1B1b

Implementing Data Structures

Lists, Hash Tables and Trees

Agenda

- Classes and abstract data types.
- Containers.
- Iteration.
- Lists
- Hash Tables
- Trees

Note – here we only deal with the *implementation* of data structures. 1b12 and 1b13 cover the detailed properties of data structures.

Data Abstraction

- We know a class declaration creates a *User Defined Type*.
- We can also describe a class as an implementation of a *data abstraction* or *data type*.
- An Abstract Data Type (ADT) provides a specification of a data type.

A Data Abstraction

```
class Pair
{
   private int x ;
   private int y ;
   ...
   public Pair(int a, int b)
   { ... }
  ...
}
```

A new data abstraction is created here.

Also a new type.

A Data Abstraction (2)

```
. . .
Pair p = new Pair(1, 3) ;
Pair q = new Pair(34, -23) ;
. . .
```

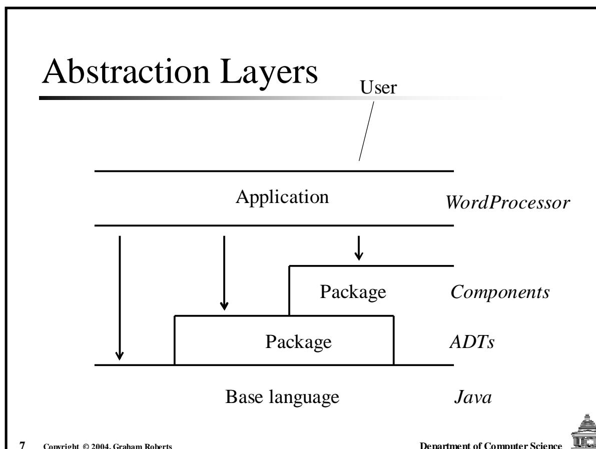
- A Pair can now be directly used, rather than having to manage two separate variables.
- Pair is (a bit) more abstract and hides unwanted detail that would otherwise intrude.

Abstract Data Types and Classes

- A class can be used to provide an implementation that *conforms* to an ADT specification.
- Typically ADTs are associated with data structures.
  - *Collections* or *Containers*.
  - Collections are objects that act as containers in which other objects (or really object references) are stored.
  - List, Tree, ArrayList, Graph, Hash Table, etc.

Copyright © 2004, Graham Roberts

Abstraction Layers



Implementing a container

- Obviously use a class…
- Need a data structure to store contained object references:
  - one or more instance variables (private of course).
- Need algorithms to implement access operations as methods.

Implementation properties

- Need to consider:
  - Memory use.
  - Speed of operation.
- Typically trading off one property against another.
- Need to select implementations that match the needs of your program.

Iterators

- Every collection has to provide a mechanism for dealing with each element in the collection in turn.
- Such an mechanism is called an *iterator*.
- Algorithms such as linear searching, comparison, function application depend on use of iterators.

ArrayList – Using an Iterator

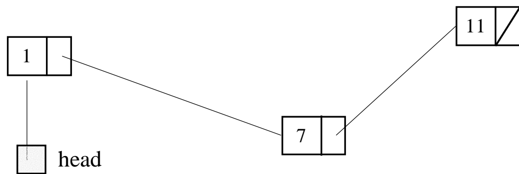
```
ArrayList a = new ArrayList () ;
...
for (Iterator i = a.iterator() ; i.hasNext() ; )
{
   doSomething((String)i.next()) ;
}
```

No longer need an integer counter, instead an Iterator object is used to access each ArrayList element.

Questions?

Linked Lists

- A linked list is implemented as a chain of linked elements (objects).



 Copyright © 2004, Graham Roberts Department of Computer Science

Linked Lists

- Each element or node consists of a stored value and a reference to the next element.
- A reference is maintained to the *head* of the list.
- An individual element is located by following the chain from the head.

 Copyright © 2004, Graham Roberts Department of Computer Science

Sequence

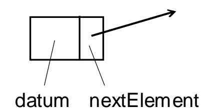
- Elements in a list (or vector, or array) are stored in *sequence*.
- Accessing elements often relies on the sequence.
- A list is a *sequence container*.

 Copyright © 2004, Graham Roberts Department of Computer Science

List Implementation – List Element

```
class ListElement
{
   Object datum ; // Note use of Object
   ListElement nextElement ;

   . . .
}
```



 Copyright © 2004, Graham Roberts Department of Computer Science

Complete ListElement

```
public class ListElement
 {
  private ListElement next;
  private Object value;

  public ListElement(Object value,
                     ListElement next)
  {
    this.value = value;
    this.next = next;
  }

  public Object getValue()
  {
    return value;
  }

  public ListElement getNext()
  {
    return next;
  }
 }
```

 Copyright © 2004, Graham Roberts Department of Computer Science

Using ListElements

- Lists can be constructed directly using ListElements.
- Useful for creating chains of objects.
- Need to implement methods to access the chain (add, remove, search, etc.).

