COMP114 Experimental Methods in Computing

Selection and Use of

Random Data in Experiments

Review – why use random data?

- The example in Assessment 2 considered data supplied by a *user* population as the basis for comparing two systems.
- The actual users were chosen at random: the *data were not*.
- In this case the "typical" behaviour of the systems being compared was modelled in terms of the "typical" experience of the *systems' users*.

Review – why use random data?

- There are a number of measures that could be used quantitatively to assess how good a solution is – for example,

“*worst-case*” run-time with input “size n”

“*typical*” run-time with input “size n”

“*worst*” solution found with input “size n”

“*best*” solution found with input “size n”

“*typical*” solution found with input “size n”

Example setting

- A parcel delivery service has a number of depots (20 say) and only 1 van to move items between these. The delivery service is keen to reduce the total amount of travel involved (petrol costs, hours worked by drivers, etc.). The schedule of deliveries needed between each depot will differ each day.
- This “scheduling problem” is unlikely to have a “fast program” solution.

Example – modelling the problem

- For each pair of locations – D(i) and D(j) – moving items (with no stop) between D(i) and D(j) will incur some cost – C(i,j) – which depends on the cost of travelling between the two locations (this cost is not fixed, since it varies with the number of items to be moved).
- The main problem is to visit every depot involved with items to deliver so as to reduce the total cost involved.
- A software company offers a program, which it claims will assist in scheduling deliveries.

Criteria for Evaluating Solutions

- "worst-case" run-time = the *maximum* amount of time taken to *produce* a schedule.
- "typical" run-time = the "*average*" time taken to *produce* a schedule.
- "best" solution = the *lowest cost* schedule found by the program.
- "typical" solution = the "*average*" *cost* of schedules found.
- "worst" solution = the *highest cost* schedule found by the program.

Measuring “typical” behaviour.

- In this example we have the following:

a. A *fixed* number of depots (but this may change over time, e.g. if more are added).

b. *Varying* day-to-day costs of moving from one depot to another – C(i,j).

- Two measures of “typical” performance – how *long* it takes to *produce a schedule* and how “*cost effective*” such a schedule is.
- Randomly generated data provides one approach to estimating these measures.

How would random data be used?

- The experimental assessment could be constructed as follows:

1. Fix the number of trials (100 say).
2. For each D(i) and D(j) choose a random numerical value for C(i,j).
3. Run the scheduling program with the resulting data, noting the time taken to find a schedule and its cost.
4. Compute the overall average time and cost.

Some complications

- How do we choose a "random" value for each cost? Typically we will only have methods for making a random choice from a *range of values*?
- The model assumes it is *always* possible to travel *directly* between any two depots. What if this is *not* the case? e.g. if the road from D(1) to D(2) is closed and so all traffic between these must pass through some alternative location.
- The evaluation considers only a fixed (20) number of depots. What if we wish to review arbitrarily large number, e.g. 50, 100, 1000+?

Random "Structures"

- The issues raised on the previous slide can be treated in terms of using *random objects* with a particular *structure* as the basis for an experiment.
- We can think of structures as having a particular *size* – e.g. the number of depots in the parcel delivery case – and as meeting given criteria.
- An example of such structures has already been used in Assessment 1 together with random methods for constructing representatives, i.e. permutations of the n numbers <1,2,3, …, n>.
- In this case: *size*=n; "*structure*" = "*an ordering,* P*, of* <1,2,3,…,n> *in which each number,* k*, between* 1 *and* n *appears exactly once*".

More Examples of Structures I

- This section of the module has 2 aims:

a. To introduce some examples of frequently used structures in computing applications.

b. To describe some basic approaches that can be used with each structure type, T say, in order to generate a "*random structure* S *of type* T *whose size is* n".

- We have already looked at the example of T=Permutations and ways of generating random permutations of size n.

More Examples of Structures II

- We consider the following –

1. n-bit *numbers*.
2. *Combinations* (selections) of a given number (k, say) of objects from a collection of objects.
3. *Networks* (also called *graphs*).
4. *Binary trees*.

