Fig. The contents of a stack during the scan of an expression that contains the unbalanced delimiters `[()]}`

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 /**

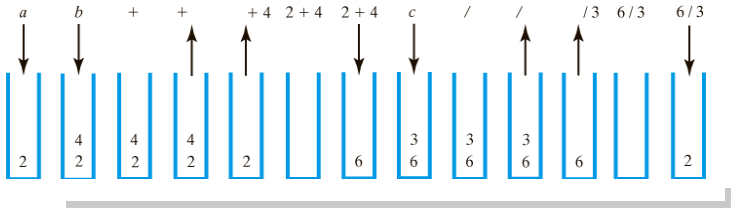


Fig. The stack during the evaluation of the postfix expression `a b + c /` when `a` is 2, `b` is 4 and `c` is 3

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/**

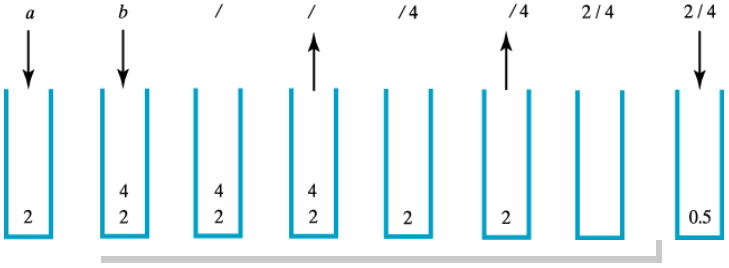


Fig. The stack during the evaluation of the postfix expression `a b /` when `a` is 2 and `b` is 4

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

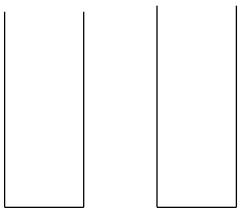


Fig. Two stacks during evaluation of **a + b * c** when **a = 2, b = 3, c = 4;** (a) after reaching end of expression; (b) while performing multiplication; (c) while performing the addition

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

Evaluating Infix Expressions using Two Stacks

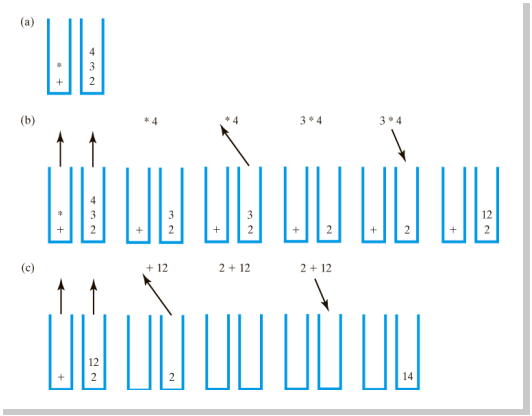


Fig. Two stacks during evaluation of **a + b * c** when **a = 2, b = 3, c = 4;** (a) after reaching end of expression; (b) while performing multiplication; (c) while performing the addition

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

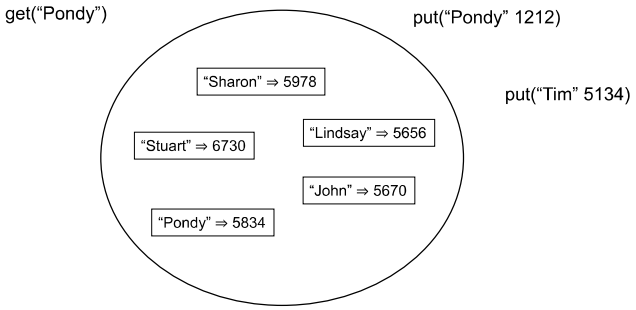
```
public void reverseNums(Scanner sc){
  Stack<Integer> myNums = new ArrayStack<Integer>();
  while (sc.hasNext())
     myNums.push(sc.Next())
  while (! myNums.isEmpty())
     textArea.append(myNums.pop() + "\n");
}
```

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

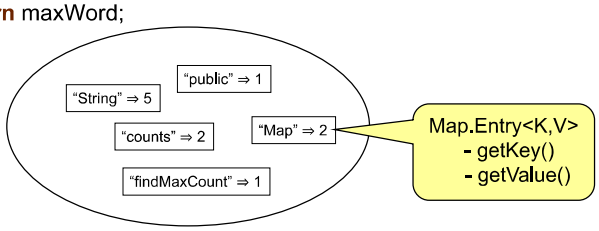
- How do you iterate through a Map? (eg, to print it out)
  - A Map isn't just a collection of items!
    ⇒ could iterate through the collection of keys
    ⇒ could iterate through the collection of values
    ⇒ could iterate through the collection of pairs
- Java Map allows all three!
  - keySet() → 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

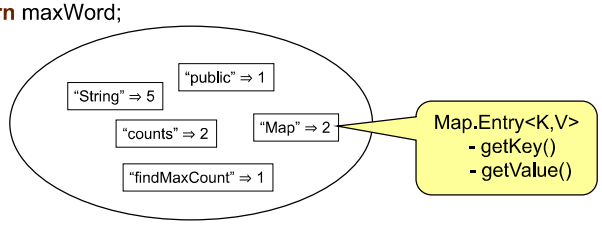
```
System.out.println( findMaxCount( countWords(file) ) );
```

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;
}
```

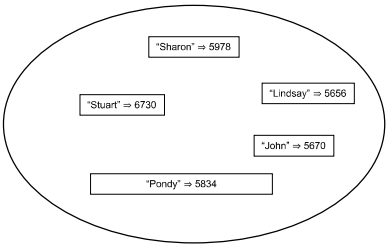


Iteration and “for each” loop

- Standard “for each” loop with collections:

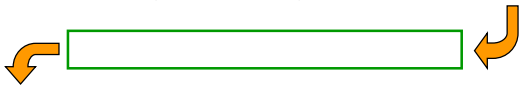
```
for (Face face : crowd) {
    face.render(canvas);
}
for (Map.Entry<String,Integer> entry : phonebook.entrySet()){
    textArea.append(entry.getKey()+" : "+entry.getValue());
}
```

- Uses Iterators.

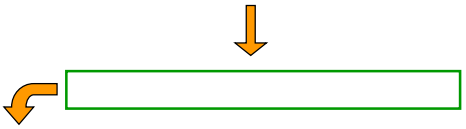


Queues

- 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

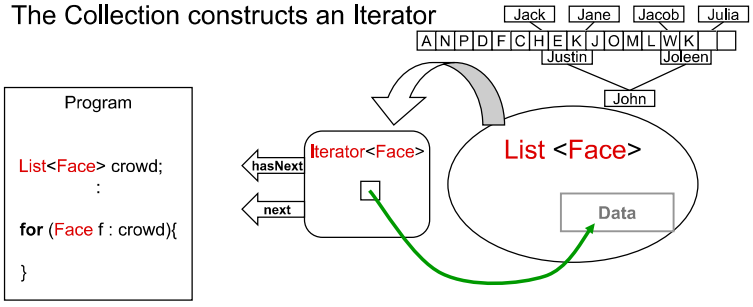


- **Priority queues**: add or remove with a given priority



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

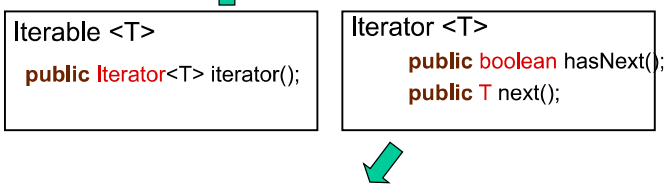
```
private class NumberSequenceIterator implements Iterator<Integer>{
    private int nextNum;
    private NumberSequence source;
    public NumberSequenceIterator(NumberSequence ns){
        source = ns;
        nextNum = ns.start;
    }
    public boolean hasNext(){
        return true;
    }
    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

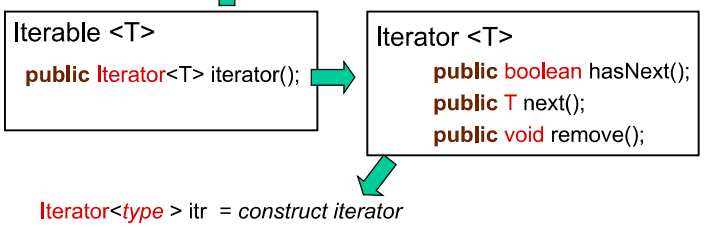
- Iterators and Iterables
- Sorting collections
- Comparators and Comparables

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

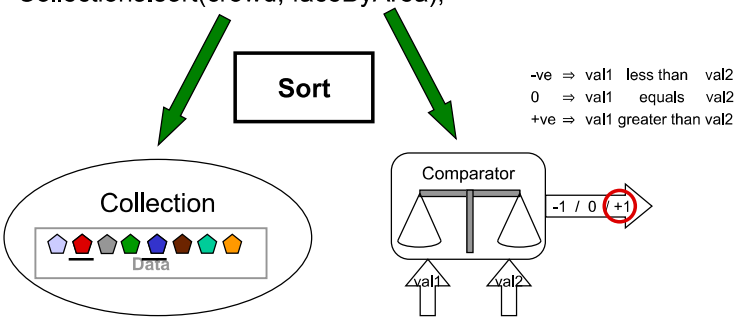
- Classes should implement the **Comparable interface** to control their ***natural ordering*.**
- Objects that implement Comparable can be sorted by **Collections.sort()** and **Arrays.sort()** and can be used as keys in a sorted map or elements in a sorted set without the need to specify a *Comparator*.

« 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

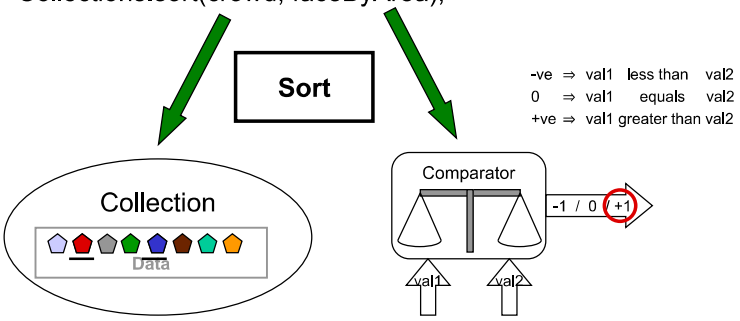
- An object defined under a comparable class will have a “natural ordering”. (T or F)
- Objects declared under a comparable class can be compared using which method?
- What is the signature of the *compareTo* method?
- Which method can be used to sort list of comparable objects?
- Comparator is an object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

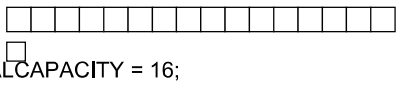
```
public class ArrayList <E> implements List<E> {
    private E [ ] data;
    private int count;
```

- 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);
}
   :
   :
```

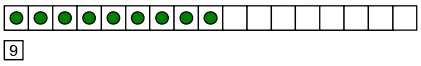
defined in terms of size and get

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;
    }
```



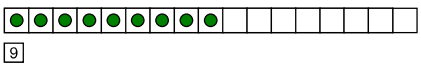
ArrayList extends AbstractList


```
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

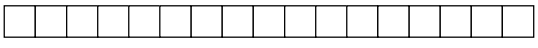
ArrayList: get


```
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 !

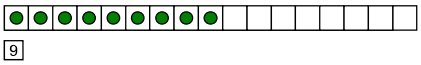
ArrayList: add


```
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???

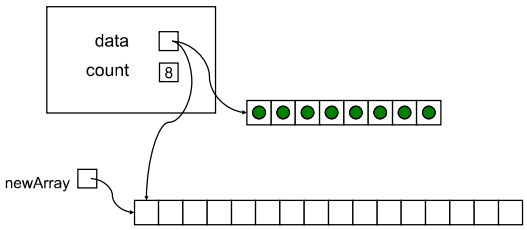
ArrayList: add (fixed)


```
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():



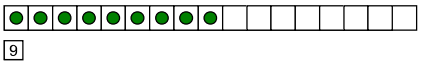
- How big should the new array be?

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
}
```

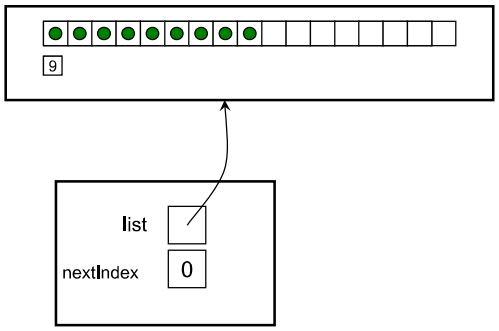

ArrayList: What else?

- 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

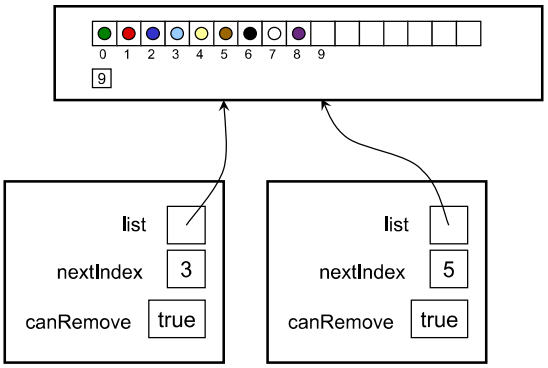
Iterator


```
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.

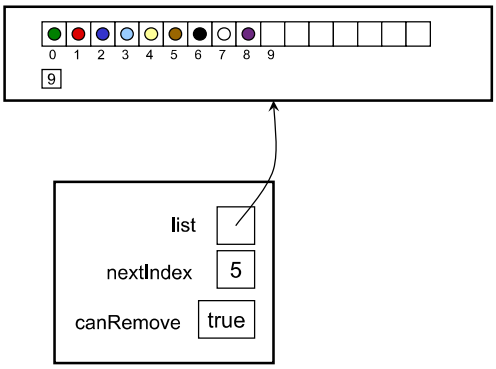
ArrayList: Cost

- 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}

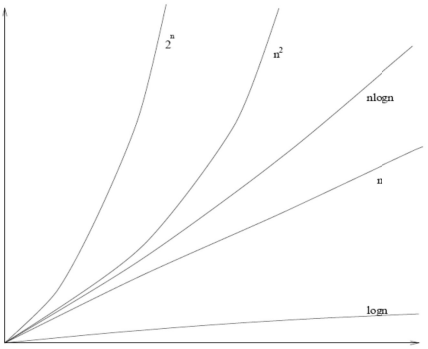
- O(1) - constant time, 10 ns

- 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"?