 Copyright © 2004, Graham Roberts Department of Computer Science

Copyright © 2004, Graham Roberts

A *List* class

```
class List
{
    private static class ListElement { ... }
    private ListElement head = null ;

    public List() { ... }
    public Object getHead() { ... }
    public List getTail() { ... }
    public boolean isEmpty() { ... }
    public Iterator iterator() { ... }
    public void insertHead(Object obj) { ... }
    // etc.
}
```

Note: The element class is *nested* inside the List class and made *private*.

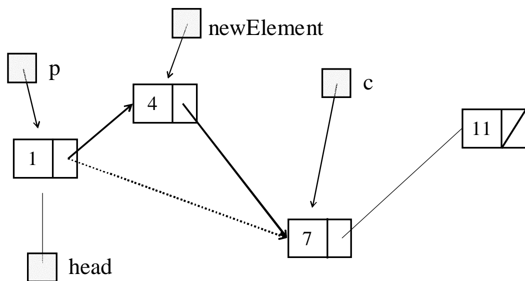
 Copyright © 2004, Graham Roberts Department of Computer Science

List Class v. List Elements

- The List class provides the public interface to lists.
  - A List object can only be used by calling the public methods declared by the List class.
- List elements are private and *cannot* be used externally.
- Implementation details are hidden (encapsulated).

 Copyright © 2004, Graham Roberts Department of Computer Science

Example algorithm – Insertion



 Copyright © 2004, Graham Roberts Department of Computer Science

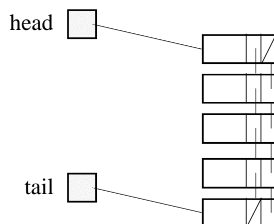
Linked List Properties

- Inserting/removing at beginning or end is fast.
- Insertion/removal in middle can be fast once the location is found.
- But there is the potential cost of linear access – O(n).
- Good for situations when elements are repeatedly inserted and deleted.

 Copyright © 2004, Graham Roberts Department of Computer Science

Double-Link List

- Links in both directions.
- Head and tail references.
- Some algorithms easier to implement but extra storage cost for each element.



 Copyright © 2004, Graham Roberts Department of Computer Science

Questions?

 Copyright © 2004, Graham Roberts Department of Computer Science

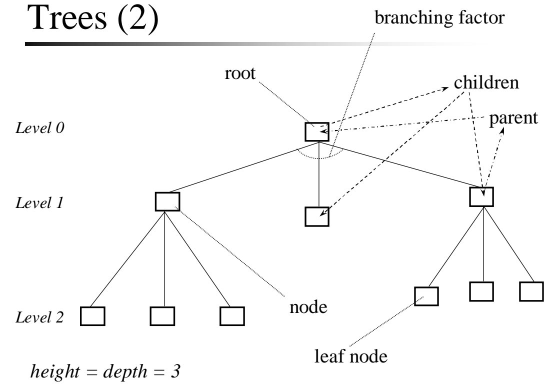
Copyright © 2004, Graham Roberts

Trees

- Trees are another variation of data structures based on linked elements.
- They use a hierarchical organisation of elements rather than straight chains.

 Copyright © 2004, Graham Roberts

Trees (2)



height = depth = 3

 Copyright © 2004, Graham Roberts

Trees (3)

- Crucial properties of Trees:
  - Links only go down from parent to child.
  - Each node has one and only one parent (except *root* which has no parent).
  - There are no links up the data structure; no child to parent links.
  - There are no sibling links; no links between nodes at the same level.

 Copyright © 2004, Graham Roberts

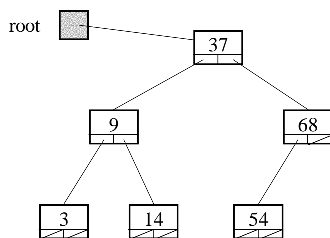
Trees (4)

- Trees are immensely useful for sorting:
  - insertion automatically sorts!
- and searching:
  - sorted structure minimises the number of comparisons.

 Copyright © 2004, Graham Roberts

Binary Trees

- The simplest kind of tree.



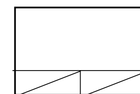
This is a complete binary tree. Each node has a maximum of 2 child nodes.

Nodes are ordered so that left child nodes have a value less than parent, right child nodes greater than or equal to.