- It is assumed that we *already have available* methods such as those provided in, e.g. the class Random from java.util

Informal examples and applications I

- n-bit numbers: in many cases instead of methods for producing arbitrary random values there may only be "reliable" methods for generating "random bits", e.g. if a coin lands Heads=1; Tails=0.
- How can "random bit" methods be used to construct

a. Random numbers from given range?

b. What about random floats or doubles?

Informal examples and applications II

- Combinations: in lotteries such as the UK National Lottery, we are not so much interested in the *ordering* of outcomes but in the particular choice of 6 from 49 numbers selected.
- In simulating such processes techniques for choosing k values from n possibilities are used.

Informal examples and applications III

- Networks
- A huge range of problems – such as the scheduling example described earlier – can be modelled in terms of networks.
- In assessing how good such methods are as solutions, techniques for building random networks are used.
- Network:

  Collection of *nodes* {v(1), v(2), … , v(n)}

  Collection of *links* (edges) <v(i), v(j)> (links in networks may be directed or undirected)

Informal examples and applications IV

- Binary Trees.
- A number of applications can be modelled in terms of looking at properties of a logical expression that is built from the problem data, e.g. developing workable timetables.
- Given a set of *variables* {v1,v2, … , vk} and a set of *operations*, e.g. {+, – , ×, ÷} or {Ù,Ú,Ø} – a *random expression* over the variables and operations can be formed by building a special type of network called a (*binary*) *tree*.

Arbitrary Numbers from random bits

- Recall from COMP103 that numerical values are represented as a sequence of *binary* “*digits*” (or *bits*).
- So, if we only have a random bit source to use, we can build a generator of random integers between 0 and $2^k - 1$ simply by using the bit source for each of the k bits in turn.
- Note that it is *not necessary* to cast an array of k Booleans to an integer in order to do this.

Arbitrary Numbers from random bits

```
int x = 0;          // x will hold the random int.
for (int i=0; i<k; i++) {
   b = Random bit;      // b=1 or b=0
   x = 2*x + b; };
return x;
```

What about floats/doubles?

- The line “x=2*x + b” can be adapted to generate a k-bit (for suitable k) float or double value that is at least 0 and *less than* 1.

```
double x = 0.0;         // x is the random double.
double y;
for (int i=0; i<k; i++) {
  b = Random bit;      // b=1 or b=0
  if (b==1) y=0.5 else y=0.0;
  x = x/2.0 + y; };
return x;
```

Combinations – choosing k from n items

- The method is very similar to the more efficient permutation generator.
- Instead of using “m = the number of values still to be chosen – m = n, initially”, for selecting only k items the starting value of m is set to be k.

Choosing k random items from n

```
1. int[ ] Q = new int [k]     // holds the k choices
a. int m = k;
b. int t = n-1;
c. while (m>0) {
d. r =  a random integer  between 0 and t;
e. Q[m-1] = P[r];   // P is collection of n items.
f. Swap P[r] with P[t];   t-- ; m--  ;  };
g. return Q;
```

Graphs and Networks

- A *network*, H, is a structure defined by two components:

  V = {v(0), v(2), … , v(n-1)}

  F = { e(0), e(2), … e(m-1)}
- V is called the set of *nodes* in H.
- F is the set of *links*.
- Each link e(k) is specified by a pair <v(i),v(j)> of nodes from V.
- A link <v(i),v(j)> is *not the same* as a link <v(j),v(i)> .

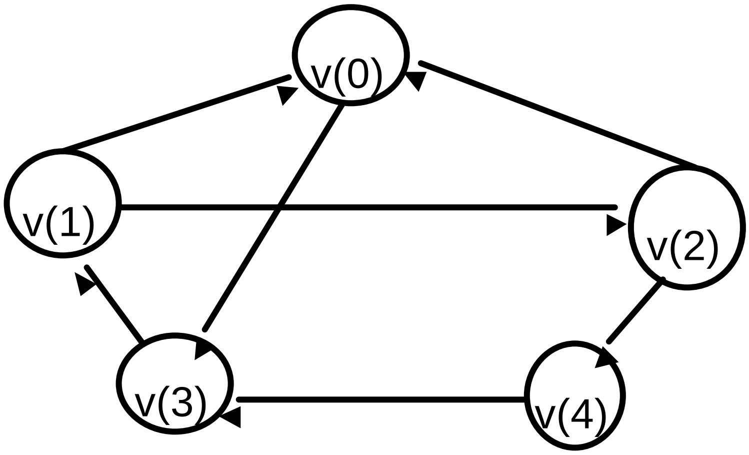
Graphs and Networks II

- A *graph*, G, is a similar structure defined by two components:

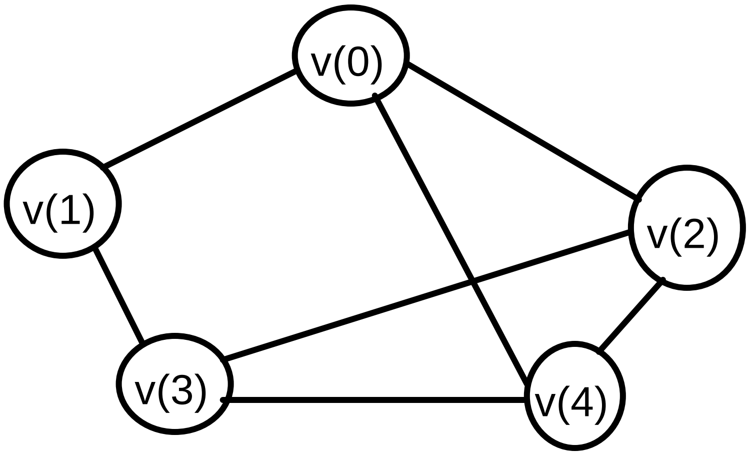
  V = {v(0), v(2), … , v(n-1)}

  E = { e(0), e(2), … e(m-1)}
- V is again called the set of *nodes* in G.
- E is now called the set of *edges*.
- Each edge e(k) is also specified by a pair {v(i),v(j)} of nodes from V.
- For graphs, however, an edge {v(i),v(j)} is *exactly the same* as an edge {v(j),v(i)}.

A network with 5 nodes; 7 links;



A graph with 5 nodes; 7 edges;



Uses of graphs and networks in C.S.

- Map data – e.g. nodes correspond to towns; links/edges to rail connections.
- Timetable constraints – nodes model lecture requirements; edges indicate when two lectures *cannot* be scheduled at the same time.
- Program structure – nodes = methods and statement blocks; links = interaction and sequencing of these.
- Many other applications are possible.

Why use "random networks"?

a. It may be difficult to determine *exact behaviour* by *analytic* techniques.
b. Even when performance guarantees *can* be given, it is often the case that "typical" performance is *much better*, e.g. one may be able to show an approach always finds a solution whose value is "at least" 1/3rd of the best possible; in practice, the *same method* may often find solutions which are ½ the optimal value.

Random networks – possible problems

a) Typical behaviour as determined by "genuinely random" networks – e.g. if one considers techniques in which every n node network is equally likely – may be very different from "typical instances" seen in real contexts.
b) For complex structures such as graphs, networks, trees, construction of random data and interpretation of results requires some care to be taken.

Java Representation

- The simplest (and very widely used) approach is via *2-dimensional arrays*.
- If there are no weights associated with links/edges then a boolean 2-d array can be used otherwise an int 2-d array is adopted (with a suitable convention for the weight assigned for links/edges that are not present).

Network *class* (*unweighted links*) *I*

- Fields

```
private boolean H[ ][ ];  // The network
private int n;   // Number of nodes
private int m;  // Number of links
```

Network *class* (*unweighted links*) *II*

- Instance Methods

```
public void AddLink(int i,j);
// Adds a link from node i to node j.
public void RemoveLink(int i,j);
// Removes the link from node i to node j.
public boolean TestLink(int i,j);
// Returns true if there is a link from node i to
node j. Otherwise returns false
```