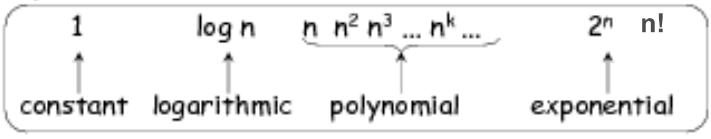
- Count
  - actions in the innermost loop (happen the most times)
  - actions that happen every time round (not inside an "if")
  - actions involving "data values" (rather than indexes)
  - representative actions (don't need every action)

```
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

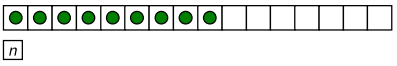
- We can define a hierarchy of functions each having a **greater** order of magnitude than its predecessor:



- 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

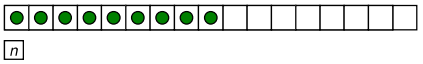
- Count the most expensive actions:
  - Adding 2 numbers is cheap
  - Raising to a power is not so cheap
  - Comparing 2 strings *may* be expensive
  - Reading a line from a file *may* be very expensive
  - Waiting for input from a user or another program may take forever…
- Sometimes we need to know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

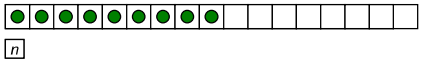
ArrayList costs: Summary

• 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:

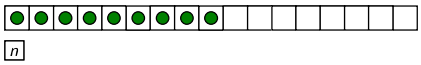
```
public void add(int index, E item){
  if (index<0 || index>count) throw new IndexOutOfBoundsException();
  ensureCapacity();
  for (int i=count; i > index; i--)
       data[i]=data[i-1];
  data[index]=item;
  count++;
}
```

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

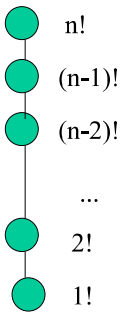
recursive functions

- 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);

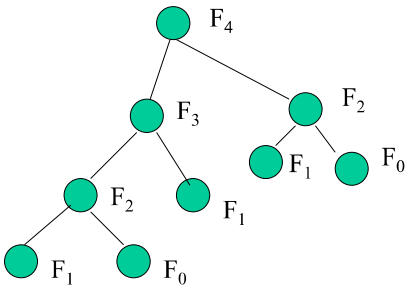
How does fibonacci work?

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

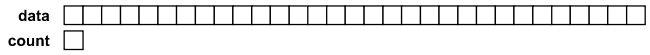
- Write a **test method**
  - As part of the class or as a separate testing class
  - Should test all the operations
  - Should test normal and extreme cases
- Good practice:
  - write it first (**black box testing**)
  - implement the collection
  - extend the test method to cover the special cases 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

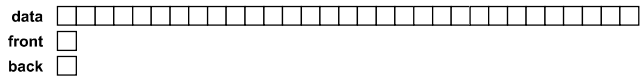
- Testing collection implementations
- Queues
- Motivation for linked lists
- Linked structures for implementing Collections

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?

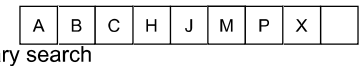
Where have we been?

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

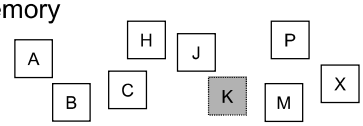


  ⇒ items must be sorted, to use binary search

- Arrays stored in contiguous chunks of memory.

  ⇒ inserting new items will be slow

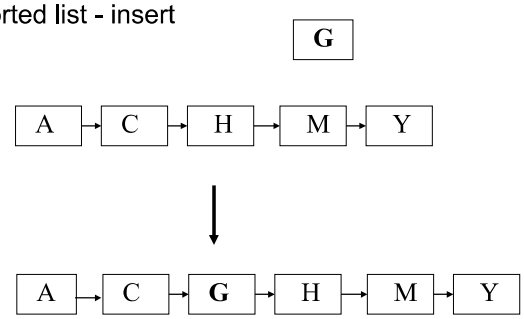
- Can't insert fast with an array!
- To insert fast, we need each item to be in its own chunk of memory



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

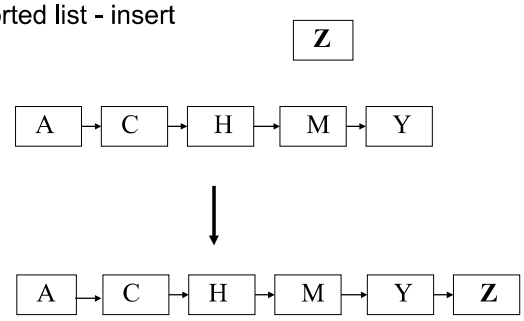
linked lists - insertion

- sorted list - insert

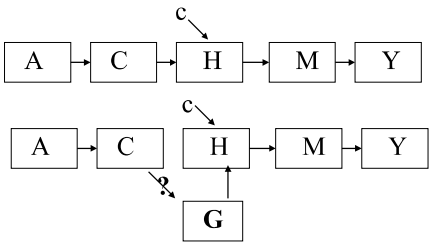


linked lists - insertion

- sorted list - insert



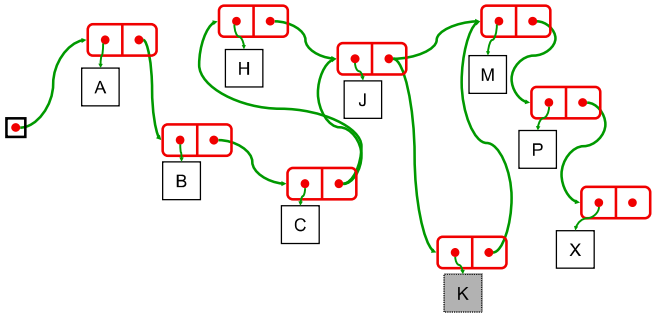
Insert in action



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

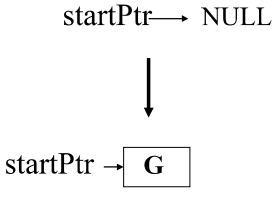
Linked lists

- 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.

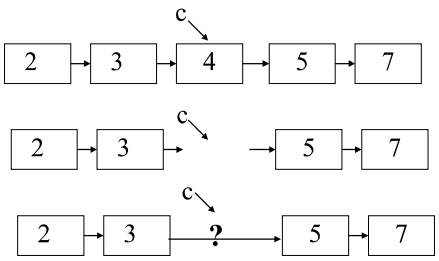
linked lists - insertion

- sorted list (empty) - insert



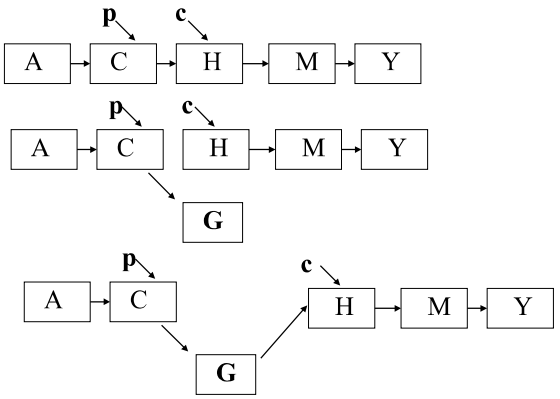
Delete in action

4



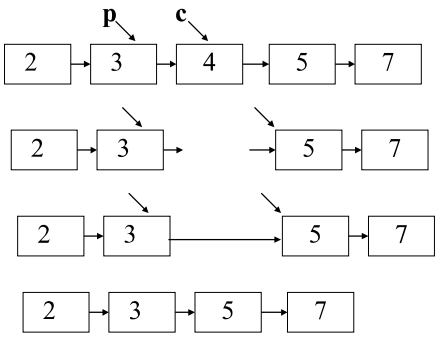
Problem: need to redirect nextPtr of 3 to 5, already lose track of 3 when c moved to 4
Solution ?

Insert in action



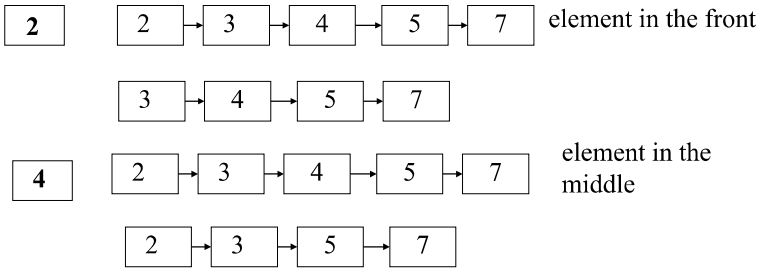
Delete in action

4



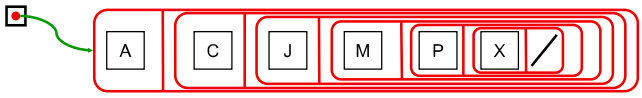
linked lists - search for deletion

- search & then delete

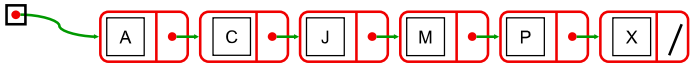


Linked List Structures – Alternative Views

- Can be drawn with Nodes inside Nodes:



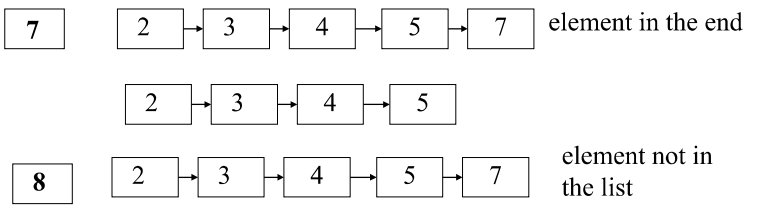
- "Pointers" are better:
  - Each node contains a reference to the next node
  - reference = memory location of / pointer to object



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

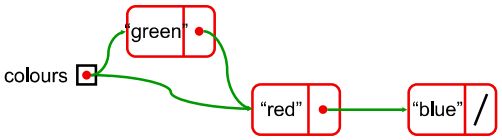
linked lists - search for deletion

- 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

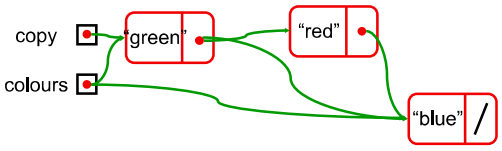
Aside: Memory allocation

- 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:

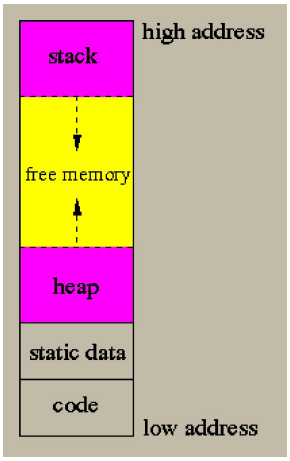
```
colours.setNext(colours.next().next());
```



- Copy colours, then remove first node

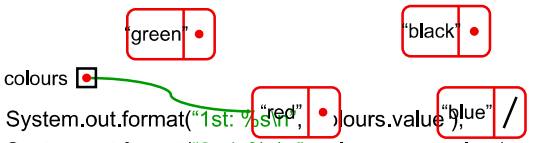
```
LinkedNode<String> copy = colours;
colours = colours.next();
```


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

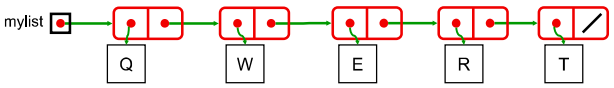
A circular list: g-r-b-b-g

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:

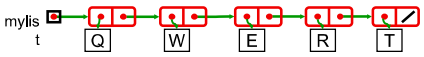

```
/** Insert the value at position n in the list (counting from 0)
   Assumes list is not empty,  n>0, and n <= length of list */
public void insert (E item, int n, LinkedNode<E>list){ ….
```



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

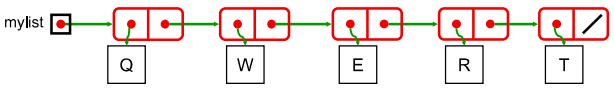
Inserting:


```
/** 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){
```

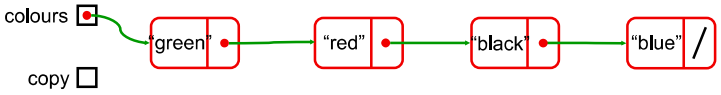


Remove R from mylist
Remove Y from mylist
Remove T from mylist

Using Simpler Linked Nodes

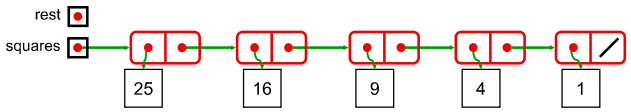
- Remove the third node:

```
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);
}
```



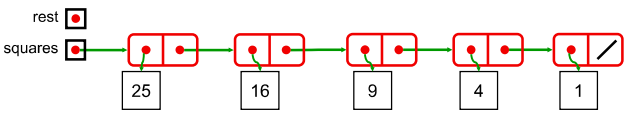
Method to print a linked list


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



Q&A

- When do you write test cases for “black box” testing? Before or after implementation?
- Explain why array implementations of queue are slow.
- Linked list allows data removal by?
- Define references/pointers.
- What is the purpose of garbage collection in memory management?

Summary

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

Readings

- [Mar07] Read 3.5
- [Mar13] Read 3.5

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){
    if (list.next==null) return;        // we are at the end of the list
    if (list.next.value.equals(item) )
        list.next =  list.next.next;
    else
        remove(item, list.next);
}
```

or

```
public void remove (E item, LinkedNode<E>list){
    LinkedNode<E> rest=list;
    while (rest.next != null  && !rest.next.value.equals(item))
        rest=rest.next;
    if (rest.next != null)
        rest.next = rest.next.next;
}
```

Why have a ‘rest’ to hold ‘list’?

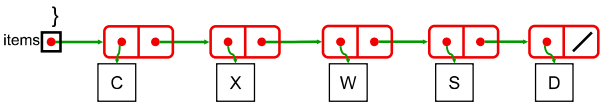
Exercise:

- Write a method to return the value in the LAST node of a list:

```
/** Returns the value in the last node of the list starting at a node */
public E lastValue (LinkedNode<E> list){  /* recursive version */



}
```

or

```
public E lastValue (LinkedNode<E> list){/* iterative version */

}
```



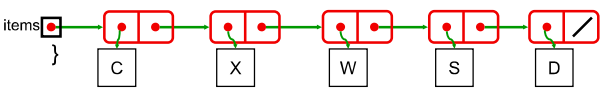
Exercise:

- Write a method to return the value in the LAST node of a list:

```
/** Returns the value in the last node of the list starting at a node */
public E lastValue (LinkedNode<E> list){



}
```

or

```
public E lastValue (LinkedNode<E> list){
```




```
}
```

List using linked nodes with header

- LinkedList extends AbstractList
- Has fields for linked list of Nodes and count
- Has an inner class: Node, with public fields
- get(index), set(index, item),
  - loop to index'th node, then get or set value
- add(index, item), remove(index):
  - deal with special case of index == 0
  - loop along list to node one before index'th node (Why?), then add or remove
  - check if go past end of list
- remove(item),
  - deal with special case of item in first node (i.e. conversion into empty set after removal)
  - loop along list to node one before node containing item (Why?), then remove
  - check if go past end of list

More Linked Structures

Lectures 13-14

A Linked List class:

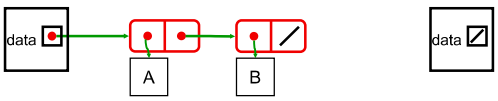

```
public class LinkedList <E> extends AbstractList <E> {
    private Node<E> data;
    private int count;
    public LinkedList(){
        data = null;
        count = 0;
    }
    /** Inner class:  Node */
    private class Node <E> {
        public E value;
        public Node<E> next;
        public Node(E val, Node<E> node){
            value = val;
            next = node;
        }
    }
```

Menu

- Linked structures for implementing Collections
- A collection class – Linked List
- Linked List methods

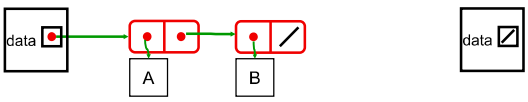
Linked List: get


```
public E get(int index){
    if (index < 0) throw new IndexOutOfBoundsException();
    Node<E> node=data;
    int i = 0;  // position of node
    while (node!=null && i++ < index)  node=node.next;
    if (node==null)   throw new IndexOutOfBoundsException();
    return node.value;
}
```



How do you make a good list class

- Must have an object that represent the empty list as an object
  - separate "header" object to represent a list



Linked Collections: Cost

- Linked structures allow fast insertion and deletion
  Does it help?

Linked List:

- Cost of get / set:
- Cost of insert:
- Cost of remove:

Linked Set (items in sorted order):

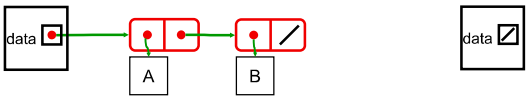
- Cost of contains:
- Cost of insert:
- Cost of remove

No advantage to Linked List?

Linked List: set


```
public E set(int index, E value){
    if (index < 0) throw new IndexOutOfBoundsException();
    Node<E> node=data;
    int i = 0;  // position of node
    while (node!=null && i++ < index) node=node.next;
    if (node==null)   throw new IndexOutOfBoundsException();
    E ans = node.value;
    node.value = value;
    return ans;
}
```

Same as get

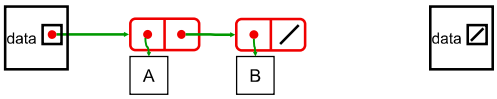


Menu

- Linked structures for implementing Collections
- A collection class – Linked List
- Linked List methods

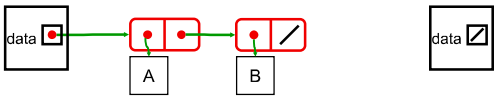
Linked List: add


```
public void add(int index, E item){
    if (item == null) throw new IllegalArgumentException();
    if (index==0){              // add at the front.
        data = new Node(item, data);
        count++;
        return;
    }
    Node<E> node=data;
    int i = 1;                  // position of next node
    while (node!=null && i++ < index) node=node.next;
    if (node == null) throw new IndexOutOfBoundsException();
    node.next = new Node(item, node.next);
    count++;
    return;
}
```



Linked List: remove


```
public boolean remove (Object item){
    if (item==null || data==null) return false;
    if (item.equals(data.value))   // remove the front item.
        data = data.next;
    else {          // find the node just before a node containing the item
        Node<E> node = data;
        while (node.next!=null && !node.next.value.equals(item))
            node=node.next;
        if (node.next==null)   return false;   // off the end
        node.next = node.next.next;   // splice the node out of the list
    }
    count--;
    return true;
}
```

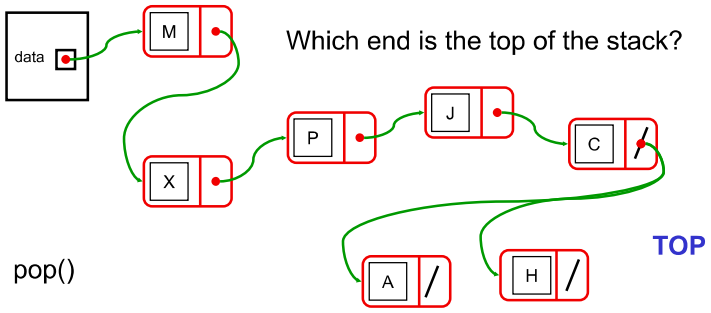


Linked Stacks and Queues

Lecture 15

A Linked Stack

- Implement a Stack using a linked list.



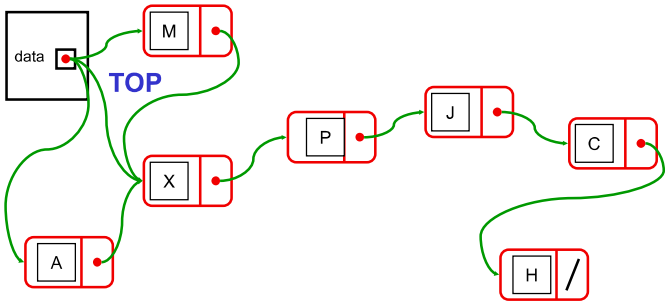
Which end is the top of the stack?

- pop()
- push("A")

Why is this a bad idea?

A Linked Stack

- Make the top of the Stack be the front of the list.



- pop()
- push("A")

Menu

- A Stack using a Linked List with a header
- A Queue using a Linked List with a header

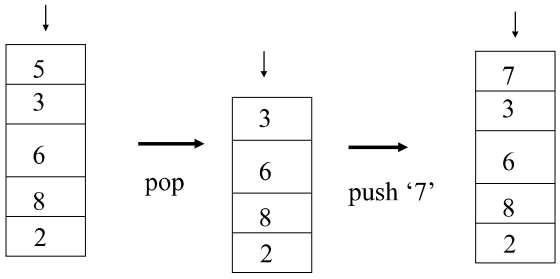
Implementing LinkedStack

- Use the LinkedNode class:

```
public class LinkedStack <E> extends AbstractCollection <E> {
    private Node<E> data = null;
    public LinkedStack(){ }
    public int size(){…
    public boolean isEmpty(){…
    public E peek(){…
    public E pop(){…
    public void push(E item){…
    public Iterator <E> iterator(){
```


Stacks (LIFO)

- insertions (push) & deletions (pop) only at the end (top)

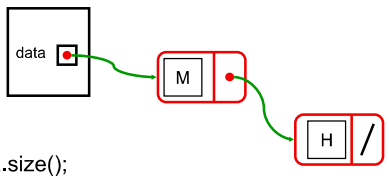


Application of Queues

- user job queue
- print spooling queue
- I/O event queue
- incoming packet queue
- outgoing packet queue

LinkedStack


```
public boolean isEmpty(){
    return data==null;
}
public int size () {
    if (data == null)   return 0;
    else                return data.size();
}
```

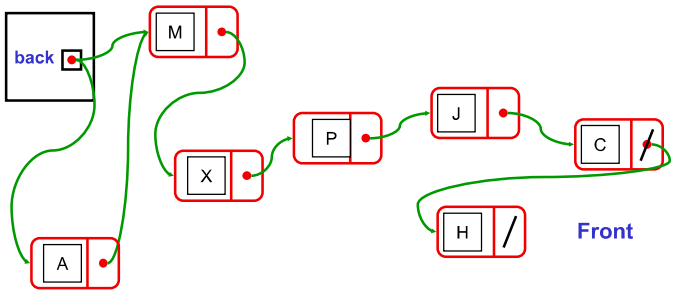


- Need size() method in Node class:

```
public int size (){
    int ans = 0;
    for (Node<E> rest = data; rest!=null; rest=rest.next)
        ans++;
    return ans;
}
```

A Linked Queue #1

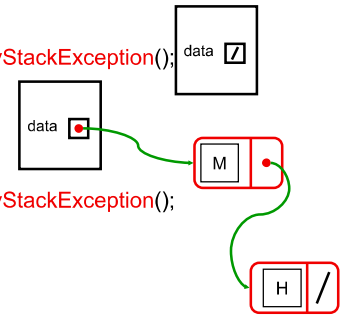
- Put the front of the queue at the end of the list



- offer("A")
- poll()

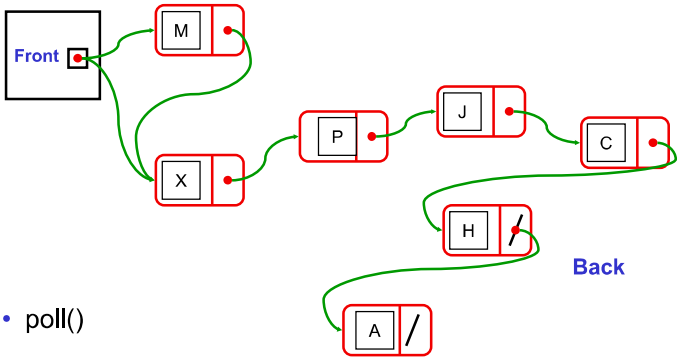
LinkedStack


```
public E peek(){
  if (data==null) throw new EmptyStackException();
  return data.value;
}

public E pop(){
  if (data==null) throw new EmptyStackException();
  E ans = data.value;
  data = data.next;
  return ans;
}
public void push(E item){
  if (item == null) throw new IllegalArgumentException();
  data = new Node(item, data);
}
public Iterator <E> iterator(){
    return new NodeIterator(data);
}
```



A Linked Queue #2

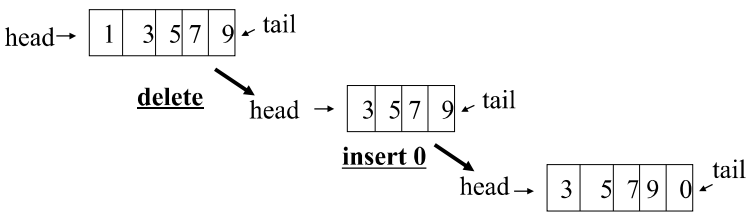
- Put the front of the Queue at the head of the list.