 Copyright © 2004, Graham Roberts

Binary Trees (2)

```
// A binary tree node
class Node
{
   public Node(Object o, Node l, Node r)
      { value = o ; left = l ; right = r ; }
   Object value ;
   Node left ;
   Node right ;
}
```



 Copyright © 2004, Graham Roberts

Copyright © 2004, Graham Roberts

Binary Trees (3)

```
public class Tree
{
    private class Node { … }
    private Node root = null ;
    public Tree() { ... }
    public void insert(Object obj) { ... }
    public void delete(Object obj) { ... }
    public boolean includes(Object obj) { ... }

    // Iterator(s)
    public Iterator iterator() { ... } // But which order?
  ...
}
```

 Copyright © 2004, Graham Roberts

Binary Tree Iteration

- Four ways of iterating through a tree:
  - In-order.
  - Pre-order.
  - Post-order.
  - Level-order.

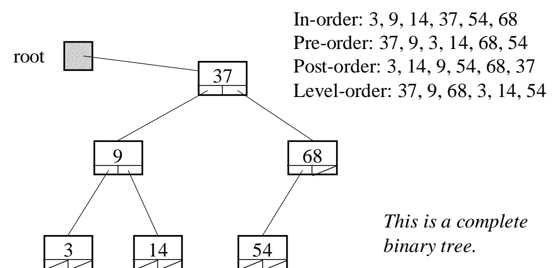
 Copyright © 2004, Graham Roberts

Binary Tree Iteration (2)

- Pre-order, post-order and in-order are related since they just rearrange order of iteration.
  - Depth-first searches.
- Level-order is different.
  - Breadth-first search.

 Copyright © 2004, Graham Roberts

Binary Tree Iteration (3)



In-order: 3, 9, 14, 37, 54, 68
Pre-order: 37, 9, 3, 14, 68, 54
Post-order: 3, 14, 9, 54, 68, 37
Level-order: 37, 9, 68, 3, 14, 54

This is a complete binary tree.

 Copyright © 2004, Graham Roberts

Binary Tree Iteration (4)

- In-order iteration:

```
void inOrder ()
{
    if (left != null) { left.inOrder(); }
    System.out.println(value);
    if (right != null) { right.inOrder(); }
}
```

 Copyright © 2004, Graham Roberts

Binary Tree Iteration (5)

- Pre-Order Iteration:

```
void preOrder ()
{
    System.out.println(value);
    if (left != null) { left.preOrder(); }
    if (right != null) { right.preOrder(); }
}
```

 Copyright © 2004, Graham Roberts

Copyright © 2004, Graham Roberts

Binary Tree Iteration (6)

- Post-Order Iteration:

```
void postOrder ()
{
    if (left != null) { left.postOrder(); }
    if (right != null) { right.postOrder(); }
    System.out.println(value);
}
```

 Copyright © 2004, Graham Roberts Department of Computer Science

Binary Tree Iteration (7)

- Level-order iteration.
- Need a queue of nodes:

```
void levelOrder()
{
   create empty queue
   add root node to queue
   while (queue is not empty)
   {
      Node n = get and remove node at front of queue
      print n.value
      add n.left to end of queue
      add n.right to end of queue
   }
}
```

 Copyright © 2004, Graham Roberts Department of Computer Science

Searching binary tree

- Use node value to determine whether to go left or right.

```
boolean search(int n)
{
   if (value == n) {return true;}
   if ((value < n) && (left != null))
      {return left.search(n);}
   if ((value >= n) && (right != null))
      {return right.search(n);}
   return false;
}
```

 Copyright © 2004, Graham Roberts Department of Computer Science

Questions?

 Copyright © 2004, Graham Roberts Department of Computer Science

Map

- In mathematics a map (aka function) relates members of one set to members of another set:

$$m : X \rightarrow Y$$

 Copyright © 2004, Graham Roberts Department of Computer Science

Arrays

- Arrays (and ArrayLists) are implementations of maps:

  array : int → Y

- For example:

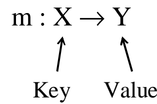
```
char array[20] ;
array[3] = ‘c’ ;
array[5] = ‘w’ ;
```

 Copyright © 2004, Graham Roberts Department of Computer Science

Copyright © 2004, Graham Roberts

Keys and values

$m : X \rightarrow Y$



- For example:
  - Key type String.
  - Value type PhoneNumber.
  - Mapping from names to phone numbers.

43 Copyright © 2004, Graham Roberts Department of Computer Science

Hash Table

- An ADT that implements a map from any class type to any class type.
  - For example:

$map : String \rightarrow Colour$

```
Colour c = (Colour)a.get("green") ;
```

- Needs a data structure to store mapping.
  - Want O(1) access.

44 Copyright © 2004, Graham Roberts Department of Computer Science

An idea?