Network *class* (*unweighted links*) *III*

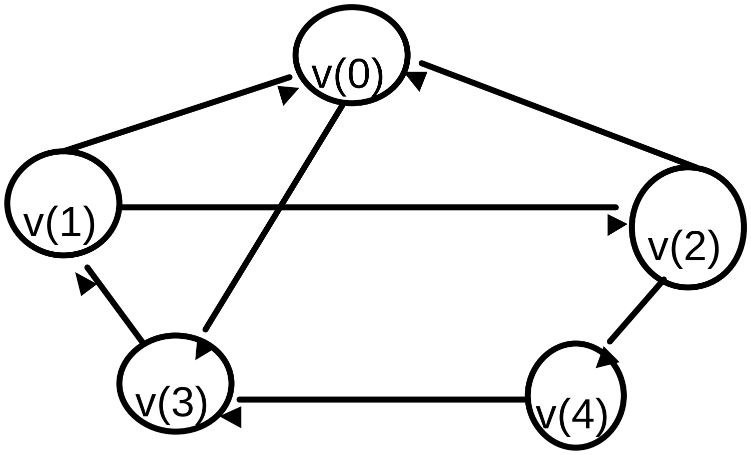
- Constructor

```
public void Network(int Nodes)
    {
    H = new boolean [Nodes][Nodes]
    n = Nodes; m = 0;
   for (int i=0; i<n; i++)
      for (int j=0; j<n; j++)
         H[ i ][ j ] = false;
    };
```

Example method realisation

```
public void AddLink(int i,j)
    {
    if (!H[ i ][ j ])      // Link i to j not present.
      m++;                // so increase link count.
    H[ i ][ j ] = true;  // Adds link to H
    };
```

A network with 5 nodes; 7 links;



Example network

H	0	1	2	3	4
0	False	False	False	True	False
1	True	False	True	False	False
2	True	False	False	False	True
3	False	True	False	False	False
4	False	False	False	True	False

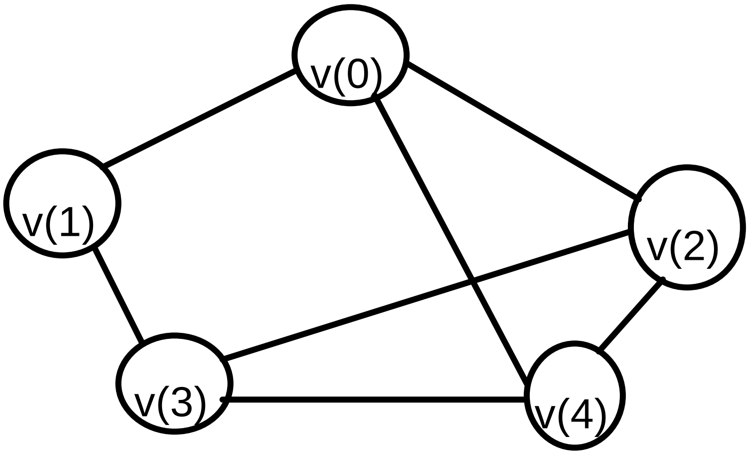
Representing Graphs

- A Graph class can be defined in a very similar way to that used for Network.
- We still have the fields n and m as before, and use boolean[ ][ ] G as the structure containing the graph.
- The main changes are to the methods AddLink and RemoveLink for which we use the names AddEdge and RemoveEdge.
- Similarly, TestLink is now called TestEdge

Adding an edge to G

```
public void  AddEdge(int i,j)
    {
    if (!G[ i ][ j ])       // Edge {i,j} not present.
       m++;                // so increase edge count.
    G[ i ][ j ] = true;  // Adds edge to G
    G[ j ][ i ] = true;  // but also need this in G
    };
```

A graph with 5 nodes; 7 edges;



Example graph

G[i][j]=G[j][i]

G	0	1	2	3	4
0	~~False~~	True	True	False	True
1	True	~~False~~	False	True	False
2	True	False	~~False~~	True	True
3	False	True	True	~~False~~	True
4	True	False	True	True	~~False~~

Random Networks and Graphs

- The basic methods are very similar.
- A “naïve” approach uses the following:

  “In a typical (i.e random) network, a link from i to j has *exactly the same* chance of being present as it has of being absent.”