- poll()
- offer("A")

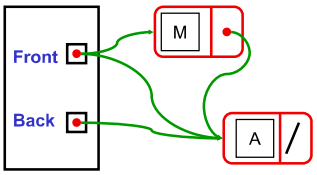
Queues (FIFO)

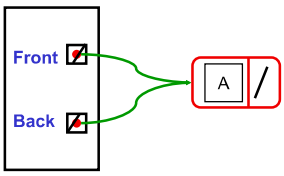
- Example: waiting lines
- Insertion at the end (tail), deletion from the front (head)



LinkedQueue

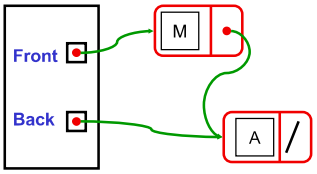
```
public E peek(){
   if (front==null)   return null;
   else               return front.value;
}

public E poll(){
   if (front==null) return null;
   E ans = front.value;
   front = front.next;
   if (front==null)  back = null;
   return ans;
}
```

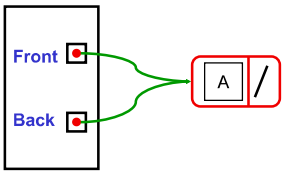




Exercise: work out the method body of 'offer'

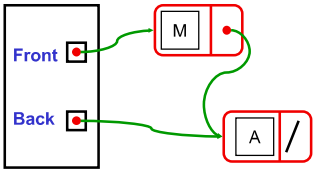
```
public boolean offer(E item){

}
```

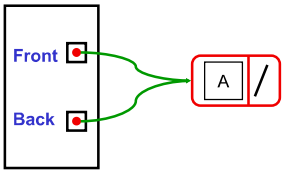




LinkedQueue

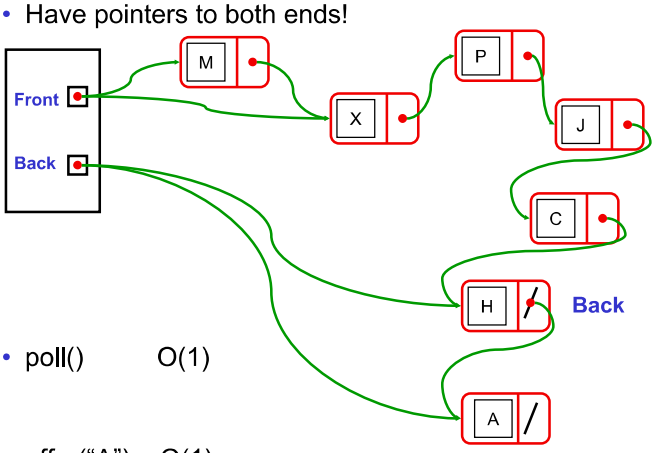
```
public boolean offer(E item){
   if (item == null) return false;
   if (front == null){
        back = new Node(item, null);
        front = back;
   }
   else {
        back.next = (new Node(item, null));
        back= back.next;
   }
   return true;
}
```





A Better Linked Queue

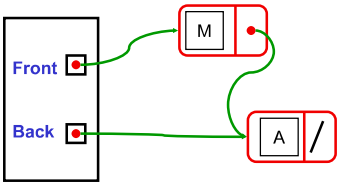
- Have pointers to both ends!



- poll() O(1)
- offer("A") O(1)

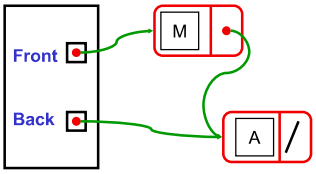
Implementing LinkedQueue

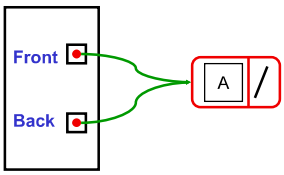
```
public class LinkedQueue <E> implements AbstractQueue <E> {

   private Node<E> front = null;
   private Node<E> back = null;

   public LinkedQueue(){ }

   public int size(){…

   public boolean isEmpty(){…

   public E peek(){…

   public E poll(){…

   public void offer(E item){…

   public Iterator <E> iterator(){
```

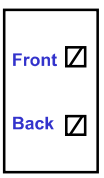


LinkedQueue

```
public boolean isEmpty(){
    return front==null;
}
public int size () {
   if (front == null)   return 0;
   else                 return front.size();
}
```







- Always three cases: 0 items, 1 item, >1 item

Linked Stack and Queue

- Uses a “header node”
  - contains link to head node, and maybe last node of linked list
- Important to choose the right end.
  - easy to add or remove from head of a linked list
  - hard to add or remove from the last node of a linked list
  - easy to add to last node of linked list if have pointer to tail
- Linked Stack and Queue:
  - all main operations are O(1)
- Can combine Stack and Queue
  - addFirst, addLast, removeFirst
  - also need removeLast to make a “Deque” (double-ended queue)
    ⇒ need doubly linked list (why?)
  - See the java “LinkedList” class.

Summary

- A Stack using a Linked List with a header
- A Queue using a Linked List with a header

Readings

- [Mar07] Read 3.6, 3.7, 6.2
- [Mar13] Read 3.6, 3.7, 6.2

ArraySet and Binary Search

Lecture 16

Menu

- Cost of ArraySet operations
- Binary Search
- Cost of SortedArraySet with Binary Search

ArrayList costs: Summary

• get	O(1)	
• set	O(1)	
• remove	O(n)	
• add (at i)	O(n)	(worst and average)
(have to shift up may have to double capacity)		
• add (at end)	O(1)	(most of the time)
(when doubles cap)	O(n)	(worst)
	O(1)	(amortised average) (if doubled each time)

ArraySet costs

What about ArraySet:

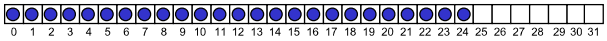
- Order is not significant

  ⇒ add() can choose to put a new item anywhere. where?

  ⇒ can reorder when removing an item. how?

- **Duplicates not allowed.**

  ⇒ must check if item already present before adding



ArraySet algorithms

Contains(value):

```
search through array,
    if value equals item
        return true
return false
```

Costs?

Add(value):

```
if  not contains(value),
    place value at end, (doubling array if necessary)
    increment size
```

Remove(value):

```
search through array
    if value equals item
        replace item by item at end. (why?)
        decrement size
        return
```

ArraySet costs

Costs:

- contains, add, remove: O(n)
- Question:
- How can we speed up the search?

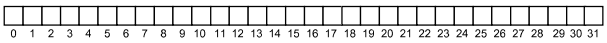
Binary Search

```
private boolean contains(Object item){
    Comparable<E> value = (Comparable<E>) item;
    int low = 0;                          // min possible index of item
    int high = count-1;                   // max possible index of item
                                          // item in [low .. high]  (if present)
    while (low <= high){
        int mid  =  (low + high) / 2;
        int comp =  value.compareTo(data[mid]);
        if (comp == 0)                    // item is present
            return true;
        if (comp < 0)                     // item in [low .. mid-1]
            high = mid - 1;               // item in [low .. high]
        else                              // item in [mid+1 .. high]
            low = mid + 1;                // item in [low .. high]
    }
    return false;    // item in [low .. high] and low > high,
                     // therefore item not present
}
```

low mid high

Binary Search: Cost

- What is the cost of searching if *n* items in set?
  - key step = ?



- Iteration Size of range Cost of iteration

  1 n

  2

  k 1

Time complexity

Let T(n) denote the time complexity of binary search algorithm on n numbers.

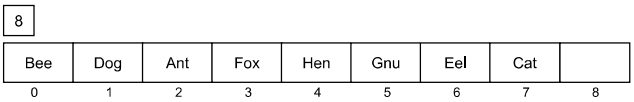
$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/2) + 1 & \text{otherwise} \end{cases}$$

We call this formula a **recurrence**.

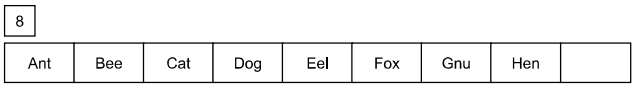
Making ArraySet faster.

All the cost is in the searching:

- Searching for “Eel”



- but if sorted…



Making ArraySet faster

- Binary Search: Finding “Eel”

- If the items are sorted (“ordered”), then we can search fast
  - Look in the middle:
    - if item is middle item ⇒ return
    - if item is before middle item ⇒ look in left half
    - if item is after middle item ⇒ look in right half

Divide and Conquer

One of the **best-known** algorithm design techniques.

Idea:

- A problem instance is ***divided*** into several **smaller** instances of the same problem, ideally of about same size
- The smaller instances are **solved**, typically **recursively**
- The solutions for the smaller instances are ***combined*** to get a solution to the original problem

Substitution method

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/2) + 1 & \text{otherwise} \end{cases}$$

Make a guess, $T(n) \le 2 \log n$

We prove statement by MI.

Assume true for all $n' < n$ [assume $T(n/2) \le 2 \log (n/2)$]

$T(n) = T(n/2) + 1$

$\le 2 \log (n/2) + 1$ ← *by hypothesis*

$= 2(\log n - 1) + 1$ ← ***log(n/2) = log n - log 2***

$< 2 \log n$

i.e., $T(n) \le 2 \log n$

More Example

Prove that $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$ is $O(n \log n)$

Guess: $T(n) \le 2 n \log n$

More Example

Prove that $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$ is $O(n \log n)$

Guess: $T(n) \le 2 n \log n$

Assume true for all $n' < n$ [assume $T(n/2) \le 2 (n/2) \log(n/2)$]

$T(n)$

$\le 2 (2 (n/2) \log (n/2)) + n$

$= 2 n (\log n - 1) + n$

$= 2 n \log n - 2n + n$

$\le 2 n \log n$

For the base case when n=2,
L.H.S = T(2) = 2T(1)+2 = 4,
R.H.S = 2 * 2 log 2 = 4
L.H.S ≤ R.H.S

i.e., $T(n) \le 2 n \log n$

Recurrence

A recurrence is an equation or inequality that describes a function in terms of ***its value on smaller inputs***.

E.g., $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 1 & \text{if } n > 1 \end{cases}$

To **solve** a recurrence is to derive ***asymptotic bounds*** on the solution

$Log_2(n)$ or $log(n)$

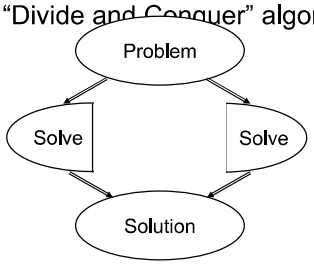
The number of times you can divide a set of n things in half.

log(1000) =10, log(1,000,000) = 20, log(1,000,000,000) =30

Every time you double n, you add one step to the cost! (why?)

- log(2n) = log(n)+log2 = log(n) +1
- Arises all over the place in analysing algorithms

Especially "Divide and Conquer" algorithms:



Substitution method

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/2) + 1 & \text{otherwise} \end{cases}$$

Make a guess, $T(n) \le 2 \log n$

We prove statement by MI.

Base case? When n=1, statement is FALSE!

L.H.S = **T(1) = 1** R.H.S = **c log 1 = 0** < L.H.S

Yet, when n=2,

L.H.S = T(2) = T(1)+1 = 2

R.H.S = 2 log 2 = 2

L.H.S ≤ R.H.S

Summary

- Cost of ArraySet operations
- Binary Search
- Cost of SortedArraySet with Binary Search

Readings

- [Mar07] Read 4.3
- [Mar13] Read 4.3

ArraySet with Binary Search

ArraySet: unordered

- All cost in the searching: O(n)
  - contains: O(*n*)
  - add: O(*n*)
  - remove: O(*n*)

SortedArraySet: with Binary Search

- Binary Search is fast: O(log(*n*))
  - contains: O(log(*n*))
  - add:
  - remove:

- All the cost is in keeping it sorted!!!!

Making SortedArraySet fast

- If you have to call add() and/or remove() many items, then SortedArraySet is no better than ArraySet!
  - Both O(*n*)
  - Either pay to search
  - Or pay to keep it in order

- If you only have to construct the set once, and then many calls to contains(),
  then SortedArraySet is much better than ArraySet.
  - SortedArraySet contains() is O(log(*n*))

- But, how do you construct the set fast?
  - Adding each item, one at a time

Alternative Constuctor

- Sort the items all at once

```
public SortedArraySet(Collection<E> col){
    // Make space
    count=col.size();
    data = (E[]) new Object[count];
    // Put items from collection into the data array.
    col.toArray(data);
    // sort the data array.
    Arrays.sort(data);
}
```

- How do you sort?

Sorting

Lecture 17

$Log_2(n)$ or $log(n)$:

The number of times you can divide a set of n things in half.

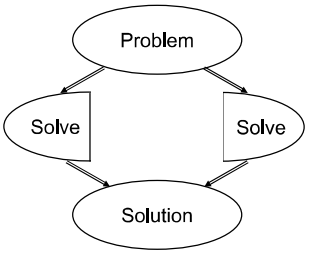
log(1000) =10, log(1,000,000) = 20, log(1,000,000,000) =30

Every time you double n, you add one step to the cost!

$$2^{\log(x)} = x$$

- Arises all over the place in analysing algorithms

  Especially "Divide and Conquer" algorithms:



Menu

- Binary Search
- Sorting
  - approaches
  - selection sort
  - insertion sort
  - bubble sort
  - analysis
  - fast sorts

SortedArraySet algorithms

Contains(value):
index = findIndex(value),
return (data[index] equals value)

Add(value):
index = findIndex(value),
if index == count or data[index] not equal to value,
double array if necessary
move items up, from position count-1 down to index
insert value at index
increment count

Remove(value):
index = findIndex(value),
if index < count and data[index] equal to value
move items down, from position index+1 to count-1
decrement count

Assume data is sorted

Binary Search: Cost

- What is the cost of searching if n items in set?
  - key step = ?

Iteration	Size of range	Cost of iteration
1	n	
2		
k	1	

Sets with Binary Search

ArraySet: unordered

- contains: O(n)
- add: O(n)
- remove: O(n)

⇒ All the cost is in the searching: O(n)

SortedArraySet: with Binary Search

- Binary Search is fast: O(log(n))
  - contains: O(log(n))
  - add:
  - remove:

⇒ All the cost is in keeping it sorted!!!!

Sort them!

Rat	Pig	Owl	Kea	Fox	Hen	Yak	Ant_1	Tui	Dog	Cat	Man	Jay	Bee	Eel	Gnu	Ant_2	Sow

73	3	6931	427	5	45	463	941	7273	64	9731	61	873	44	74	465	6929	75

How to do it?

Ways of sorting

- Selecting sorts:
  - Find the next largest/smallest item and put in place
  - Builds the correct list in order
- Inserting Sorts:
  - For each item, insert it into an ordered sublist
  - Builds a sorted list, but keeps changing it
- Compare and Swap Sorts:
  - Find two items that are out of order, and swap them
  - Keeps "improving" the list
- Radix Sorts
  - Look at the item and work out where it should go.
  - Only works on some kinds of values.
- …

Analysing Sorting Algorithms

- Efficiency
  - What is the (worst-case) order of the algorithm?
  - Is the average case much faster than worst-case?
- Requirements on Data
  - Does the algorithm need random-access data? (*vs* streaming data)
  - Does it need anything more than "compare" and "swap"?
- Space Usage
  - Can the algorithm sort in-place? or does it need extra space?

Making SortedArraySet fast

- If you have to call add() and/or remove() many items, then SortedArraySet is no better than ArraySet!
  - Both O(*n*)
  - Either pay to search
  - Or pay to keep it in order
- If you construct the set once but many calls to contains(), then SortedArraySet is much better than ArraySet.
  - SortedArraySet contains() is O(log(*n*))
- But, how do you construct the set fast?

Why Sort?

- Constructing a sorted array one item at a time is very slow :
  - add(item) is n/2 on average [O(n)]
    ⇒ 1/2 + 2/2 + 3/2 + …. n/2 is $n^2/4$ [$O(n^2)$]
  - ≈ 2,500,000,000,000,000 steps for 100,000,000 items
    ⇒ 25,000,000 seconds = 289 days at 10ns per step.
- There are sorting algorithms that are much faster if you can sort whole array at once: O(n log(n))
  - ≈ 2,700,000,000 steps for 100,000,000 items
    ⇒ 27 seconds at 10ns per step.

Alternative Constuctor

- Sort the items all at once

```
public SortedArraySet(Collection<E> colln){

    // Make space
    count=colln.size();
    data = (E[]) new Object[count];

    // Put items from collection into the data array.
    colln.toArray(data);

    // sort the data array.
    Arrays.sort(data);
}
```

- How do you sort?

Example

- sort (34, 8, 64, 51, 32, 21) in ascending order

Sorted part	Unsorted part	int moved
	34 8 64 51 32 21	
34	**8** 64 51 32 21	-
8 34	**64** 51 32 21	34
8 34 64	**51** 32 21	-
8 34 51 64	**32** 21	64
8 32 34 51 64	**21**	34, 51, 64
8 21 32 34 51 64		32, 34, 51, 64

In-place sorting

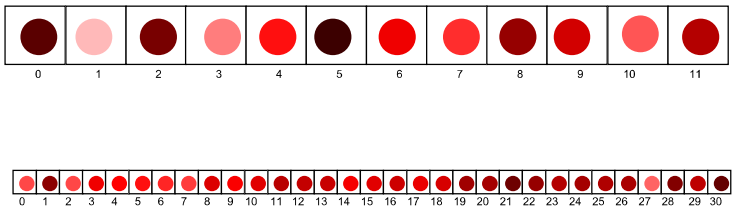
Selection Sort – Example

- sort (34, 10, 64, 51, 32, 21) in ascending order

Sorted part	Unsorted part	Swapped
	34 **10** 64 51 32 21	10, 34
10	34 64 51 32 **21**	21, 34
10 21	64 51 **32** 34	32, 64
10 21 32	51 64 **34**	51, 34
10 21 32 34	64 **51**	51, 64
10 21 32 34 51	**64**	--
10 21 32 34 51 64		

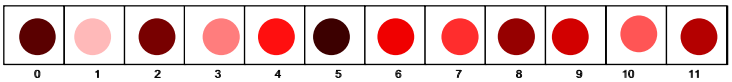
In-place sorting

Compare and Swap Sorts



- Bubble Sort (terrible)
- QuickSort (the fastest) (Divide and Conquer)

Inserting Sorts



- Insertion Sort (slow)
- Merge Sort (fast) (Divide and Conquer)

Bubble Sort

starting from the last element, swap adjacent items if they are not in ascending order

when first item is reached, the first item is the smallest

repeat the above steps for the remaining items to find the second smallest item, and so on

In-place sorting

Insertion Sort

look at elements one by one

build up sorted list by inserting the element at the correct location

Implementing Sorting Algorithms

- Could sort Lists
  - ⇒ general and flexible
  - but efficiency depends on how the List is implemented
- Could sort Arrays.
  - ⇒ less general
  - but efficiency is well defined
  - easy to convert any Collection to an array: **toArray()** method.
- Comparing items:
  - require items to be comparable (natural order)
  - provide comparator (prescribed order)
  - handle both.

