DEPARTMENT OF COMPUTER SCIENCE

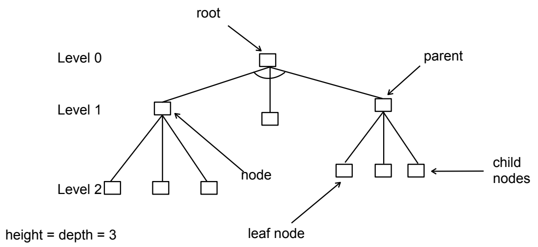
UCL

COMP1008 Implementing Data Structures Binary Trees and Hash Tables

Trees

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

Trees (2)



Trees (3)

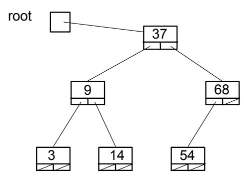
- Crucial properties of Trees:
  - Links only go down from parent to child.
  - Each node has one and only one parent (except root which has no parent).
  - There are no links up the data structure; no child to parent links.
  - There are no sibling links; no links between nodes at the same level.

Trees (4)

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

Ordered Binary Trees

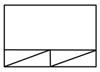
- The simplest kind of tree.



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 parent.

Ordered Binary Trees (2)



```
// A binary tree node
private static class TreeNode
{
   public Node(Comparable o, TreeNode l, TreeNode r)
      { value = o ; left = l ; right = r ; }
   Comparable value ;
   TreeNode left;
   TreeNode right;
   // etc...
}
```

Anything put in a binary tree must be Comparable.

Not a generic class but doesn't need to be as value stored in node must be Comparable.

Ordered Binary Trees (3)

```
public class BinaryTree
{
      private class TreeNode { … }
      private TreeNode root = null ;
      public BinaryTree() { ... }
      public void insert(Comparable obj) { ... }
      public void delete(Comparable obj) { ... }
      public boolean includes(Comparable obj) { ... }

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


Binary Tree Iteration

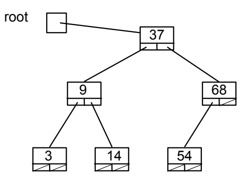
- Four ways of iterating through a tree:
  - In-order.
  - Pre-order.
  - Post-order.
  - Level-order.

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.

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



Binary Tree Iteration (4)

In-order iteration:

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

Binary Tree Iteration (5)

Pre-Order Iteration:

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


Binary Tree Iteration (6)

Post-Order Iteration:

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


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
  }
}
```


Binary Tree Iteration (8)

- Actually need a family of iterator classes and iterator() methods in class BinaryTree.
- But all iterator classes can implement interface Iterator.
- Once specific iterator is selected, client code doesn’t need to now which kind it is.
  - Programming to an interface.

Searching Ordered 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;
}
```


More Trees

- Only looked at basic binary trees,
- But there are many more kinds
  - AVL trees
  - Balanced trees
  - etc.
- See text book.

Questions?

Map

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

$$m : X \rightarrow Y$$

Arrays

- Arrays (and ArrayLists) are implementations of maps:

  array : int → Y

- For example:

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


Generalise: Keys and values

$$m : X \rightarrow Y$$

↑ Key ↑ Value

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

Hash Table

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

    map : String → Colour

    Colour c = (Colour)a.get("green") ;
- Need a data structure to store mapping.
  - Want O(1) access.

Mapping

- Want to implement a generalised mapping, so:
  - Set up a mapping from the key to an int value,
  - and then use the int as an array index.

$$G : X \rightarrow int$$
$$H : int \rightarrow Y$$

$$m = H . G$$

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);
- The hash function must:
  - return an integer within the array bounds of the storing array.
  - map keys consistently and evenly to the integers.
    - Don't want too many keys mapping to same integer.
  - be quick to calculate.
- Hard to write a good hashing function.

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.

Hash Function (3)

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

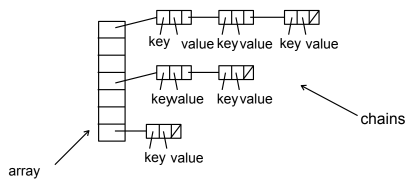
Hash Function (4)

- Given that there are more keys that array entries, there will be "multiple hits" or collisions.
  - 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 to use hash function to access array element.
- The second stage of search is a linear search along the linked chain of nodes at array element.
- The chains allow for overflow when hash values collide.

Chained Hashing (2)



A good hash function keeps chains even lengths, otherwise table turns into a linked list...

Chain Node Class

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

Non-generic version using Object references.

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

Hash Table class

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

Hash table has a fixed size array of nodes.

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.

Open Hashing (1)

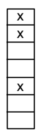
- Have seen linked lists used as the overflow technique in an hash table.
- There is one other major technique for handling hash collisions: open hashing.
  - Also known as linear probing.

Open Hashing (2)

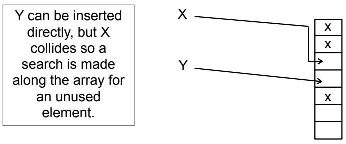
- The array holds the data itself (object reference), not chains of nodes holding the data.
- If the slot determined by the hash function is full, linearly search down the array for the next empty slot.

Open Hashing (3)

Node array with some elements used (marked x).



Open Hashing (4)



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.

Hash Table Summary

- Various implementations.
- Maps one type to another.
- Widely used, useful data structure.
- O(1) access and update.

Example code

- See the 1008 web page for example code for a Linked List, Binary Tree and Chained Hash Table.
- Make sure you study this code and understand how it works.
- See PartII of text book for in-depth description of data structures.
- See Java Collections Framework for classes provided with Java.

You Should...

- Understand the principles of lists, trees and hash tables.
- Understand iterators.
- Be able to implement straightforward list, binary tree and hash table classes.
- Be able to write code that uses chains or trees of element/node objects.
- Be able to select the right data structure for the job in hand.

Summary

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