$\therefore$ For each possible link H[i][j] choose a random value, x, between 0 and 1: if x<0.5 set H[i][j]=true; else set H[i][j]=false

Random Network Method I

```
public Network RandomNetwork(int n) {
  Random S = new Random();
  Network H = new Network(n);
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      if ((S.nextDouble()<0.5) && (i != j))
       H.AddLink(i,j);
  return H;   };
```

Random Graph Method I

```
public Graph RandomGraph(int n) {
  Random S = new Random();
  Graph G = new Graph(n);
  for (int i=0; i<n; i++)
    for (int j= i+1; j<n; j++)
      if (S.nextDouble()<0.5)
        G.AddEdge(i,j);
  return G;  };
```

RandomNetwork *and* RandomGraph

- The main distinction is the form of the inner for loop: this is

  (j=i+1; j<n; j++)

  in the RandomGraph method but

  (j=0; j<n; j++)

  in RandomNetwork.
- This ensures each edge {i,j} is looked at exactly once by RandomGraph.

Properties

- Let N=n*n-n (for networks) and (n*n-n)/2 (for graphs).
- The methods RandomGraph and RandomNetwork have

  $$2 \times 2 \times 2 \ \ldots \times 2 \times 2$$

  N times

  ways of producing a network or graph.
- Each possibility is *equally likely*.

Problems with these methods

- A “typical” structure output by these methods will have roughly N/2 links (edges).
- Networks and graphs arising in applications, however, are rarely this “dense” – around $N^{0.5}$ is more likely. For example –
- Consider the network associated with all direct flights between airports in which a particular airline operates, e.g. EasyJet use ca. 90-95 locations. Do EasyJet offer 4000+ direct flights (that is 45+ from each airport)? Only 20 destinations are available from Liverpool (one of the main bases).

Possible Solutions I

A. Fix the number of links (edges) in advance and develop a method that produces only n-node, m-link (edge) random networks (graphs).
B. Allow the probability that a link (edge) appears to be specified as a parameter (instead of assuming it is always ½).

Possible Solutions II

- Solution A could be realised as a “special case” of generating a random combination of m items from N possible.
- While this is a reasonable approach, if solutions with “*roughly*” m links (rather than *exactly* m) are acceptable, there is an easy way of achieving this by adopting Solution B.

Random Network Method II

```
public Network RandNet(int n , double p) {
  Random S = new Random();
  Network H = new Network(n);
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      if ((S.nextDouble()<p) && (i != j))
        H.AddLink(i,j);
  return H;   };
```

Random Graph Method II

```
public Graph RandGraph(int n, double p) {
  Random S = new Random();
  Graph G = new Graph(n);
  for (int i=0; i<n; i++)
    for (int j= i+1; j<n; j++)
      if (S.nextDouble()<p)
        G.AddEdge(i,j);
  return G;  };
```

RandNet *and* RandGraph *Properties*

- For these methods a link (edge) is included with probability p.
- This means that if N is the maximum possible number of links (edges) in an n-node network (graph) –
- a typical network produced by RandNet(n,p) will have about $m=N\times p$ links.
- a typical graph produced by RandGraph(n,p) will have about $m=N\times p$ edges.

What value of p *should be used?*

- If n-node networks (graphs) with roughly m links (edges) are being considered, then a value of p = n/N will have the effect required – recall that N=(n*n-n) for *networks*; N=(n*n-n)/2 for *graphs*.
- For experimental evaluations where "sparse" networks arise in practice, ranges of p as multiples of 1/n or log(n)/n are often used:

2/n, 3/n, …, k/n

2×log(n)/n, 3×log(n)/n, …, k×log(n)/n

Binary Trees I

- Although binary can be treated as a special type of network, it is more useful to represent their structure using techniques which reflect how binary trees are defined.
- A binary tree contains *two distinct types* of node – *leaf* nodes and *internal* nodes.
- An n-leaf binary tree has *exactly* n–1 internal nodes: the total number of nodes (leaves+internal) is *always* an *odd number*.