Sort methods

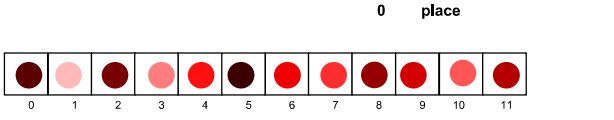
We will use:

```
public void …Sort(E[] data, int size, Comparator<E> comp)
```

- sorts first *size* elements of an array of some type, given a Comparator for that type.
- Could be used inside SortedArraySet, or standalone.
- Designing very flexible code that can be used in many different places is tricky!!

Selection Sort


```
public void selectionSort(E[ ] data, int size, Comparator<E> comp){
    // for each position, from 0 up, find the next smallest item
    // and swap it into place
    for (int place=0; place<size-1; place++){
        int minIndex = place;
        for (int sweep=place+1; sweep<size; sweep++){
            if (comp.compare(data[sweep], data[minIndex]) < 0)
                minIndex=sweep;
            }
        swap(data, place, minIndex);
    }
}
```



Bubble Sort – Example

round	(34	10	64	51	32	21)
	34	10	64	51	**32**	**21**
1	34	10	64	**51**	**21**	32
	34	10	**64**	**21**	51	32
	34	**10**	**21**	64	51	32
	34	**10**	21	64	51	32
	10	34	21	64	**51**	**32**
2	***10***	34	21	**64**	**32**	51
	10	34	**21**	**32**	64	51
	10	**34**	**21**	32	64	51
	10	***21***	34	32	64	51

Bubble Sort – Example (2)

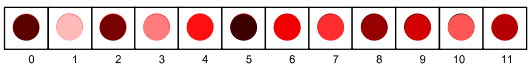
round	***10***	***21***	34	32	**64**	**51**
3	***10***	***21***	34	**32**	**51**	64
	10	***21***	**34**	**32**	51	64
	10	***21***	***32***	34	**51**	**64**
4	***10***	***21***	***32***	**34**	**51**	64
	10	***21***	***32***	***34***	**51**	**64**
5	***10***	***21***	***32***	***34***	***51***	64

bubble sort

- also called as sinking sort
- smaller values bubble their way up during the process
- need several passes through the data (how many?)
- During each pass, successive pairs of elements are compared
  - if a pair is in order, leave them as they are
  - if a pair not in order, swap

Bubble Sort

```
public  void bubbleSort(E[] data, int size, Comparator<E> comp){
    // Repeatedly scan array, swapping adjacent items if out of order
    // Builds a sorted region from the end
    for (int top=size-1; top>0; top--){
       for (int sweep=0; sweep<top; sweep++)
          if (comp.compare(data[sweep], data[sweep+1]) >0)
              swap(data, sweep, sweep+1);
    }
}
```



Bubble Sort Analysis

- Cost:
  - step?
  - best case:
    - what is it?
    - cost:
  - worst case:
    - what is it?
    - cost:
  - average case:
    - what is it?
    - cost:

Bubble Sort Analysis

- Efficiency
  - worst-case: **O(n^2)**
  - average case: **O(n^2),** no better than worst
- Requirements on Data
  - Needs random-access data, but can modify for files
  - Needs compare and swap
- Space Usage
  - in-place

Selection Sort Analysis

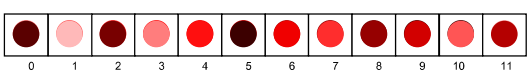
- Cost:
  - step?
  - best case:
    - what is it?
    - cost:
  - worst case:
    - what is it?
    - cost:
  - average case:
    - what is it?
    - cost:

Selection Sort Analysis

- Efficiency
  - worst-case: **O(n^2)**
  - average case exactly the same.
- Requirements on Data
  - Needs random-access data, but easy to modify for files
  - Needs compare and swap
- Space Usage
  - in-place

Exercise: Bubble Sort Implementation

```
public  void bubbleSort(E[] data, int size, Comparator<E> comp){
    // Repeatedly scan array, swapping adjacent items if out of order
    // Builds a sorted region from the end




}
```



Slow Sorts

- Insertion sort, Selection Sort, Bubble Sort:
  - All slow (except Insertion sort on almost sorted lists)
  - Insertion sort is better than the others
- Problem:
  - Insertion and Bubble
    - only compare adjacent items
    - only move items one step at a time
  - Selection
    - compares every pair of items –
      - ignores results of previous comparisons.
- Solution:
  - **Must be able to compare and swap items at a distance**
  - **Must not perform redundant comparisons**

Summary

- Binary Search
- Sorting
  - approaches
  - selection sort
  - insertion sort
  - bubble sort
  - analysis
  - fast sorts

Readings

- [Mar07] Read 7.1, 7.2, 7.3
- [Mar13] Read 7.1, 7.2, 7.3

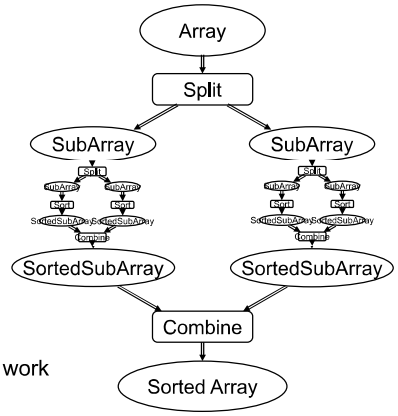
Fast Sorting

Lecture 18

Divide and Conquer Sorts

To Sort:

- Split
- Sort each part (recursive)
- Combine

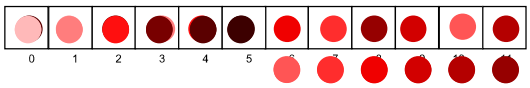


Where does the work happen?

- MergeSort:
  - split trivial
  - combine does all the work
- QuickSort:
  - split does all the work
  - combine trivial

Merge Sort

- Split the array exactly in half
- Sort each half
- "Merge" them together.



Need a temporary array

Menu

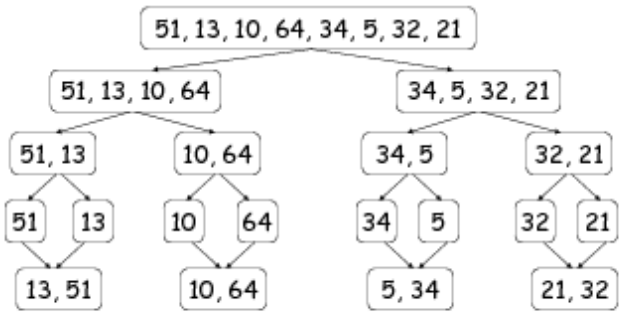
- Sorting
  - Design by Divide and Conquer
  - Merge Sort
  - QuickSort

51, 13, 10, 64, 34, 5, 32, 21

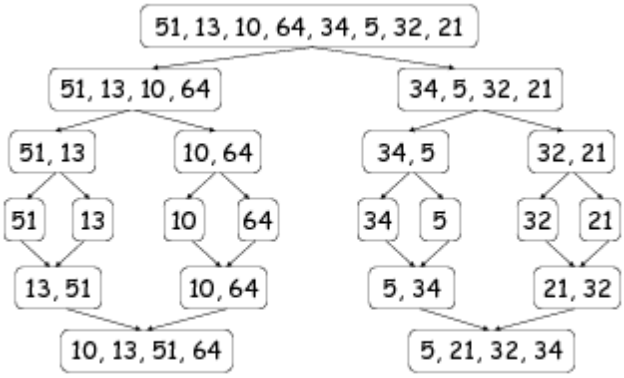
we want to sort these 8 numbers,
divide them into two halves

Slow Sorts

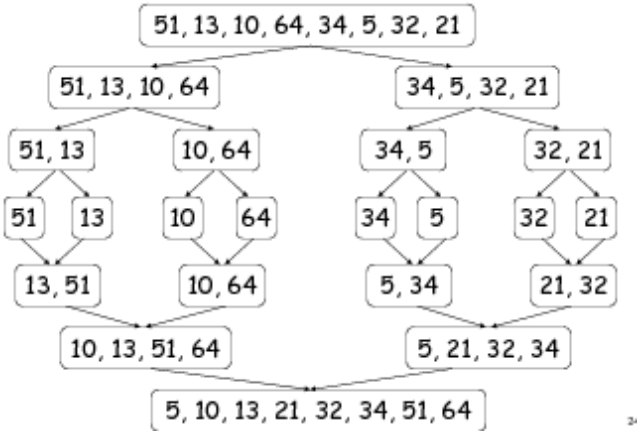
- Insertion sort, Selection Sort, Bubble Sort:
  - All slow (except Insertion sort on almost sorted lists)
  - $O(n^2)$
- Problem:
  - Insertion and Bubble
    - only compare adjacent items
    - only move items one step at a time
  - Selection
    - compares every pair of items –
      - ignores results of previous comparisons.
- Solution:
  - compare and swap items at a distance
  - do not perform redundant comparisons

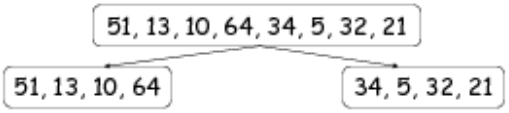


then **merge** again into sequences of 4 sorted numbers



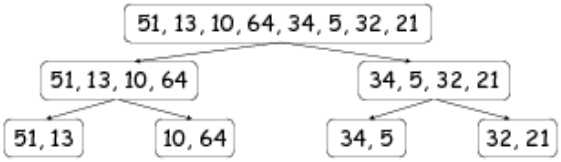
one more merge give the **final** sorted sequence



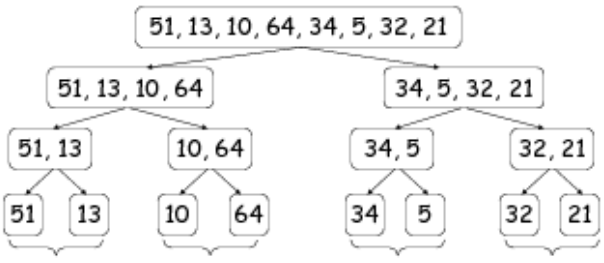


divide these 4 numbers into halves

similarly for these 4



further divide each shorter sequence ... until we get sequence with only **1** number



merge pairs of single number into a sequence of 2 sorted numbers

MergeSort


```
private static <E> void mergeSort(E[] data, E[] temp, int low, int high,
                                  Comparator<E> comp){
    // sort items from low..high-1 using temp array
    if (high > low+1){
        int mid = (low+high)/2;
        // mid = low of upper 1/2, = high of lower half.
        mergeSort(data, temp, low, mid, comp);
        mergeSort(data, temp, mid, high, comp);
        merge(data, temp, low, mid, high, comp);
        for (int i=low; i<high; i++)   data[i]=temp[i];
    }
}
```

Sort each half

merge into temp

copy back

Mergesort – merging details

- given two sorted arrays, merge them into one sorted array
- keep track of the smallest element in each array, output the smaller of the two to a third array
- Continue until both arrays are exhausted
- If any array is exhausted first, then simply output the rest of another array

- This so-called 2-way merging can be generalized to multi-way merging

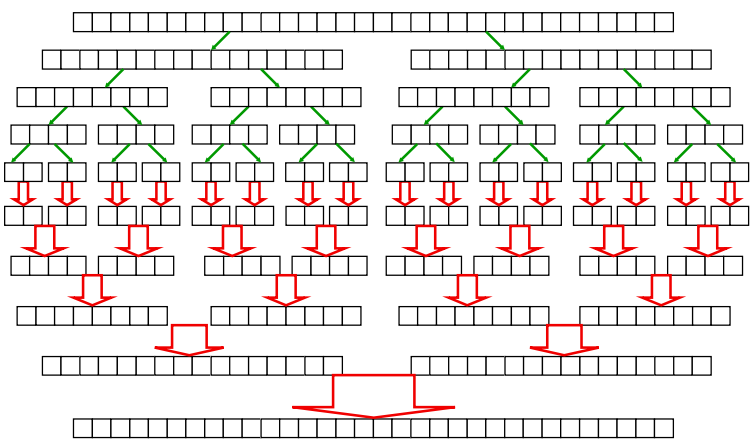
Merge


```
/** Merge from[low..mid-1] with from[mid..high-1] into  to[low..high-1.*/
private static <E> void merge(E[] from, E[] to, int low, int mid, int high,
                              Comparator<E> comp){
    int index = low;        // where we will put the item into "to"
    int indxLeft = low;     // index into the lower half of the "from" range
    int indxRight = mid;    // index into the upper half of the "from" range
    while (indxLeft<mid && indxRight < high){
        if (comp.compare(from[indxLeft], from[indxRight]) <=0)
            to[index++] = from[indxLeft++];
        else
            to[index++] = from[indxRight++];
    }
    //copy over the remainder. Note only one loop will do anything.
    while (indxLeft<mid)
        to[index++] = from[indxLeft++];
    while (indxRight<high)
        to[index++] = from[indxRight++];
}
```

Merging process in details

3 4 7 33 78 (2) 11 54 69 71 82 99 ← **2**	find the smallest of each array; compare
(3) 4 7 33 78 **11** 54 69 71 82 99 ← **2 3**	output the smaller **2**; remove 2 from the input array; find the smallest of each array; compare
(4) 7 33 78 **11** 54 69 71 82 99 ← **2 3 4**	output the smaller **3**;remove 3 from the input array; find the smallest of each array; compare
(7) 33 78 **11** 54 69 71 82 99 ← **2 3 4 7**	output the smaller **4**;remove 4 from the input array; find the smallest of each array; compare
33 78 (11) 54 69 71 82 99 ← **2 3 4 7** 11	output the smaller **7**;remove 7 from the input array; find the smallest of each array; compare

MergeSort



MergeSort

- Needs a temporary array for copying
  - create a temporary array
  - [fill with a copy of the original data.]

```
public static <E> void mergeSort(E[] data, int size,
                                 Comparator<E> comp){
    E[] other = (E[])new Object[size];
    for (int i=0; i<size; i++) other[i]=data[i];
    mergeSort(data, other, 0, size, comp);
}
```

Quicksort in process

7 4 3 9 0 8 6	find an 'i';use v = '6' to compare
l **r** **v**	use two pointers **l** & **r**, **l** scan from the left, stop when a[l] > v; r scan from right, stop when a[r]<v
0 4 3 9 **7** 8 6	swap a[l] & a[r]
0 4 3 **9** **7** 8 6	scan again from where we stop
l=r	stop again (when l >= r); i is found
0 4 3 **6** 7 8 **9**	swap a[l] with v; now every element to the left of 6 is less than 6, every element to the right of 6 is greater than 6
0 4 3 6 **7 8** 9	apply the same process to each partition
l r v **l r v**	
0 **4** 3 6 **7** 8 9	right partition sorted.
l=r	left partition stops scanning, new i found
0 **3** **4**	swap a[l] with v (3); left partition sorted
0 3 4 6 7 8 9	Done.

Exercise

- Rewrite **mergeSort** to improve its performance

QuickSort – in brief

- Divide and Conquer,
  but does its work in the "**split**" step
- It splits the array into two (possibly unequal) parts:
  - choose a "**pivot**" item
  - make sure
    - **all** items < pivot are in the **left** part
    - **all** items > pivot are in the right part
- Then (**recursively**) sorts each part

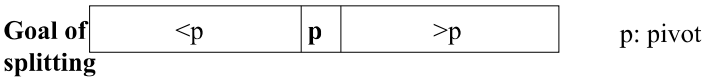
```
public static <E> void quickSort(E[] data, int size, Comparator<E> comp){
    quickSort(data, 0, size, comp);
}
```

Quicksort

- Invented by C.A.R. Hoare
- Used more widely than others

- works well for different types of data
- divide & conquer on sorting

- **O(N log N) on average**
- $O(N^2)$ worst case



Quicksort in Python

- The following quickSort code in Python from Wikipedia.

```
def quickSort(arr):
    less = []
    pivotList = []
    more = []
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        for i in arr:
            if i < pivot:
                less.append(i)
            elif i > pivot:
                more.append(i)
            else:
                pivotList.append(i)
        less = quickSort(less)
        more = quickSort(more)
        return less + pivotList + more
```

Quicksort ideas

- partition the array into two parts
- partitioning involves the selection of a[i] where the following conditions are met:
  - a[i] is in its final place in the array for some i
  - none in a[1], ... , a[i-1] is greater than a[i]
  - none in a[i+1], ... , a[r] is less than a[i]

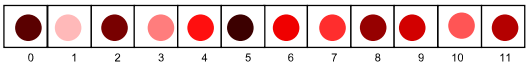
- apply quicksort recursively to each part independently

Readings

- [Mar07] Read 7.7, 7.8
- [Mar13] Read 7.6, 7.7

QuickSort in Java

```
public static <E> void quickSort(E[] data, int low, int high,
                                 Comparator<E> comp){
    if (high-low < 2)      // only one item to sort.
        return;
    else {
        // split into two parts,  mid = index of boundary
        int mid = partition(data, low, high, comp);
        quickSort(data, low, mid, comp);
        quickSort(data, mid, high, comp);
    }
}
```



Reflection upon Quicksort

- Quicksort makes use of ONE pivot element for partitioning.
- What if more than one pivot is used for partitioning?
- Is it feasible?

- What are the advantages of multi-pivot quicksort if feasible?

Summary

- Sorting
  - Design by Divide and Conquer
  - Merge Sort
  - QuickSort

Analysing Sorting Algorithms

Lecture 19

Menu

- Analysing Fast Sorting Algorithms
  - MergeSort
  - QuickSort

Sorting Algorithm costs:

- Insertion sort, Selection Sort, Bubble Sort:
  - All slow (except Insertion sort on almost sorted lists)
  - **O(n^2)**
- Merge Sort?
- Quick Sort?
  - There's no inner loop!
  - How do you analyse recursive algorithms?