- Could set up a mapping from the key to an int value,
- and then use the int as an *array* index.

$G : X \rightarrow \texttt{int}$

$H : \texttt{int} \rightarrow Y$

$m = H . G$

45 Copyright © 2004, Graham Roberts Department of Computer Science

Hash Function

- Use a *hash function* to map the search key into an integer that can be used as an index into the array:

```
int hash(X key) ;
```

46 Copyright © 2004, Graham Roberts Department of Computer Science

Hash Function (2)

- The hash function must:
  - return an integer within the array bounds of the storing array,
  - map keys consistently and evenly to the integers; and,
  - be quick to calculate.

47 Copyright © 2004, Graham Roberts Department of Computer Science

Hash Function example

- Consider the case where keys are strings.
- Need a mapping from the string to an integer array index.
- If we use characters as the key then:

```
int key = (key[0] + 3*key[1]) % tableSize
```

is a possible hash function.

48 Copyright © 2004, Graham Roberts Department of Computer Science

Hash Function (3)

- Hashing is *so* important that in Java *every* object has a hash code to enable easy storage in hash tables.
- See the method hashCode implemented by all objects. (Inherited from Object.)

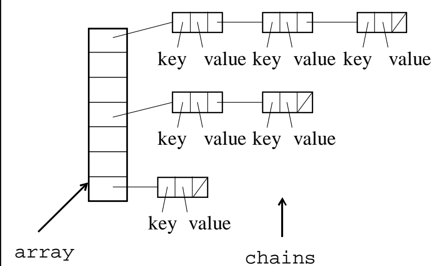
Hash Function (4)

- Given that there are more keys that array entries, there will be "multiple hits".
- The hash function will return the same integer for a number of keys.
- Need a mechanism for handling this.

Chained Hashing

- The hash table is an array of linked nodes (like linked lists).
- The first stage of search is an hash lookup.
- The second stage of search is a linear search along the linked chain of nodes.
- The chains allow for *overflow* when hash values *collide*.

Chained Hashing (2)



Chain Node Class

```
private static class Node
{
    public Node next ;
    public Object key ;
    public Object val ;
    etc.
}
```

- Like a LinkedList node, but with an extra field.
- Rest of class is a simplified list class.

Hash Table class

```
class HashTable
{
    private Node[ ] table =
        new Node[tableSize] ;
    . . .
}
```

Copyright © 2004, Graham Roberts

Chained Hashing (3)

- Values are inserted by:
  - Hashing key and performing array index.
  - Creating new node.
  - Inserting new node at head of chain.
- Look-up:
  - Hash key and perform array index.
  - Linear search of chain to find node with matching key.
  - Return value from node.
- Allows duplicate key/values pairs to exist.

 Copyright © 2004, Graham Roberts Department of Computer Science

Open Hashing (1)

- Have seen linked lists used as the overflow technique in an hash table.
- There is one other major technique for handling hash collisions: open hashing.

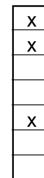
 Copyright © 2004, Graham Roberts Department of Computer Science

Open Hashing (2)

- The array holds the data itself not chains of nodes holding the data.
- If the slot determined by the hash function is full, linearly search down the array for the next empty slot.

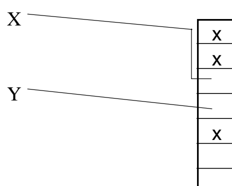
 Copyright © 2004, Graham Roberts Department of Computer Science

Open Hashing (3)



 Copyright © 2004, Graham Roberts Department of Computer Science

Open Hashing (4)



 Copyright © 2004, Graham Roberts Department of Computer Science

Open Hashing (5)

- Can do this linearly, e.g. step by 1 if there is a clash.
- Can also do this quadratically, or even exponentially.
- But number of elements that can be stored is limited by array size.

 Copyright © 2004, Graham Roberts Department of Computer Science

Copyright © 2004, Graham Roberts

Example code

- See the 1b1b web page for example code for a Linked List and a Chained Hash Table.
- Make sure you study this code and understand how it works.

61 Copyright © 2004, Graham Roberts Department of Computer Science

Questions?

62 Copyright © 2004, Graham Roberts Department of Computer Science

Further Information

- Check the 1b1b web page for example code.
- Read Part 2 of the text book.
  - Next year you need to know and understand the content of part 2 in detail.
  - There won't be lectures on the material.

63 Copyright © 2004, Graham Roberts Department of Computer Science

Summary

- Looked at the key data structures:
  - List
  - Tree
  - Hash Table (Map)
- All rely on object references (pointers).
- Have different performance properties.

64 Copyright © 2004, Graham Roberts Department of Computer Science

Copyright © 2004, Graham Roberts