Binary Trees II

- n-leaf binary trees are defined *recursively*, that is in terms of *smaller* (number of leaf nodes) binary trees:

A. A *single* node is a 1-leaf binary tree.

B. An n-leaf binary tree, T, has *three separate parts*:

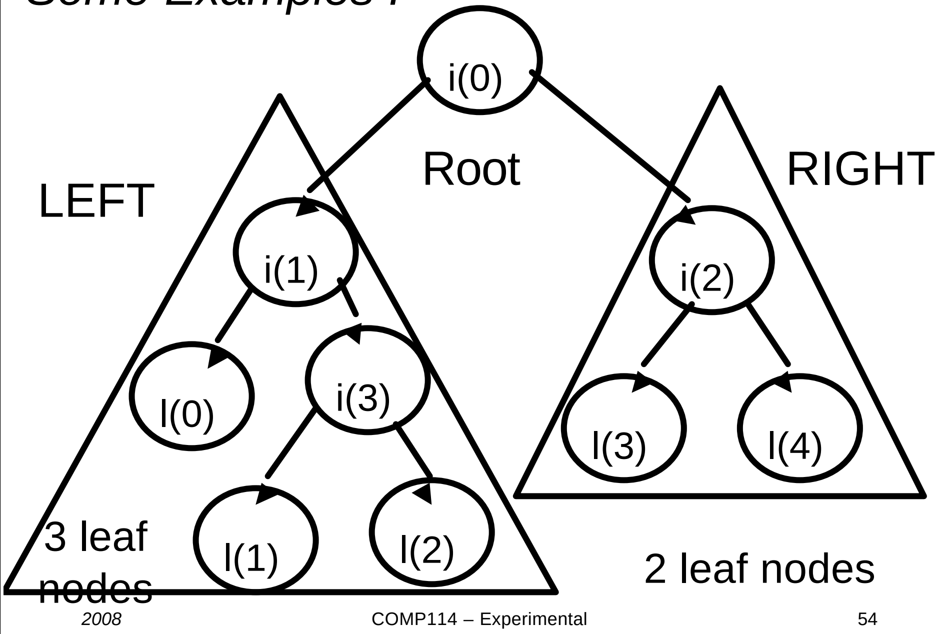
An internal node called the tree *root* – r(T)

A *Left* binary tree with k leaf nodes – L

A *Right* binary tree with n-k leaf nodes – R

- There are links from r(T) to r(L) and r(T) to r(R).
- The number of leaf nodes in L (and in R) is *at least* 1 and *at most* n–1

Some Examples I



Some applications in Computing

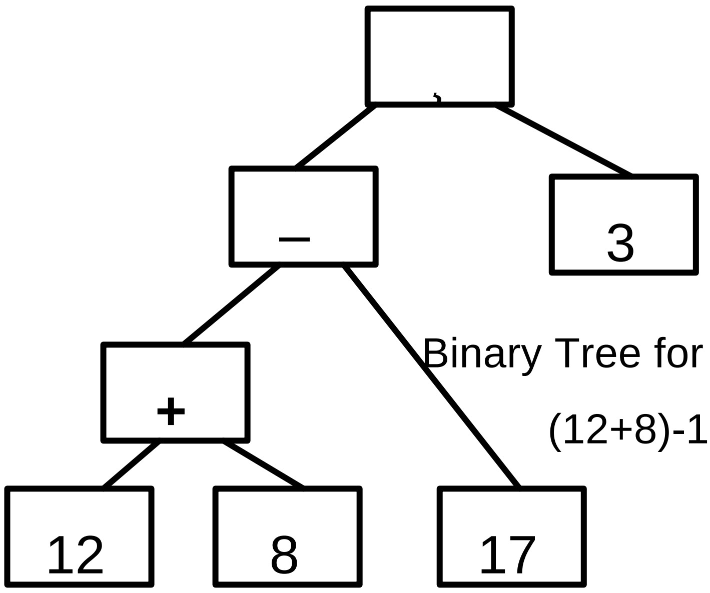
a. Representing arithmetic and logic expressions.

b. Maintaining ordered collections of information, e.g. Telephone directories and other "look-up" tables.