MergeSort

```
private static <E> void mergeSort(E[] data, E[] temp, int low, int high,
                                  Comparator<E> comp){
  if (high > low+1){
      int mid = (low+high)/2;
      mergeSort(temp, data, low, mid, comp);
      mergeSort(temp, data, mid, high, comp);
      merge(temp, data, low, mid, high, comp);
  }
}
```

- Cost of mergeSort:
  - Three steps:
    - first recursive call: cost?
    - second recursive call: cost?
    - merge: has to copy over (high-low) items

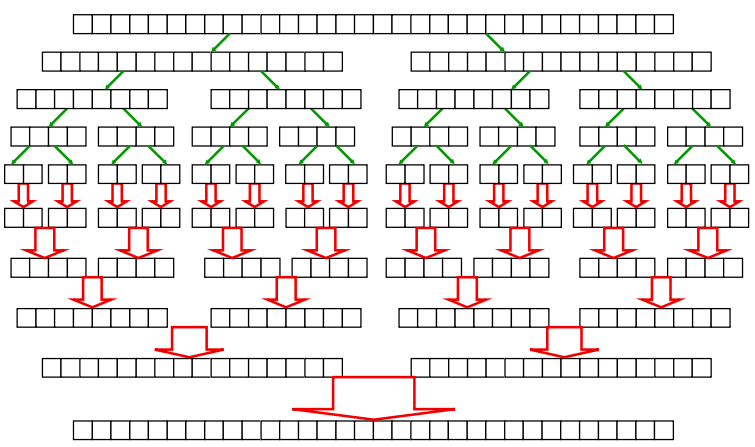
QuickSort

```
public static <E> void quickSort(E[] data, int low, int high,
                                 Comparator<E> comp){
   if (high > low +2){
      int mid = partition(data, low, high, comp);
      quickSort(data, low, mid, comp);
      quickSort(data, mid, high, comp);
   }
}
```

Cost of Quick Sort:

- three steps:
  - partition: has to compare (high-low) items
  - first recursive call
  - second recursive call

MergeSort Cost (real order)



QuickSort Cost

- If Quicksort divides the array exactly in half, then:
  - C(n) = n + 2 C(n/2)
    = n log(n) comparisons = O(n log(n))

  (best case)
- If Quicksort divides the array into 1 and n-1:
  - C(n) = n + (n-1) + (n-2) + (n-3) + … + 2 + 1
    = n(n-1)/2 comparisons = $O(n^2)$

  (worst case)
- Average case?
  - Very hard to analyse.
  - Still O(n log(n)), and very good.
- Quicksort is "in place", MergeSort is not

MergeSort Cost

- Level 1: 2 * n/2 = n
- Level 2: 4 * n/4 = n
- Level 3: 8 * n/8 = n
- Level 4: 16 * n/16 = n
- Level k: n * 1 = n
- How many levels?
- Total cost? = O( )
- n = 1,000:
  n = 1,000,000
  n = 1,000,000,000

Where have we been?

Implementing Collections:

- ArrayList: O(n) to add/remove, except at end
- Stack: O(1)
- ArraySet: O(n) (cost of searching)
- SortedArraySet O(log(n)) to search (with binary search)
  O(n) to add/remove (cost of inserting)
  $O(n^2)$ to add n items
  O(n log(n)) to initialise with n items.
  (with fast sorting)

Analysing with Recurrence Relations

```
private static <E> void mergeSort(E[] data, E[] temp, int low, int high,
                                  Comparator<E> comp){
  if (high > low+1){
      int mid = (low+high)/2;
      mergeSort(temp, data, low, mid, comp);
      mergeSort(temp, data, mid, high, comp);
      merge(temp, data, low, mid, high, comp);
  }
}
```

- Cost of mergeSort = C(n)
  - C(n) = C(n/2) + C(n/2) + n
    = 2 C(n/2) + n
- Recurrence Relation:
  - Solve by repeated substitution & find pattern
  - Solve by general method

Summary

- Analysing Fast Sorting Algorithms
  - MergeSort
  - QuickSort

Solving Recurrence Relations

C(n) = 2 C(n/2) + n

= 2 [2 C(n/4) + n/2] + n

= 4 C(n/4) + 2 * n

= 4 [2 (C(n/8) + n/4] + 2 * n

= 8 C(n/8) + 3 * n

= 16 C(n/16) + 4 * n

:

$= 2^k C(n/2^k) + k * n$

when $n = 2^k$, $k = \log(n)$

= n C (1) + log(n) * n

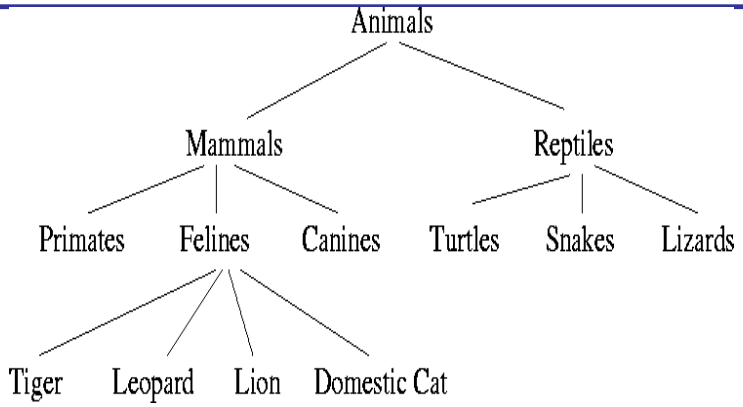
since C(1) = fixed cost

C(n) = log(n) * n

Readings

- [Mar07] Read 2.3
- [Mar13] Read 2.3

Example: Taxonomy Tree



Introduction to Trees

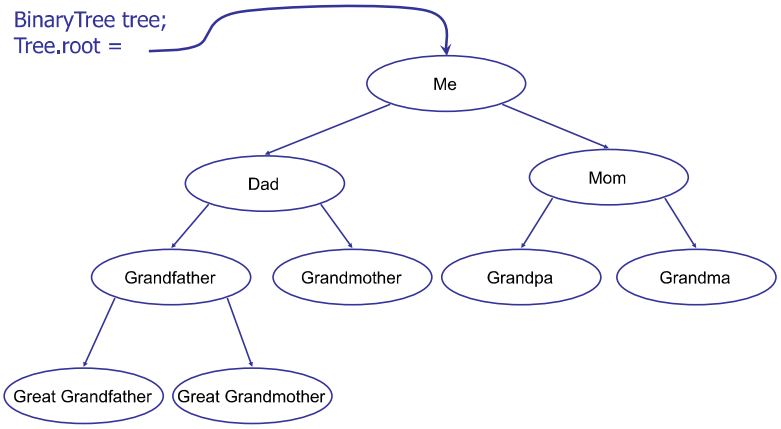
Lecture 20

Trees

- linear data structures: lists, stacks, queues
- non-linear data structures: trees

Menu

Binary Tree



Trees

- Some data are not linear (it has more structure!)
  - –Family trees
  - –Organisational charts
  - –…
- Linear implementations are sometimes inefficient
  - –Linked lists don't store such structure information
- Trees offer an alternative
  - –Representation
  - –Implementation strategy
  - –Set of algorithms

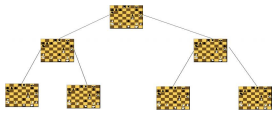
What is a tree exactly?

- Models the parent/child relationship between different elements
  - –Each child only has one parent
  - –Each parent has ? children
- From mathematics:
  - –A “directed acyclic graph” (DAG)
  - –At most one path from any one node to any other node
- Different kinds of trees exist
- Trees can be used for different purposes

- In what order do we visit elements in a tree?

Terminology

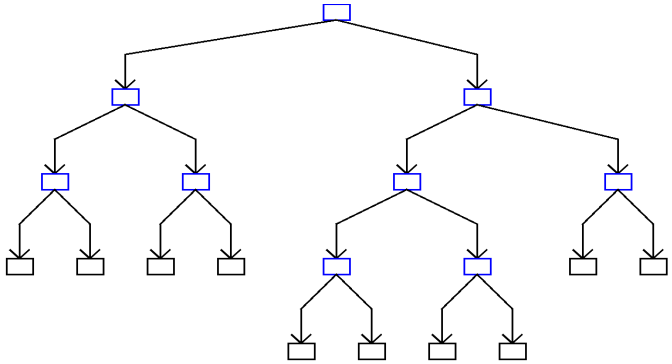
- Level:
  - –Equivalent to the “row” that a value is in
- Depth:
  - –The number of levels in the tree
- Leaf Nodes
  - –A node with no children
- Non-leaf Nodes
- Balanced:
  - –All leaf nodes are on levels n and (n+1), for some value of n (*and are grouped on the bottom level “to the left”*)



How many types of tree are there?

- Far too many:
  - –Red-Black Tree
  - –AVL Tree
  - –…
- Different types are used for different things
  - –To improve speed
  - –To improve the use of available memory
  - –To suit particular problems
- But we’ll look at three:
  - –Binary Tree
  - –General Tree (n-ary tree)
  - –AVL Tree

Binary Tree

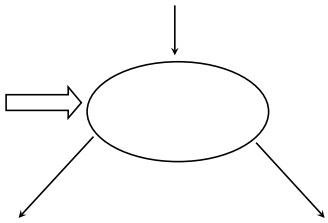


Size is limited by the depth of the tree.

Binary Tree

- Every node in the tree has a left child (or null)
- Every node in the tree has a right child (or null)
- The root of the tree is just another node

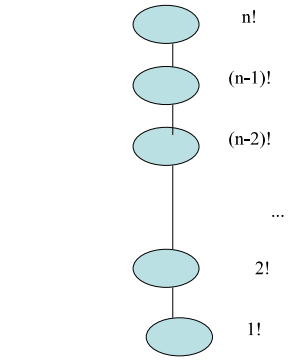
This is what you
Make a class for:



Binary tree

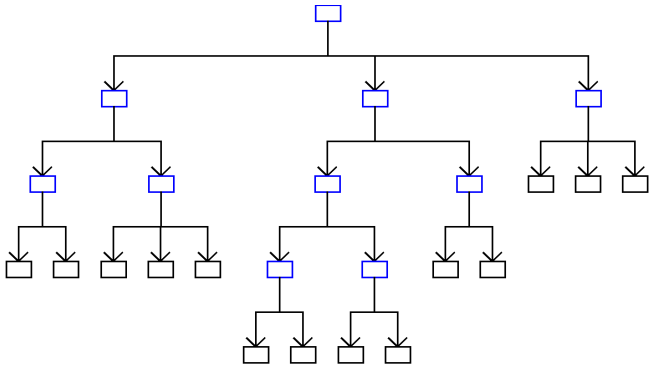
- A binary tree is either:
  - –empty or
  - –a root node together with two binary trees - left subtree & right subtree of the root

Recursion tree for n!



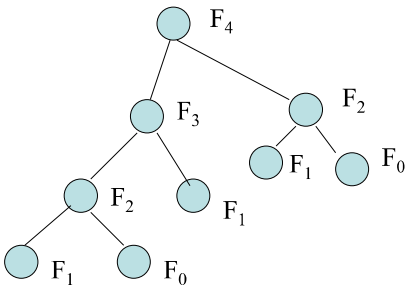
recursion tree for n!

General Trees



Question: Why is this *not* a Binary Tree?

Recursion tree for fibonacci(4)



Design Issue - Ordering of Values

- Do the children of a parent have a particular order?
  - –Does it matter which is the “first”, “second”, or “third” child?
  - –Is there an inherent ordering there?
- Are the values of the children comparable with the values of the parent?

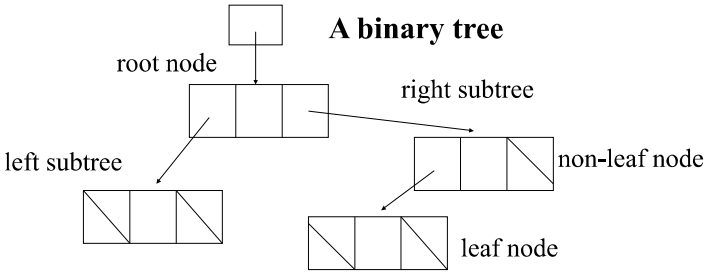
What is a tree useful for?

- Artificial Intelligence – planning, navigating, games
- Representing things:
  - –Simple file systems
  - –Class inheritance and composition
  - –Classification, e.g. taxonomy (the is-a relationship)
  - –HTML pages
  - –Parse trees for languages
  - –Essential in compilers like Java, C# etc.
  - –3D graphics (e.g. BSP trees)

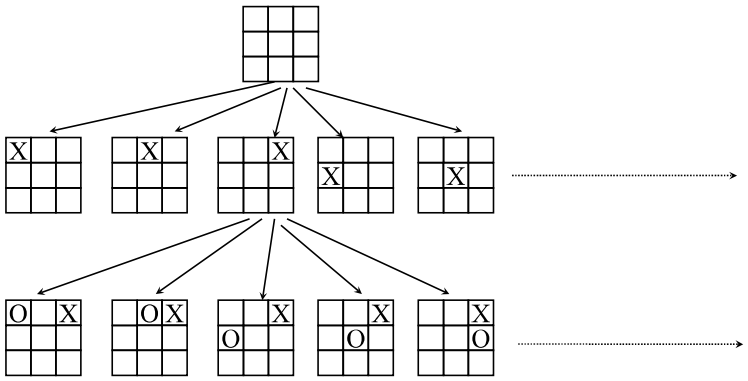
Trees and Recursion

- Recursively defined data structure
- Recursion is very natural for trees – important!
- Recursion tree
- If you don’t use recursion then use iteration

Binary tree implementation



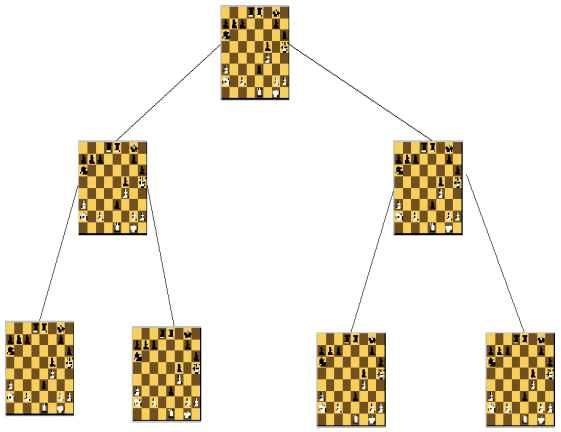
Example: Tic Tac Toe



In brief

- Data representation of tree is important
- Efficiently adding, accessing and removing data from a tree is important
- Trees can be made efficient and help you organise data
- Trees are useful in:
  - –Computer graphics
  - –Artificial Intelligence
  - –Databases
  - –…

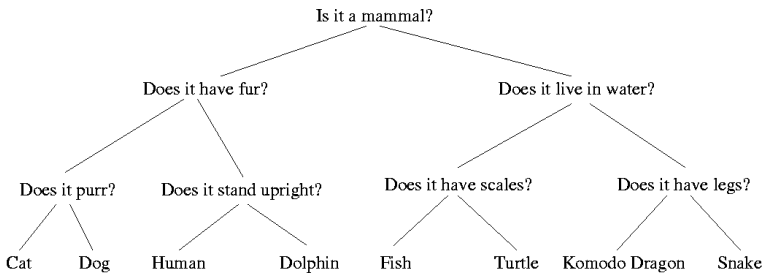
Example: Chess



Q&A

- Tree represents an efficient 1-dimensional data structure. (T or F?)
- A leaf node in a tree has no children. (T or F?)
- Binary tree has no ordering upon its sibling nodes. (T or F?)
- Name 3 applications for tree.
- Relationship among recursive calls can be expressed in what type of tree?

Example: Decision Tree



Summary

- Introduction to Trees
  - What are trees?
- Binary Tree
- General Tree
- Terminology
- Different Types of Tree
- Tree Ordering
- Trees and Recursion
- What are they useful for?
  - Tic Tac Toe example
  - Chess
  - Taxonomy Tree
  - Decision Tree

Readings

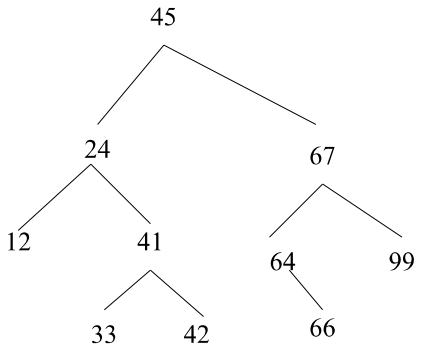
- [Mar07] Read 4.1, 4.2, 4.6
- [Mar13] Read 4.1, 4.2, 4.6

Search Lists

- a linked list with key, info, and next
- **O(N)** in average for insert, lookup, and remove

Binary Search Trees
Lecture 21

Binary Search Tree



Menu

- Maps
- Search lists
- Binary search trees
- Tree traversal
  - Preorder
  - Inorder
  - Postorder
- Balanced Search Trees
  - AVL Trees

Binary Search Tree Definitions

- A binary search tree is a binary tree where each node has a key
- The key in the left child (if exists) of a node is less than the key in the parent
- The key in the right child (if exists) of a node is greater than the key in the parent
- The left & right subtrees of the root are again binary search trees

Tables (Maps)

- indexed container
- Associate information with a key
  - *key* is often a character string
  - *info* is any information
- E.g. a phone book
  - key is the name of a person or business
  - info is their phone number & address
- Typical Operations on Tables

```
void insert(string key, Object o);
object lookup(string key);
void remove(string key);
```

- Alternative implementations include
  - Search Lists
  - Binary Search Trees
  - Hash Tables

Binary Tree Traversal

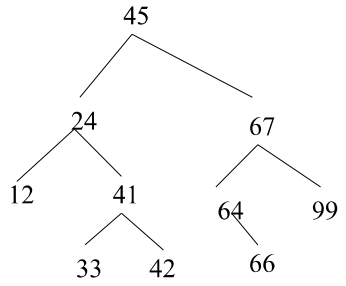
- inOrder
- preOrder
- postOrder

Binary Search Trees (BST)

- similar to a linked list, but two next pointers
- we call them *left* and *right*
- for each node n, with key k
  - n->left contains only nodes with keys < k
  - n->right contains only nodes with keys > k
- **O(log N)** in average for insert, lookup, and remove

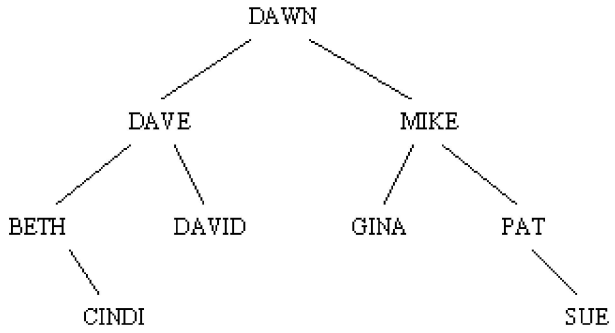
inOrder traversal: recursive

- traverse the left subtree inOrder
- process (display) the value in the node
- traverse the right subtree inOrder



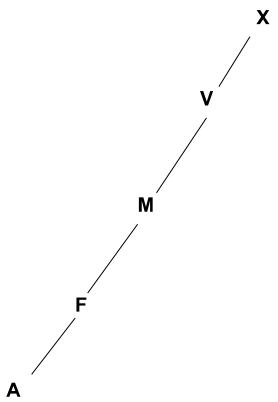
Worst Cases

- operations can degenerate to O(N) – worst case!
- degenerates to a linked list
  - when keys are inserted in ascending order
    - all keys are to the right
  - when keys are inserted in descending order
    - all keys are to the left
- ideal is mid first, then successive middles, etc.
- random order also works fairly well

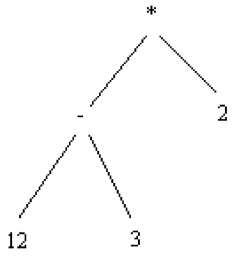


BETH, CINDI, DAVE, DAVID, DAWN, GINA, MIKE, PAT, SUE

Degenerated Tree

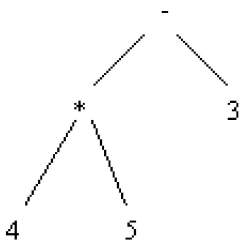


inorder traversal of the binary expression tree for (12-3)*2



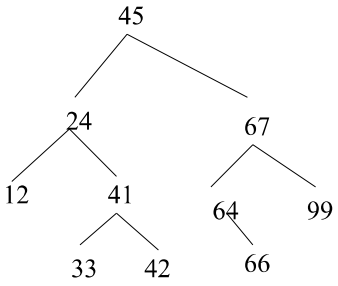
12 – 3*2

Exercise: inorder traversal of the binary expression tree for 4 * 5 - 3

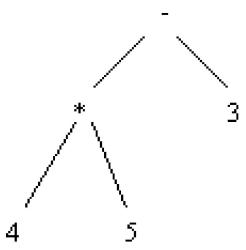


preOrder traversal: recursive

- process (display) the value in the node
- traverse the left subtree preOrder
- traverse the right subtree preOrder
- ?

