

Data Structures G5029

Lecture 5

Kingsley Sage
Room 5C16, Pevensey III
khs20@sussex.ac.uk

© University of Sussex 2006

Lecture 5

- Order
 - Arrays vs binary search
- Generic methods
 - The `Comparable` interface
- Binary search trees

Order

- Many data structures provide sorting and searching routines.
- The idea of sorting pre-supposes that the data items are ordered in some way.
- Searching does not pre-suppose an order, but it is quicker to search for an item if the items have been previously sorted.
- Many data types have an intrinsic ordering (like integers). Other orderings can be created e.g. ASCII values for characters, lexicographical values for strings.
- Order affects searching efficiency ...

Efficiency of search

- Let's say we have an un-ordered set of N integers. What is the computational complexity associated with finding the largest one?
 - We need to examine each one, so order N , or $O(N)$.
- And if the set of integers is ordered?
 - It will be at one end of the set, so for array, computational complexity is constant time k , but we do need to factor in sorting the integers in the first place.
 - What about for a linked list?

Efficiency of search

- Let's say we have an un-ordered set of N entries in a telephone directory. What is the computational complexity associated with finding a particular entry in the directory?
 - Well, it's $O(N)$ again.
- What if the directory is sorted?
 - Can perform a binary search. Chop the list in half and see if the end value is the target value, greater or less. If greater, the target is in the lower part of the set, if less, the target is in the upper part of the list. Repeat this process until there is a list of just one – the target value. The number of chops required is $\log_2(N)$, so the complexity is $O(\log_2 N)$ instead of $O(N)$.

Generic methods

- Let's say that we want to define a method `contains` that is true if an item of a particular type is present in an array of that type ...

```
public static boolean contains(float item, float[] array);  
  
// For different types, you might overload contains so  
// there is a method for ints, strings, whatever ...
```

- It would be an advantage if we only had to write a single routine for searching an array of objects, no matter what its elements may be. All that matters is that such objects provide a method for comparing objects in that domain ...

Generic methods

```
public interface Comparable {
    public int compareTo(Object o);
}
```

- The wrapper classes for Java primitive types all implement this interface.
- `compareTo` returns a negative integer, zero, or a positive integer according to whether this object is less than, equal to, or greater than the specified object `o`.
- See the on-line course notes for information on implementing `compareTo` for you own particular class.

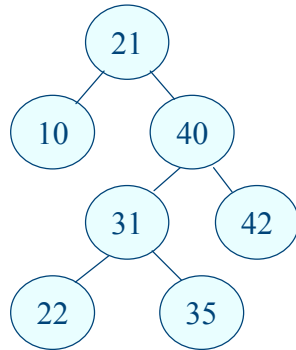
Generic binary search

- We can use this generic approach for binary search ...

```
public static final boolean contains(Comparable item,
                                     Comparable[] array,int n) {
    int low = 0;
    int high = n-1;
    while (low <= high) {
        int mid = (low+high) / 2;
        int compare = item.compareTo(array[mid]);
        if (compare < 0) {
            high = mid - 1;
        } else if (compare > 0) {
            low = mid + 1;
        } else {
            return true;
        }
    }
    return false;
}
```

Binary search trees

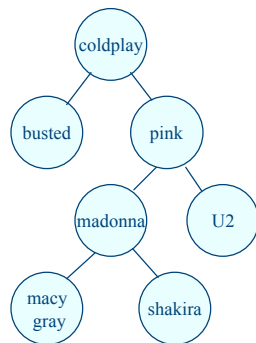
- A binary tree is an important linked data structure.



- A binary tree is either empty, or else consist of some data item together with a left and right descendant binary tree.
- A binary search tree is a binary tree in which the value stored at each node of the tree is greater than the values stored in the left sub tree, and less than the values stored in the right sub tree.

Binary search trees

- Could organise using lexicographical values

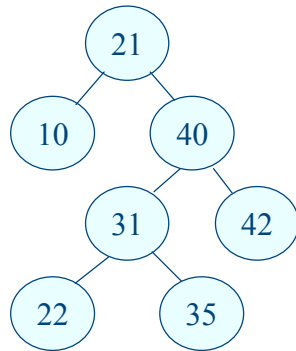


LHS: value is "smaller"
RHS: value is "larger"

The concepts of "smaller" and "larger" are defined according to the data you are processing ...

Binary search trees

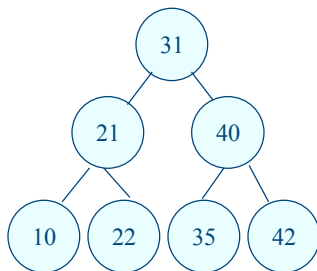
- If we want to find out whether a given object is present in a binary search tree:



- Suppose the item is the number 39. Beginning at the root of the tree, we see that 39 is greater than 21.
- If 39 is in the tree, it must be in the right hand tree with respect to the root node.
- We see that 40 is greater than 39, so we turn to the left hand tree with respect to 40.
- We eventually establish that 39 is not in the tree.
- We are not limited to just integers.

Binary search trees

- Note that all comparisons are made on a path through the tree.



- For efficiency, binary trees are much more useful if they are balanced.
- This means that the path lengths from the root node to each of the leaf nodes are roughly the same. Our previous tree could have been more balanced such as shown on the left.
- There are many algorithms for re-balancing trees. These will be discussed in a later course.

Empty trees and node trees

- We can now start to implement a Java class for binary trees ...

```
public abstract class SearchTree {  
  
    public static SearchTree empty() {  
        return new EmptyTree();  
    }  
  
    public abstract boolean isEmpty();  
  
    ...more methods  
}
```

- From this base abstract class, we can build `EmptyTree` and a `NodeTree` ...

Empty trees and node trees

```
class EmptyTree extends SearchTree {  
    protected boolean isEmpty() {}  
    public boolean isEmpty() {  
        return true;  
    }  
    ... more methods  
}  
  
class NodeTree extends SearchTree {  
    private Comparable data;  
    private SearchTree left;  
    private SearchTree right;  
    protected NodeTree(Comparable item) {  
        data = item;  
        left = empty();  
        right = empty();  
    }  
    public boolean isEmpty() {  
        return false;  
    }  
    ... more methods  
}
```

Next time ...

- Binary heaps and priority queues