- In (b) a tree node is structured as

Left Tree "***pointer***"	***Look-up key***	***Data for key***	Right tree "***pointer***"

Node "***content***"

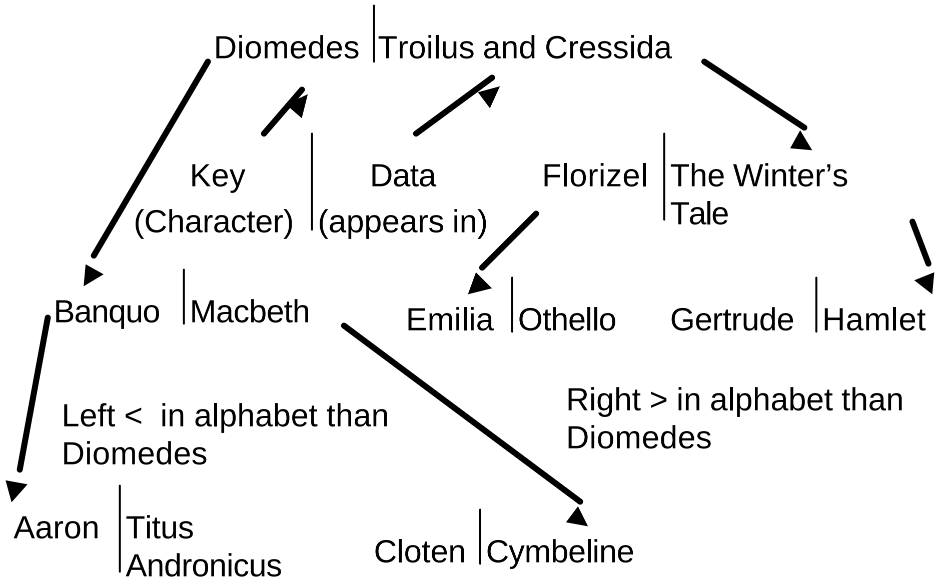


Binary Tree for expression

(12+8)-17)÷3

Summary of binary tree use

- For data having “Key-Value” structure, where “Keys” can be *ordered*.
- Store “key-value” pairs in a binary tree.
- The “*middle*” key in the ordering is stored in the tree root.
- All keys *before* this are in the Left Tree.
- All keys *after* it are in the Right tree.


Diomedes | Troilus and Cressida
Key (Character) | Data (appears in)
Florizel | The Winter’s Tale
Banquo | Macbeth
Emilia | Othello
Gertrude | Hamlet
Left < in alphabet than Diomedes
Right > in alphabet than Diomedes
Aaron | Titus Andronicus
Cloten | Cymbeline

A Binary Tree Class in Java I

Problem: How do we model the "recursive" structure of binary trees?

Solution: If the class name is, for example, BinaryTree (with the "node" content being, say, an int, then the *same class name* (BinaryTree) describes the type of the Left and Right components.

A Binary Tree Class in Java

- BinaryTree class – Fields

```
public class BinaryTree {
    protected int TreeRoot;
    protected BinaryTree LeftTree;
    protected BinaryTree RightTree;
};
```

- TreeRoot can be *any* Object (not just int).

A Binary Tree Class in Java

- BinaryTree class – Selected Methods

public boolean IsLeafNode()

Returns true if this tree has just a single node.

public void SetRootValue(int n)

Sets the data in the root of this tree to be n.

public void SetLeft(BinaryTree T)

public void SetRight(BinaryTree T)

Sets the LeftTree (RightTree) field to be T.

public static BinaryTree BuildTree(int root,
                                   BinaryTree Left, Right)

Realisations –

```
public boolean IsLeafNode() {
   return (Left==null)&&(Right==null); }

public static BinaryTree BuildTree(int root,
                                   BinaryTree Left, Right)
   BinaryTree T = new BinaryTree();
  T.SetRootValue(root);
   T.SetLeft(Left); T.SetRight(Right);
   return T; }
```

Random Binary Trees (*n leaf nodes*)

- We describe 2 approaches – both of which feature in Assessment 3 (together with a third method).
- Method 1 – RootDown

```
1. If (n>1) {
2. Choose a random integer, k, between 1 and n-1 – (each k has 1/(n-1) chance.
3. LeftTree = RootDown (k);
4. RightTree = RootDown (n-k);
```