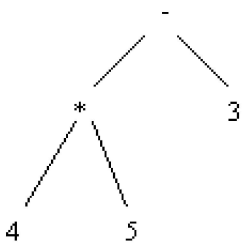


inorder traversal of the binary expression tree for 4 * 5 - 3

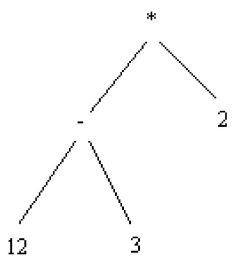


4 * 5 - 3

Exercise: preorder traversal of the binary expression tree for 4 * 5 - 3

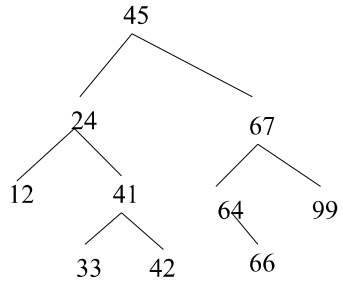


Exercise: inorder traversal of the binary expression tree for (12-3)*2

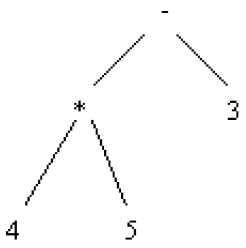


postOrder traversal: recursive

- traverse the left subtree postOrder
- traverse the right subtree postOrder
- process (display) the value in the node
- ?

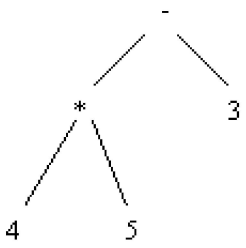


preorder traversal of the binary expression tree for 4 * 5 - 3



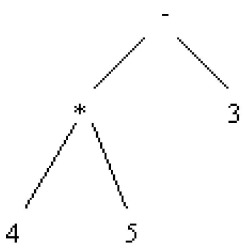
- * 4 5 3

Exercise: postorder traversal of the binary expression tree for 4 * 5 - 3



Ex: How would you draw the subtree for (4-5)*3 to be evaluated correctly in preorder?

postorder traversal of the binary expression tree for 4 * 5 - 3

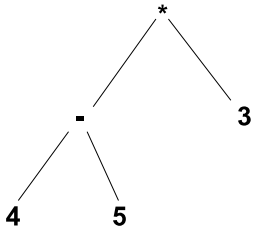


4 5 * 3 -

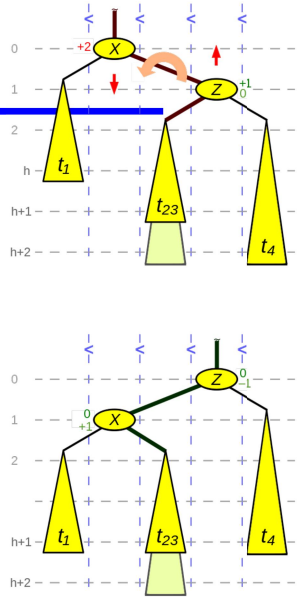
Ex: How do you construct the binary tree for 4-5*3 to be evaluated correctly in postorder?

Ex: How would you draw the subtree for (4-5)*3 to be evaluated correctly in preorder?

- Correct evaluation in preorder should be: * - 4 5 3
- Corresponding tree:



Simple rotation *rotate_Left(X,Z)*



Breadth-First traversal

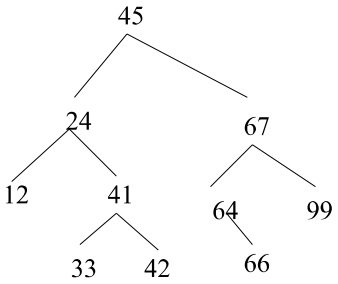
- all previous traversals are *Depth-First* traversals
- visit all the nodes at depth 0, then depth 1, etc.
- may use a **queue** to traverse across levels

AVL Trees (Adelson-Velskii and Landis)

- An AVL Tree is a form of binary search tree
- Unlike a binary search tree, the worst case scenario for a search is O(log n).
- AVL data structure achieves this property by placing restrictions on the difference in height between the sub-trees of a given node - height balanced to within 1
- and re-balancing the tree if it violates these restrictions.

Breadth-First traversal

- ?



AVL Tree Balance Requirements

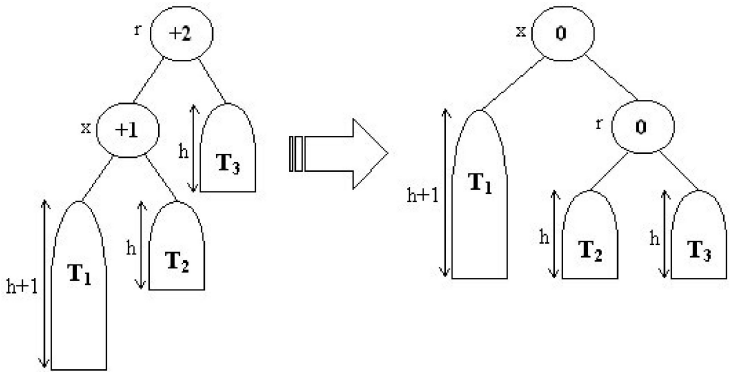
- A node is only allowed to possess one of three possible states:
- **Left-High (balance factor -1)**
  The left-sub tree is one level taller than the right-sub tree
- **Balanced (balance factor 0)**
  The left and right sub-trees both have the same heights
- **Right-High (balance factor +1)**
  The right sub-tree is one level taller than the left-sub tree

- If the balance of a node becomes -2 or +2 it will require re-balancing.
- This is achieved by performing a **rotation** about this node
- **Rotation does not break the existing properties for a search tree**

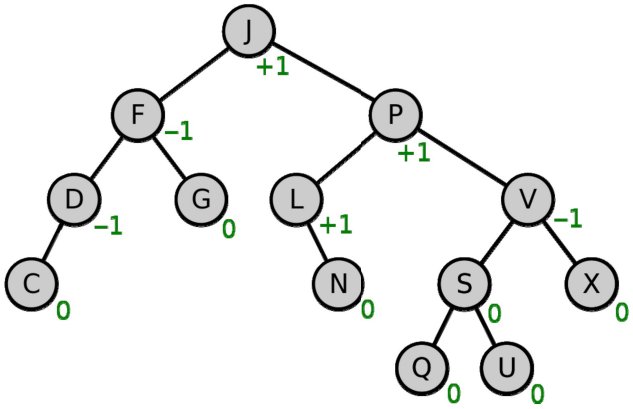
Balanced Search Trees

- use **rotations** to ensure tree is always 'full'
- prevents degenerative cases mentioned above
- truely O(log N) worst case for insert, lookup, remove
- insertion/removal takes more time
- but lookup is faster
- trickier to code correctly

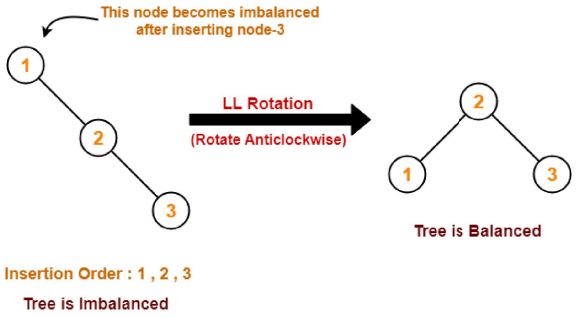
AVL Rotation - RR



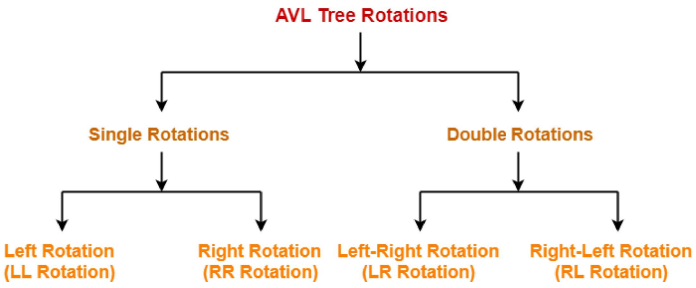
AVL tree with balance factors



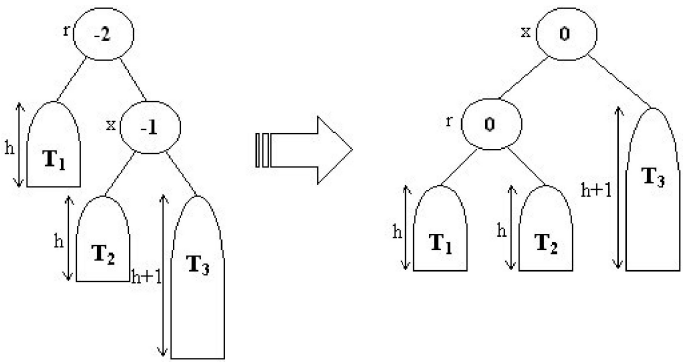
AVL Rotation - LL



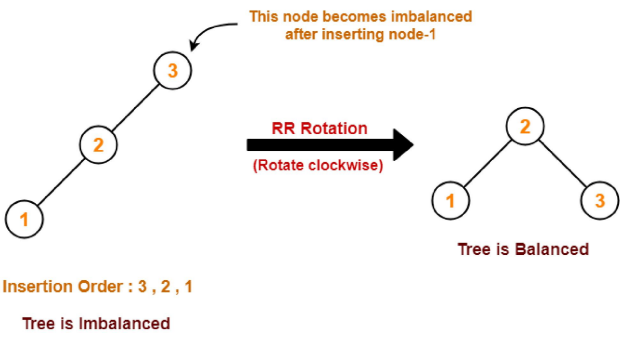
AVL Tree Rotations



AVL Rotation - LL



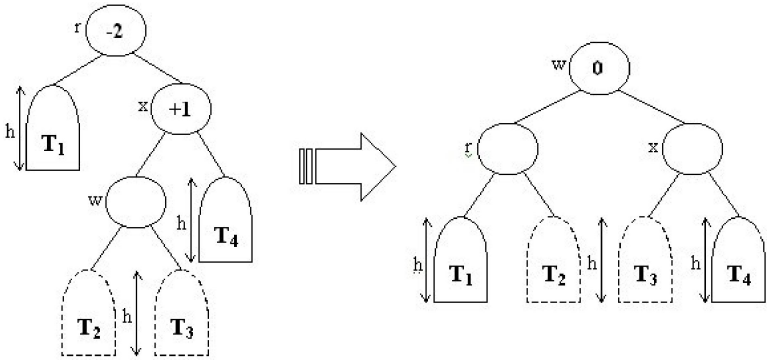
AVL Rotation - RR



AVL Tree Insertion

- AVL requires two passes for insertion:
- one pass down tree (to determine insertion)
- one pass back up to update heights and rebalance

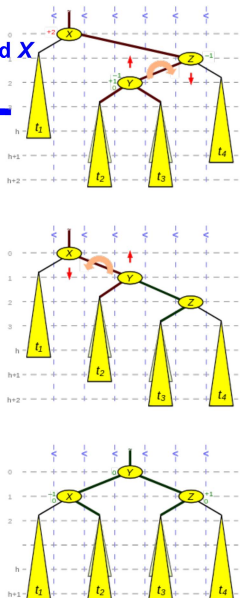
AVL Rotation - RL



AVL Tree Insertion

- Animation showing the insertion of several elements into an AVL tree. It includes left, right, left-right and right-left rotations.

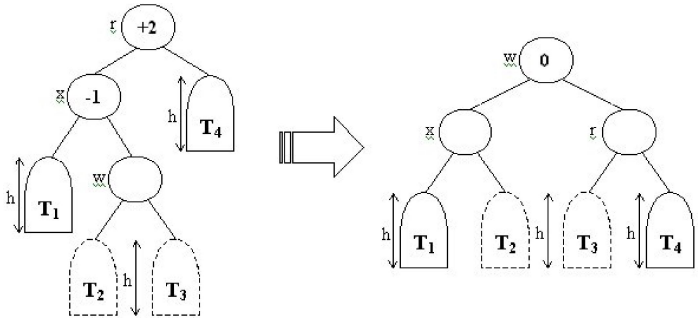
AVL Double rotation *rotate_RightLeft(X,Z)* = *rotate_Right* around *Z* followed by *rotate_Left* around *X*



AVL Time complexity in big O notation

Algorithm	Average	Worst case
Search	O(log *n*)	O(log *n*)
Insert	O(log *n*)	O(log *n*)
Delete	O(log *n*)	O(log *n*)
Space	O(*n*)	O(*n*)

AVL Rotation - LR



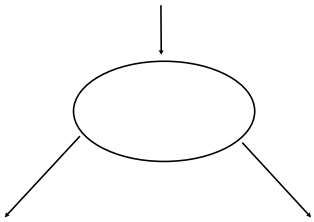
Summary

- Maps
- Search lists
- Binary search trees
- Tree traversal
  - Preorder
  - Inorder
  - Postorder
- Balanced Search Trees
  - AVL Trees

Readings

- [Mar07] Read 4.2, 4.3, 4.4, 4.8, 9.6
- [Mar13] Read 4.2, 4.3, 4.4, 4.8

What does the Binary Tree ADT/class need?



Implementing Trees

Lecture 22

Binary Tree

- Each node contains a value
- Each node has a left child and a right child (may be null)

```
public class BinaryTree<V> {

   private V value;
   private BinaryTree<V> left;
   private BinaryTree<V> right;

   /**Creates a new tree with one node
        (a root node) containing the specified value  */
   public BinaryTree(V value) {
     this.value = value;
   }
```


Menu

- Abstract Data Type (ADT) for Trees
- Implementing Binary Trees: issues & considerations, data structure?
- Implementing General Trees: issues & considerations, data structure?

- Applications for Trees
- Tree Traversal

Get and Set

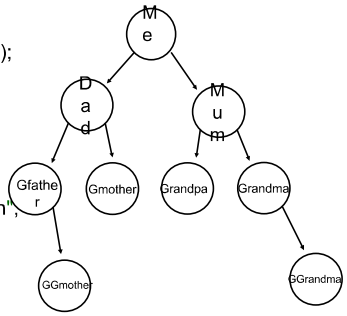
```
  public V getValue() {
      return value;
   }
   public BinaryTree<V> getLeft() {
      return left;
   }
   public BinaryTree<V> getRight() {
      return right;
   }
   public void setValue(V val) {
       value = val;
   }
   public void setLeft(BinaryTree<V> tree) {
       left = tree;
   }
   public void setRight(BinaryTree<V> tree) {
       right = tree;
   }
```


Binary Tree ADT

```
public interface BinaryTree<T> {
    BinaryTree<T> BinaryTree(T value);

    BinaryTree<T> getLeft();
    BinaryTree<T> getRight();
    void setLeft(BinaryTree<T> subtree);
    void setRight(BinaryTree<T> subtree);

    BinaryTree<T> find(T value);
    boolean contains(T value);

    boolean isEmpty();
    boolean isLeaf();
    int size();
    }
```

Using BinaryTree

```
BinaryTree<String> ma = myTree;
   while(ma.getRight()!=null)   ma = ma.getRight();

BinaryTree<String> pa = myTree;
    while(pa.getLeft()!=null)   pa = pa.getLeft();

System.out.format(
       "paternal ansc = %s, maternal ansc = %s\n\n",
       pa.getValue(), ma.getValue());
```



What would be the results of execution?

paternal ans = Gfather, maternal ans = GGrandma

isLeaf and find

```
public boolean isLeaf() {
    return (right == null) && (left == null);
 }

 public BinaryTree<V> find(V val) {
    if (value.equals(val))      return this;
    if (left != null) {
        BinaryTree<V> ans = left.find(val);
        if (ans != null)        return ans;
    }
    if (right != null) {
        BinaryTree<V> ans = right.find(val);
        if (ans != null)        return ans;
    }
    return null;
  }
}
```


Using BinaryTree

```
public static void printAll(BinaryTree<String> tree, String indent){
   System.out.println(indent+ tree.getValue());
   if (tree.getLeft()!=null)
     printAll(tree.getLeft(), indent+"  " );
   if (tree.getRight()!=null)
     printAll(tree.getRight(), indent+"  " );
}
```

What traversal scheme is this?

PreOrder traversal

Using BinaryTree

- A method that constructs a tree

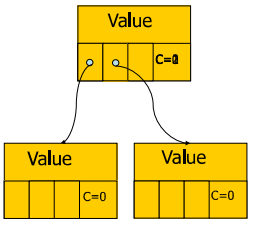
```
public static void main(String[] args){
   BinaryTree<String> myTree = new BinaryTree<String>("Me");
   myTree.setLeft(new BinaryTree<String>("Dad"));
   myTree.setRight(new BinaryTree<String>("Mum"));
   myTree.getLeft().setLeft(new BinaryTree<String>("Gfather"));
   myTree.getLeft().setRight(new BinaryTree<String>("Gmother"));
   myTree.getRight().setLeft(new BinaryTree<String>("Grandpa"));
   myTree.getRight().setRight(new BinaryTree<String>("Grandma"));
   myTree.getRight().getRight().setRight(
            new BinaryTree<String>("GGrandma"));

   BinaryTree<String> gf = myTree.find("Gfather");
   if (gf!=null)
      gf.setRight(new BinaryTree<String>("GGmother"));
}
```

Ex: Show the final tree contents after the above program execution.

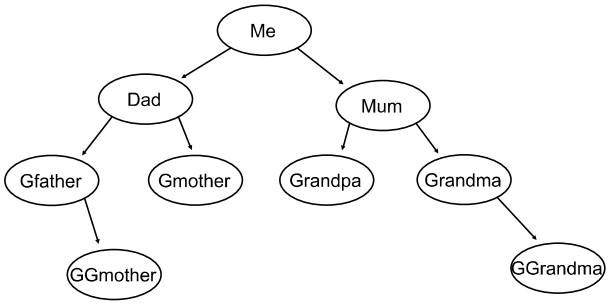
General Tree

- A node in the tree can have any number of children
- We keep the children in the order that they were added.