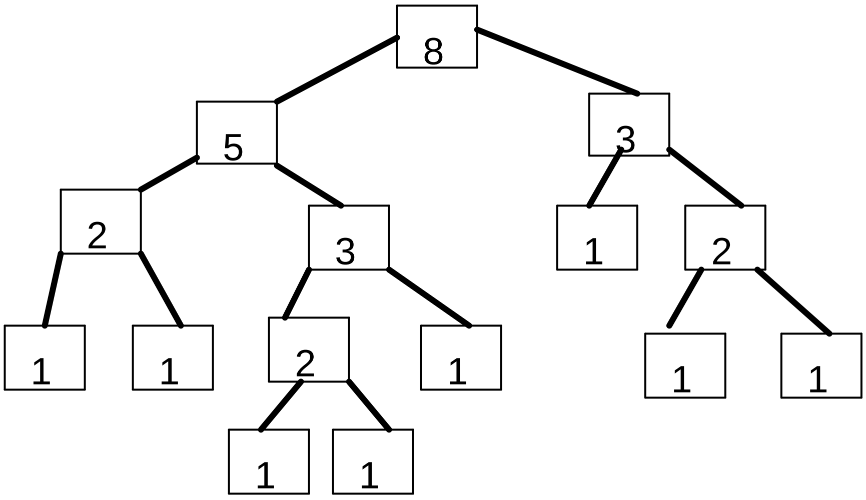
Method 2 – LeafUp

```
BinaryTree[ ] T = new BinaryTree()[n];
int m = n;
while (m>1) {
    j = Random integer between 0 and m-1
    Swap T[ j ] and T[ m-1 ];
    i = Random integer 0 and m-2;
    Temp = BuildTree(m,T[ j ], T[ i ]);
    T[ i ] = Temp; m – – ; }; return T[ 0 ];
```

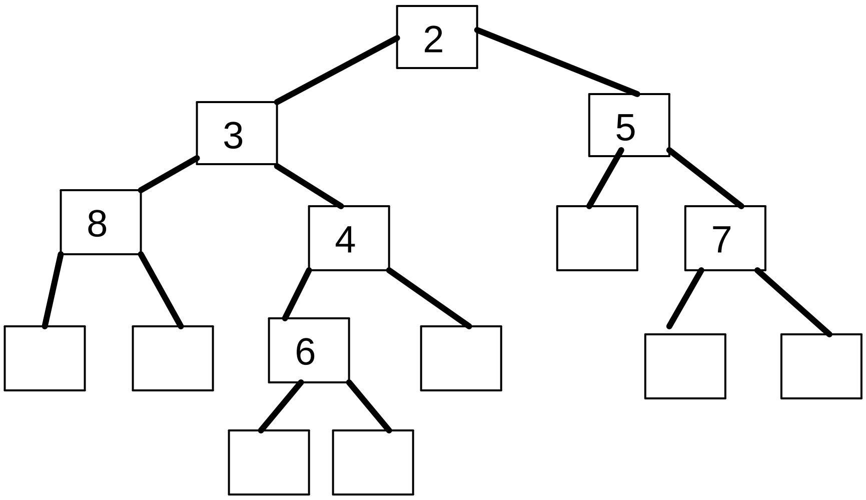
Comparison –

- RootDown *recursively* forms an n-leaf binary tree by randomly choosing the number of leaf nodes (k) in its Left tree (so that the Right tree has n-k leaf nodes).
- When k=1 the tree formed has one leaf node.
- LeafUp starts with n (1-leaf) trees; chooses two at random to give the Left and Right trees of a new tree. This will result in n-1 (n-2 single leaf + 1 2-leaf) trees.

Example – RootDown



Example – LeafUp



Properties – Tree Depth

- The *depth* of a binary tree, T, measures the number of nodes in the *longest path of links* from the root of T to any leaf node of T.
- If T is a single leaf then Depth(T)=1 else Depth(T) = 1 +

  maximum {Depth(T.Left),Depth(T.Right)

Random Trees and Depth

- In most applications where binary trees occur, these trees are "shallow": their depth is significantly smaller than the number of leaf nodes.
- For example, for <Key-Data> lookup, if a tree is very unbalanced, it may take much longer to find some of the stored keys.

Random Trees and Depth

- Both RootDown and LeafUp are *biased* to produce "*shallow*" trees.
- Neither method is "*uniform*": it is *not* the case that every n-leaf tree has an *equal chance* of being generated.
- Methods for which every n-leaf tree is equally likely are *non-trivial*.
- The characteristics of "*uniformly generated*" binary trees are *very different* from those output by RootDown and LeafUp.