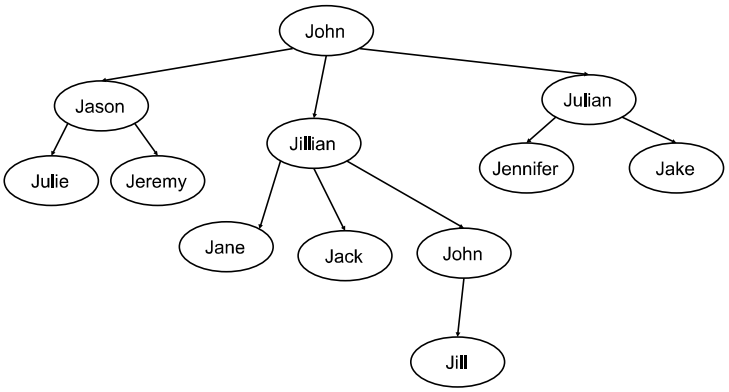
```
public class GeneralTree<V> {

   private V value;
   private List< GeneralTree<V> > children;

   public GeneralTree(V value) {
      this.value = value;
      this.children = new ArrayList< GeneralTree<V> >();
   }
...
}
```



myTree



The Tree



Get

- Children are ordered, so can ask for the *i* th child

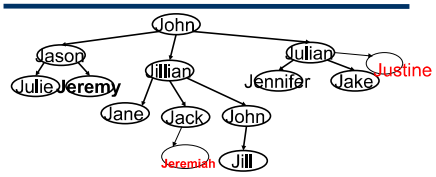
```
public V getValue() {
   return value;
}

public List< GeneralTree<V> > getChildren() {
   return children;
}

public GeneralTree<V> getChild(int i) {
   if (i>=0 && i < children.size())
      return children.get(i);
   else
      return null;
}
```


More Adding

```
GeneralTree<String> thrd = mytree.getChildren().get(2);
thrd.addChild(new GeneralTree<String>("Justine"));

GeneralTree<String> gc = mytree.find("Jack");
if (gc!=null)
    gc.addChild(new GeneralTree<String>("Jeremiah"));

System.out.println ("2nd child of 1st child: "+
                    mytree.getChild(0).getChild(1).getValue());
```

Jeremy

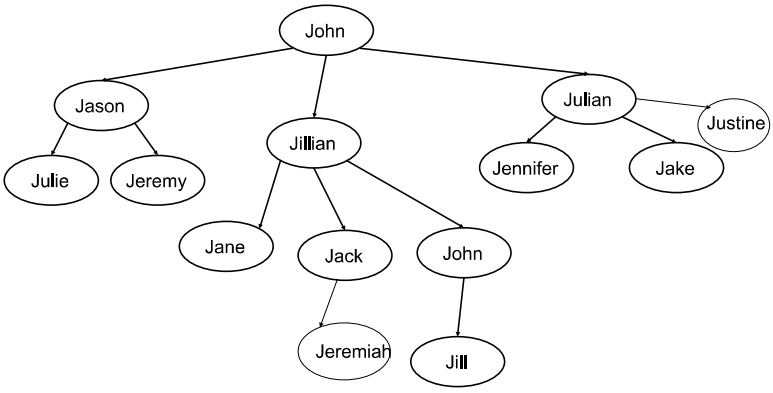
mytree = ?



add and find

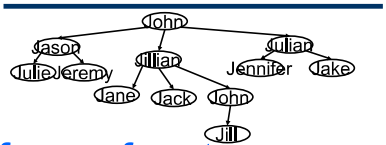
```
public void addChild(GeneralTree<V> child) {
    children.add(child);
 }
 public void addChild(int i, GeneralTree<V> child) {
    children.add(i, child);
 }

public GeneralTree<V> find(V val) {
    if (value.equals(val))       return this;
    for(GeneralTree<V> child : children){
        GeneralTree<V> ans = child.find(val);
        if (ans != null)          return ans;
    }
    return null;
}
```


The Tree



Using General Tree

```
public static void main(String[] args){
 GeneralTree<String> mytree = new GeneralTree<String>("John");
    mytree.addChild(new GeneralTree<String>("Jason"));
    mytree.addChild(new GeneralTree<String>("Jillian"));
    mytree.addChild(new GeneralTree<String>("Julian"));
    mytree.getChildren().get(0).addChild(new GeneralTree<String>("Julie"));
    mytree.getChildren().get(0).addChild(new GeneralTree<String>("Jeremy"));
    mytree.getChildren().get(1).addChild(new GeneralTree<String>("Jane"));
    mytree.getChildren().get(1).addChild(new GeneralTree<String>("Jack"));
    mytree.getChildren().get(1).addChild(new GeneralTree<String>("John"));
    mytree.getChildren().get(2).addChild(new GeneralTree<String>("Jennifer"));
    mytree.getChildren().get(2).addChild(new GeneralTree<String>("Jake"));
    mytree.getChildren().get(1).getChildren().get(2).addChild(new
            GeneralTree<String>("Jill"));
}
```



Ex. Draw the final form of mytree

printAll

```
printAll(mytree, "");

public static void printAll(GeneralTree<String> tree, String indent){
    System.out.println(indent+ tree.getValue());
    for(GeneralTree<String> child : tree.getChildren())
      printAll(child, indent+"  " );
  }
```

What traversal scheme is this?

PreOrder traversal

Summary

- Investigated the design and implementation of binary tree and general tree
  - Issues
  - Data Structure
- Tree class design and implementation is quite simple… if you get the ideas of recursion
- Every algorithm can be expressed iteratively or recursively
- One way is usually much better than the other!
  - In trees, recursion is normally nicer than iteration (but not always)

Comparison among various search algorithms – Linear Search

	name	number
0	Parker	12345
1	Davis	43534
2	Harris	32452
3	Corea	46532
4	Hancock	96562
5	Brecker	37811
6	(empty)	
…	…	
N-1	Marsalis	54323

Linear Search = O(N)

Lecture 23
Hashing and Hash Tables

Comparison among various search algorithms – Binary Search

	name	number
0	Brecker	37811
1	Corea	46532
2	Davis	43534
3	Hancock	96562
4	Harris	32452
5	Marsalis	54323
6	Parker	12345
7	(empty)	
…	…	
N-1	(empty)	

Binary Search = O(log(N))

Menu

- Hash tables
- Comparison among various search mechanisms
- Table size
- Hash function
- Modular hash function
- Hash function examples
- Collisions

Comparison among various search algorithms – Hash Table

name	hash
Brecker	6
Corea	2
Davis	2
Hancock	12
Harris	12
Marsalis	2
Parker	8

Hash code	name/number
0	
1	
2	Corea/46532, Davis/43534, Marsalis/54323
3	
4	
5	
6	Brecker/37811
7	
8	Parker/12345
9	
10	
11	
12	Hancock/96562, Harris/32452

Hash = O(1) if no collision
hash function is simply the sum of ASCII codes of characters in a name (considered all in lowercase) computed mod N=13.

Hash Tables

- another kind of Table
- **O(1)** in average for insert, lookup, and remove
- use an array named T of *capacity* N
- define a hash function that returns an integer int **H(string key)**
- must return an integer between 0 and N-1
- store the key and info at T[H(key)]
- H() must always return the same integer for a given key

Hash Functions

- a good hash function has the following characteristics:

- avoids collisions
- spreads keys evenly in the array
- inexpensive to compute - must be O(1)

Table Size

- Table size is usually *prime* to avoid bias
- overly large table size means wasted space
- overly small table size means more collisions
- what happens as table size approaches 1?

Hash Function for Signed Integer Keys

- remainder after division by table length

```
int hash(int key, int N) {
return abs(key) % N;
}
```

- if keys are positive, you can eliminate the abs

What is a Hash Function?

- A **hash function** is any well-defined procedure or mathematical function for turning data into an index into an array.
- The values returned by a hash function are called **hash values** or simply **hashes.**
- A hash function H is a transformation that
  - takes a variable-size input k and
  - returns a fixed-size string (or int), which is called the ***hash value*** h (that is, h = H(k))

- In general, a hash function may map several different keys to the same hash value.

Hash Functions for Strings

- must be careful to cover range from 0 through capacity-1
- some poor choices
  - summing all the ASCII codes
  - multiplying the ASCII codes
- important insight
  - letters and digits fall in range 0101 and 0172 octal
  - so all useful information is in lowest 6 bits
- hash(s) is O(1)

Example of a Modular Hash Function

- H(k) = **k mod m** (or **k % m**)
- message1 = '723416'
- hash function = modulo 11
- Hash value$_1$ = (7+2+3+4+1+6) mod 11 = 1

- message2 = 'test' = ASCII '74', '65', '73', '74'
- Hash value$_2$ = (74+65+73+74) mod 11 = 0
- another hash function example: **a*k mod m**

Dealing with Collisions

- ***open addressing*** - collision resolution
- key/value pairs are stored in array slots
- ***linear probing***
  - hash(k, i) = (hash1(k) + i) mod N
  - increment hash value by a constant, 1, until free slot is found
  - simplest to implement
  - leads to *primary clustering*
- ***quadratic probing***
  - hash(k, i) = (hash1(k) + c_1*i + c_2*i*i) mod N
  - leads to *secondary clustering*
- ***double hashing***
  - hash(k, i) = (hash1(k) + i*hash2(k)) mod N
    - avoids clustering

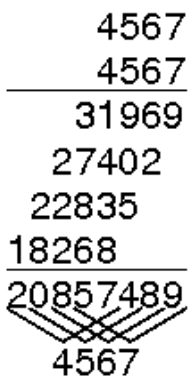
Hash Functions for Integer Keys

- *Mid-square* method
- Squaring the key value first, and then takes out the middle *r* bits of the result, giving a value in the range 0 to 2^r-1.
- This works well because most or all bits of the key value contribute to the result

Dealing with Collisions

- *separate chaining*
- each array slot is a SearchList
- never gets 'full'
- deletions are not a problem

Mid-square Method



Dealing with A Full Table

- allocate a larger hash table
- rehash each from the smaller into the larger
- delete the smaller

Hash Functions for String Keys

- This function takes a string as input. It processes the string 4 bytes at a time, and interprets each of the four-byte chunks as a single long integer value.
- The integer values for the 4-byte chunks are added together.
- The resulting sum is converted to the range 0 to *M*−1 using the modulus operator.
- There is nothing special about using 4 characters at a time. Other choices could be made.

Why hash table can give us O(1) performance?

- It appears most search mechanisms have performance at O(N) or O(logN)
- So why hash table can give us best performance?
- Where does the magic come from?

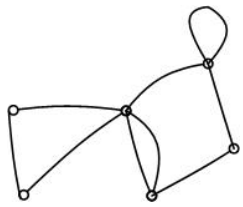
Summary

- Hash tables
- Comparison among various search mechanisms
- Table size
- Hash function
- Modular hash function
- Hash function examples
- Collisions

Readings

- [Mar07] Read 5.1-5.4, 5.6
- [Mar13] Read 5.1-5.4, 5.6

Give the number of vertices and the number of edges of the following graph:



- $|V| = 6$;
- $|E| = 9$.

Introduction to Graph Theory
Lecture 24

Incidence, adjacency and neighbors

- Two vertices are **adjacent** if they are joined by an edge.
- Adjacent vertices are said to be **neighbors.**
- The edge which joins vertices is said to be **incident** to them.

Menu

- Basic definitions of graph theory
- Properties of graphs
- Paths
- Trees
- Digraphs and their applications, network flows

Multiple edges, loops and simple graphs

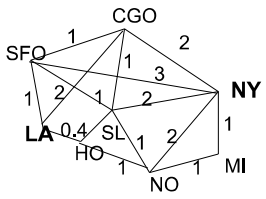
- Two or more edges joining the same pair of vertices are **multiple edges.**
- An edge joining a vertex to itself is called a **loop.**
- A graph containing no multiple edges or loops is called a **simple graph**

Definitions

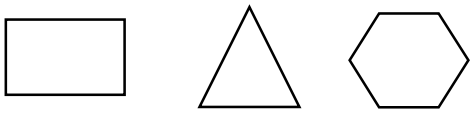
- A **graph** G consist of :
  - a *finite* set of **vertices** $V(G)$, which cannot be empty,
  - and a *finite* set of **edges** $E(G)$, which connect pairs of vertices.
- The number of vertices in G is called the **order** of G, denoted by $|V|$.

Why study graph theory?

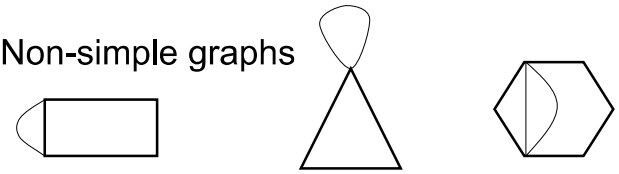
- Routing problem: find the minimal delay path from LA to NY.



Simple graph: examples

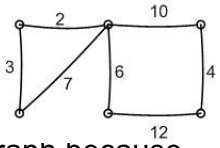


- Non-simple graphs



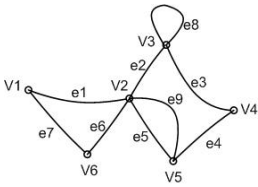
Weighted graphs

- A **weighted graph** has a number assigned to each of its edges, called its **weight.**
- The weight can be used to represent distances, capacities or costs.
- Is the following weighted graph a simple graph? Justify your answer



- The weighted graph is a simple graph because it has no multiple edges or loops

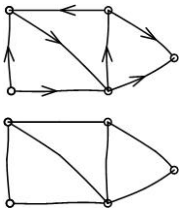
In the following graph:
Identify the neighbours of *V*4
Identify the edge incident to *V*3 and *V*4
Identify multiple edges
Identify the loop



- The neighbours of *V*4 are: *V*3 and *V*5
- The edge incident to *V*3 and *V*4 is: *e*3
- *e*5 and *e*9 are multiple edges
- *e*8 is a loop

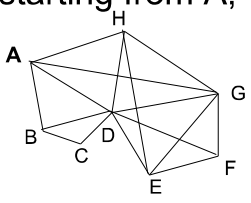
Digraphs

- A **digraph** is a *directed* graph, a graph where instead of edges we have directed edges with arrows (**arcs**) indicating the direction of flow.
- Sketch the underlying graph of the digraph:



Why study graph theory?

- There are many engineering or computer science related problems that can be modelled using 'graphs'
- For example, **travelling salesman problem**: find the minimal cost path to cover all the cities A-H, starting from A, ending at A



Degree

- For a digraph we get

$$\sum_i d_-(V_i) = |A|$$

$$\sum_i d_+(V_i) = |A|$$

- where |A| is the number of arcs.

Degree

- The *number of times* edges are incident to a vertex *V* is called its **degree**, denoted by *d*(*V*).
- The **degree sequence** of a graph consists of the degrees of the vertices written in non-*increasing order*, with repeats where necessary.
- The sum of the values of the degrees, *d*(*V*), over all the vertices of a simple graph is twice the number of edges:

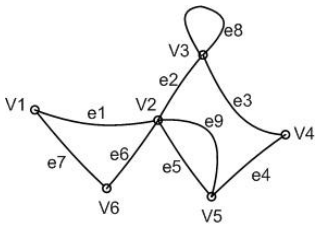
$$\sum_i d(V_i) = 2|E|$$

- Why?

Subgraphs

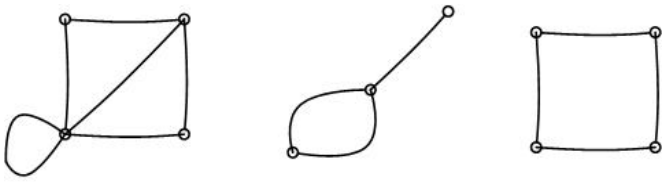
- A **subgraph** of *G* is a graph, *H*, whose vertex set is a subset of *G*'s vertex set, and whose edge set is a subset of the edge set of *G*.
- If a subgraph *H* of *G* spans all of the vertices of *G*, i.e. *V*(*H*) = *V*(*G*), then *H* is called a **spanning subgraph** of *G.*

Give the degrees of the vertices *V*1 and *V*3 of the graph of



- *d*(*V*1) = 2 and *d*(*V*3) = 4

For the graph (a) which of the subgraphs (b) and (c) is a spanning subgraph?



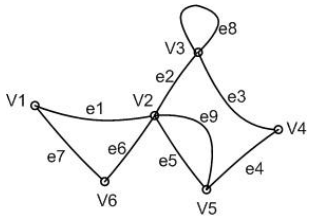
- Subgraph (c) is a spanning subgraph of graph (a).

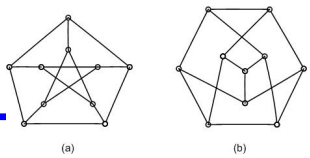
Degree

- A vertex of a digraph has an in-degree of *d*-(*V*) and an out-degree *d*+(*V*).

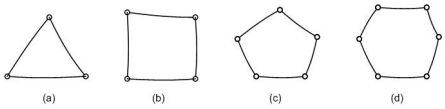
Self test

- 1. Write down the vertex set and edge set of the graph in:

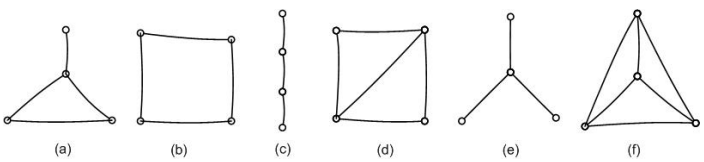




- 2. Which graphs below are subgraphs of those shown above.



- 3. Write down the degree sequence in the graphs below. Verify that the sum of the values of the degrees are equal to twice the number of edges in the graph.



Summary

- Definitions of **graphs: vertices, edges, order**
- Definitions of: **multiple edges, loops**
- Definitions of: **simple graphs**
- **Digraph**: directed graph
- **Weighted graphs**
- The number of times edges are incident to a vertex *V* is called its **degree**

Summary

- The sum of the values of the degrees, $d(V)$, over all the vertices of a simple graph is twice the number of edges: $\sum_i d(V_i) = 2|E|$
- Definitions of: **subgraphs, spanning subgraphs**

Readings

- [Mar07] Read 9.1
- [Mar13] Read 9.1

1. Answers:

- The vertex set is {V1,V2,V3,V4,V5,V6},
- The edge set is {e1,e2,e3,e4,e5,e6,e7,e8, e9}.

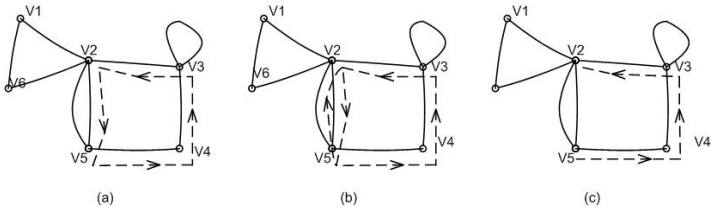
Answers

- 2. (c), (d)

- **Answers**

- 3. (a) (3, 2, 2, 1);
  (b) (2, 2, 2, 2);
  (c) (2, 2, 1, 1);
  (d) (3, 3, 2, 2);
  (e) (3, 1, 1, 1);
  (f) (3, 3, 3, 3);

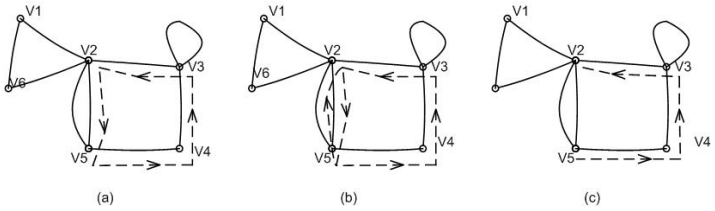
Example: Identify whether a path is marked on the graph in each case:



- Solution: (c) is a path, length?

Introduction to Graph Theory
Lecture 25

Example: Identify whether a trail, path or circuit is marked on the graph in each case:



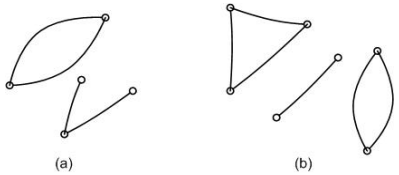
- Solution: (a) circuit (b) trail (c) path

Menu

- Paths
- Connected graphs
- Incidence matrix and adjacency matrix of a graph

Connected graphs

- A graph *G* is **connected** if there is a path from any one of its vertices to any other vertex**.**
- A **disconnected** graph is said to be made up of **components.**
- Example 5:
- How many components do the following disconnected graphs have?

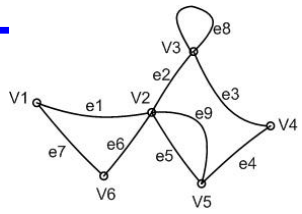


- Solution:
  (a) Two components (b) Three components

Walks, paths and circuits

- A sequence of edges of the form
  V_sV_i, V_iV_j, V_jV_k, V_kV_l, V_lV_t is a **walk** from V_s to V_t.
- If these edges are distinct then the walk is called a **trail**, and
- if the vertices are also distinct then the walk is called a **path.**
- A walk or trail is **closed** if $V_s = V_t$.
- A closed walk in which all the vertices are distinct except V_s and V_t, is called a **cycle** or a **circuit.**
- The number of edges in a walk is called its **length.**

Give the adjacency matrix of the graph below:



- The adjacency matrix for the graph is given by

$$\begin{array}{c} \\ V1 \\ V2 \\ V3 \\ V4 \\ V5 \\ V6 \end{array} \begin{array}{c} \begin{array}{cccccc} V1 & V2 & V3 & V4 & V5 & V6 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 2 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

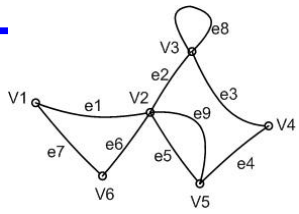
Matrix representation of a graph: the incidence matrix

- The **incidence matrix** of a graph G is a $|V| \times |E|$ matrix $\boldsymbol{A}$.
- The element a_{ij} =
- the *number of times* that vertex V_i is incident with the edge e_j

Summary

- Definitions of **paths**
- Definitions of **connected graphs**
  Definitions of **incidence matrix** and **adjacency matrix** of a graph

Give the incidence matrix of the graph below:



- The incidence matrix for the graph is given by

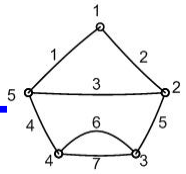
$$\begin{array}{c} \\ V1 \\ V2 \\ V3 \\ V4 \\ V5 \\ V6 \end{array} \begin{array}{c} \begin{array}{ccccccccc} e1 & e2 & e3 & e4 & e5 & e6 & e7 & e8 & e9 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Q. How would you design data structure for graphs? What type of data structure can we use to store graphs?

Matrix representation of a graph: the adjacency matrix

- The **adjacency matrix** of a graph G is a $|V| \times |V|$ matrix $\boldsymbol{A}$.
- The element a_{ij} =
- the *number of edges* joining V_i and V_j

Answers



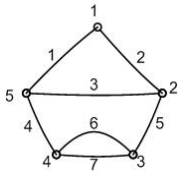
- 1. The incidence matrix is

	1	2	3	4	5	6	7
1	1	1	0	0	0	0	0
2	0	1	1	0	1	0	0
3	0	0	0	0	1	1	1
4	0	0	0	1	0	1	1
5	1	0	1	1	0	0	0

The adjacency matrix is

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	0	1
3	0	1	0	2	0
4	0	0	2	0	1
5	1	1	0	1	0

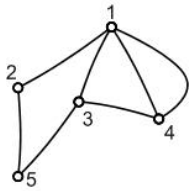
- **Self test**
- 1. Write down the adjacency and incidence matrices of the graph below.



Answers

2.

	1	2	3	4	5
1	0	1	1	2	0
2	1	0	0	0	1
3	1	0	0	1	1
4	2	0	1	0	0
5	0	1	1	0	0



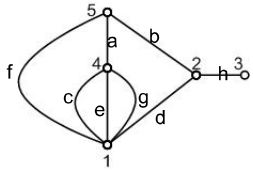
- **Self test**
- 2. Draw the graph whose adjacency matrix is given in (a)

0	1	1	2	0
1	0	0	0	1
1	0	0	1	1
2	0	1	0	0
0	1	1	0	0

(a)

	a	b	c	d	e	f	g	h
1	0	0	1	1	1	1	1	0
2	0	1	0	1	0	0	0	1
3	0	0	0	0	0	0	0	1
4	1	0	1	0	1	0	1	0
5	1	1	0	0	0	1	0	0

3.



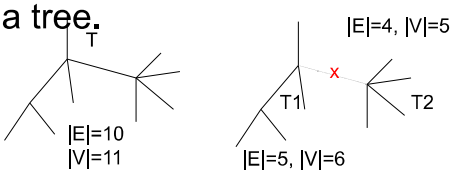
- **Self test**
- 3. Draw the graph whose incidence matrix is given in (b)

0	0	1	1	1	1	1	0
0	1	0	1	0	0	0	1
0	0	0	0	0	0	0	1
1	0	1	0	1	0	1	0
1	1	0	0	0	1	0	0

(b)

Tree properties

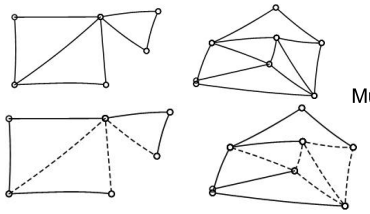
- If a tree *T* has at least two vertices then it has the following properties:
- There is exactly one path from any vertex V_i in *T* to any other vertex V_j
- The graph obtained from tree *T* by removing any edge has two components, each of which is a tree.
- $|E| = |V| - 1$



Introduction to Graph Theory
Lecture 26

Spanning trees

- A **spanning tree** of a graph *G* is
  - a tree *T*
  - a spanning subgraph of *G*.
  - That is, *T* has the same vertex set as *G*.
- Example 2 Identify a spanning tree for each of the following graphs:



Hint: remove any edge which forms a cycle

Multiple spanning trees may exist

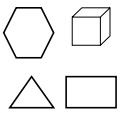
Menu

- Trees and forests
- Spanning trees
- Minimum spanning tree
- Greedy algorithm for determining a minimum spanning tree
- Shortest path problem

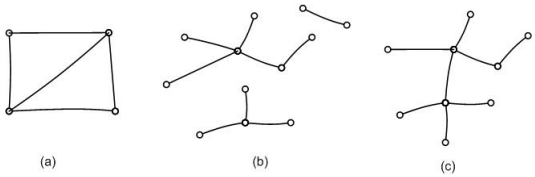
Given a graph G: How to draw a spanning tree?

- Take any vertex of *G* as an initial partial tree.
- Add edges one by one so each new edge joins a new vertex to the partial tree.
- When to stop?
- If there are *n* vertices in the graph *G* then the spanning tree will have *n* vertices and *n*-1 edges.

Trees



- A **tree** is a connected graph with no cycles.
- A **forest** is a graph with no cycles and it may or may not be connected
- Example 1: Identify which of the following graphs are trees or forests.



- Solution: (b) A forest (c) A tree

The gr$$dy algorithm for the minimum spanning tree

- Choose any start vertex to form the initial partial tree (*Vi*)
- Add the **chapst** edge, *Ei*, to a new vertex to form a new partial tree
- Repeat step 2 until all vertices have been included in the tree

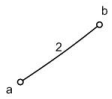
- Why is it *greedy*?

Minimum spanning tree

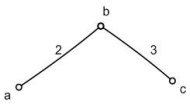
- Suppose we have a group of offices which need to be connected by a network of communication lines.
- The offices may communicate with each other directly or through another office.
- Condition: there exists one path between any two vertices.
-
- In order to decide on which offices to build links between we firstly work out the cost of all possible connections.
- This will give us a weighted complete graph as shown next.
- The **minimum spanning tree** is then the spanning tree that has the minimum cost among all spanning trees.

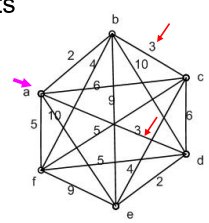
Find the minimum spanning tree for the graph representing communication links between offices as shown below.

- Start with any vertex, in this case choose the one marked *a*.
- Add the edge *ab* which is the cheapest edge of those adjacent to *a*.



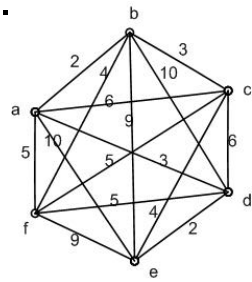
- Looking for the cheapest edge from among those incident to a or b, we find edges *bc* and *ad*, both costing 3, and that no other available edge costs less. We can choose either *bc* or *ad*. Arbitrarily we choose *bc*.





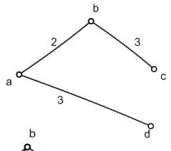
Minimum spanning tree

- A weighted complete graph.
- The vertices represent offices and the edges possible communication links.
- The weights on the edges represent the cost of construction of the link.

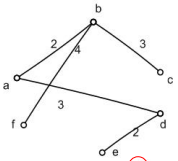


Find the minimum spanning tree for the graph representing communication links between offices as shown above.

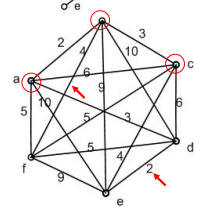
- We now look for the edge which is the cheapest remaining edge or those incident to *a* or *b* or *c* which forms a partial tree. This edge is *ad*.



- Continuing in this manner we find the minimum spanning tree shown:

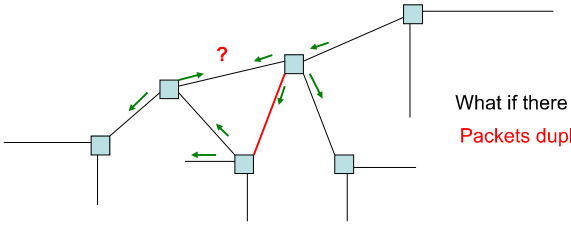


- The total cost of our solution is found to be 2+3+3+2+4=14.



What is the use of minimum spanning tree?

- **City/town planning:** design a minimum-cost road layout connecting several cities
- Used in **communications**: Ethernet **bridge** layout **autoconfiguration** – avoid packets being sent over a network segment twice, so use of minimum spanning tree is required (no cycle)



The shortest path problem

- The weights on a graph may represent *delays* in a communication network or *travel times* along roads.
- A practical problem to consider is to find the **shortest path** between any two vertices.
- **Shortest path → shortest delay**
- The algorithm to determine this will be demonstrated through an example.

Summary

- Definitions of **trees, forests** & **spanning trees**
- Shown **how to draw a spanning tree**
- Introduced the concept of a **minimum spanning tree**
- Presented the **gr$$dy algorithm** for determining a minimum spanning tree: shortest edge first
- Introduced **the shortest path problem**: to find the shortest path between any two vertices in a weighted graph

Readings

- [Mar07] Read 9.5
- [Mar13] Read 9.5

Solution - Stage 1:

- Begin at the start vertex *s*. This is the reference vertex for stage 1.
- Label all the adjacent vertices with the lengths of the paths using only one edge.
- Mark all other vertices with a very large number (larger than the sum of all the weights in the graph). In this case we choose 100. This is shown in the diagram.
- At the same time, start to form a table as shown in Table 1.

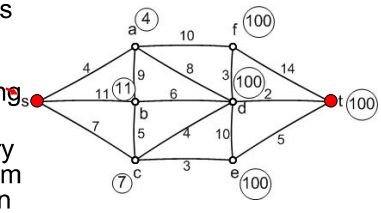


Table 1

	a	*b*	*c*	*d*	*e*	*f*	*t*
s	4	11	7	100	100	100	100

- The lengths of paths using only 1 edge from *s*

Introduction to Graph Theory
Lecture 27

Solution - Stage 2:

- Choose as the reference vertex for stage 2 the vertex with the ***smallest label*** that has not already been a reference vertex. This is vertex ***a***.
- Consider any vertex adjacent to the new reference vertex and mark it with the length of the path from *s* via *a* to this vertex if this is less than the current label on the vertex. This gives the labels shown right.
- We also add a new line to Table 1 to give Table 2, noting that as vertex *a* has been made a reference vertex the label of *s* becomes permanent and is marked with an underline in the table.

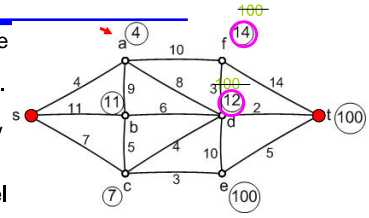


Table 2

	a	*b*	*c*	*d*	*e*	*f*	*t*
s	4	11	7	100	100	100	100
a		11	7	12	100	14	100

- The lengths of paths using up to 2 edges from *s*

Menu

- Shortest path algorithm to determine the shortest path between two vertices of a weighted graph

Solution - Stage 3:

- Choose as the reference vertex the vertex with the ***smallest label*** that has not already been a reference vertex. From table 2 we see that **c** is the reference vertex for stage 3.
- Consider any vertex adjacent to *c* that does not have a permanent label and calculate the length of the path from *s* via *c* to this vertex. If it is less than the current label on the vertex mark the vertex with this length. This gives us the labels shown right.
- We also add a new line to Table 2 to give Table 3. Note that the third line of Table 3 does not have an entry for *a* as this has already been a reference vertex.

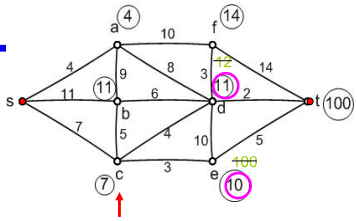


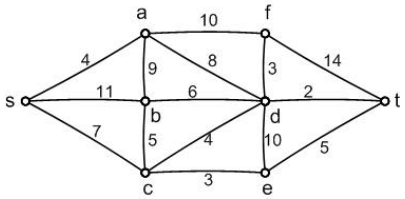
Table 3

	a	*b*	*c*	*d*	*e*	*f*	*t*
s	4	11	7	100	100	100	100
a		11	7	12	100	14	100
c		11		11	10	14	100

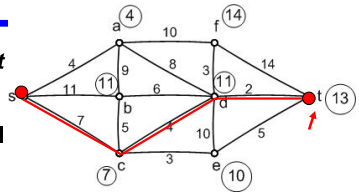
- The lengths of paths using up to 3 edges from *s*

Example 1

- The weighted graph shown below represents a communication network with weights indicating the delays associated with each edge.
- Find the minimum delay path from *s* to *t*.

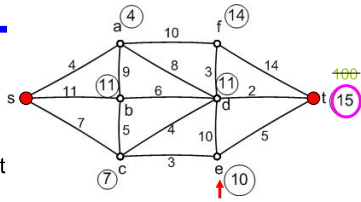


Solution - Stage 7:



- The remaining vertex with the ***smallest label*** is ***t***.
- We therefore give *t* the permanent label of 13.
- As soon as *t* receives a permanent label the algorithm stops as this label is the length of the shortest path from *s* to *t*.
- To find the actual path with this length we ***move backwards*** from *t* looking for *consistent* labels.
- This gives *t d c s*. That is, the path is *s c d t*.

Solution - Stage 4:



- Proceeding as before, the reference vertex for stage 4 is, by inspection of the third line of Table 3, vertex ***e***.
- Again we calculate the lengths of the paths from *s* via *e* to any vertices adjacent to *e* that do not have permanent labels and replace the labels on those vertices with the relevant path lengths if this is less than the existing label.
- This gives the labels shown right and Table 4.

Table 4

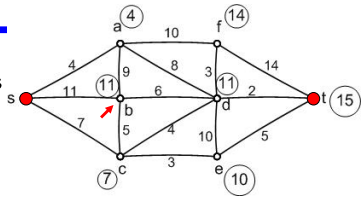
	a	*b*	*c*	*d*	*e*	*f*	*t*
s	4	11	7	100	100	100	100
a		11	7	12	100	14	100
c		11		11	**10**	14	100
e		**11**		**11**		**14**	**15**

- The lengths of paths using up to 4 edges from *s*

Dijkstra's Shortest Path Algorithm (SPA)

- Let the node at which we are starting be called the **initial node**. Let the **distance of node *Y*** be the distance from the **initial node** to *Y*. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.
- Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
- Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
- For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 9, and the edge connecting it with a neighbor *B* has length 4, then the distance to *B*(through *A*) will be 9 + 4 = 13. If B was previously marked with a distance greater than 13 then change it to 13. Otherwise, keep the current value.
- When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
- If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
- Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

Solution - Stage 5:



- Choose ***b*** as the new reference vertex (we could have chosen d instead but this would make no difference to the final result).
- Compare paths from *s* via *b* to the labels on any adjacent vertices with temporary labels and re-label if the paths are found to be shorter.
- The result of stage 5 is that the labels remain as in stage 4, but that the label on b becomes permanent giving Table 5.

Table 5

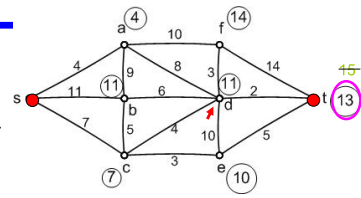
	a	*b*	*c*	*d*	*e*	*f*	*t*
s	4	11	7	100	100	100	100
a		11	7	12	100	14	100
c		11		11	10	14	100
e		**11**		11		14	15
b				**11**		**14**	**15**

- The lengths of paths using up to 5 edges from *s*

Why is SPA optimal?

- Why SPA gives us the shortest path?
- What is the complexity of SPA?
- Can SPA be generalized for related shortest path problems?

Solution - Stage 6:



- Choose ***d*** as the new reference vertex.
- The only vertices left without permanent labels are now *f* and *t*.
- The path from *s* via *d* to *t* gives *a* smaller value than the current label of 15. Hence we change the label to 11+2=13.
- The new labels are shown right together with Table 6.

Table 6

	a	*b*	*c*	*d*	*e*	*f*	*t*
s	4	11	7	100	100	100	100
a		11	7	12	100	14	100
c		11		11	10	14	100
e		11		11		14	15
b				**11**		14	15
d						**14**	**13**

- The lengths of paths using up to 6 edges from *s*

Summary

- Demonstrated the algorithm to determine the shortest path between two vertices of a weighted graph

Readings

- [Mar07] Read 9.3
- [Mar13] Read 9.3