15: THE BINARY TREE DATA STRUCTURE

Binary Search

The binary search is the standard method for searching through a sorted array. It is much more efficient than a linear search, where we pass through the array elements in turn until the target is found. It does require that the elements be in order.

The binary search repeatedly divides the array in two, each time restricting the search to the half that should contain the target element.

In this example, we search for the integer **5** in the 10-element array below:

2	5	6	8	10	12	15	18	20	21

Loop 1 - Look at whole array
Low index = 0, high index = 9
Choose element with index (0+9)/2 **= 4**

2	5	6	8	10	12	15	18	20	21

Compare value (10) to target
10 is greater than 5, so the target must be in the lower half of the array
Set high index = (4-1) = 3

Loop 2
Low index = 0, high index = 3
Choose element with index (0+3)/2 **= 1**

2	5	6	8	10	12	15	18	20	21

Compare value (5) to target
5 is equal to target
Target was found, index = 1

Efficiency

The maximum number of comparisons required to find a target in an array of n elements is (the number of times that *n* can be divided in two) + 1.

n	Maximum comparisons
3 to 4	2
5 to 8	3
9 to 16	4
17 to 32	5
…	…
1024	10

This means that to search an array of 1024 elements would take **at most 10** comparisons using a binary search, but could take **up to 1024** comparisons using a linear search.

Binary Data Structures

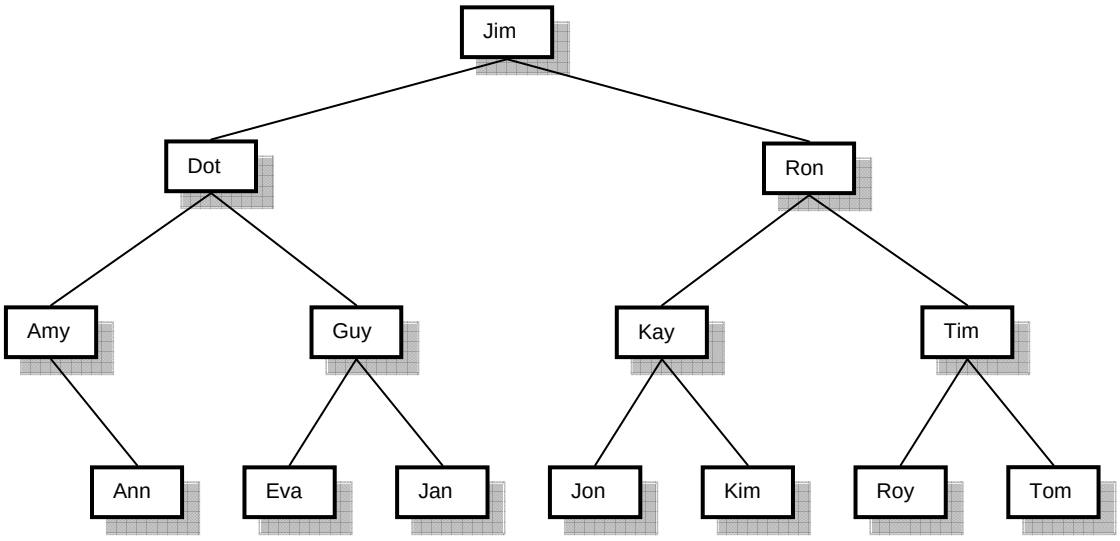
A linked list structure is not efficient when searching for a specific item as the node can only be accessed sequentially.

The **binary search algorithm** suggests a data structure which can be implemented using dynamic storage and allows searching to be done efficiently. Consider the order in which the elements of the following array would be accessed in a binary search:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
Amy	Ann	Dot	Eva	Guy	Jan	Jim	Jon	Kay	Kim	Ron	Roy	Tim	Tom

The first step would be to divide the array in two and compare the target with element 6 (*Jim*). Depending on the result of the comparison, the next element to be checked would be either 2 (*Dot*) or 10 (*Ron*). If it is Dot, then we would next check either 0 (*Amy*) or 4 (*Guy*), and so on.

We can describe the possible sequence of comparisons in a diagram:



For example, to search for the target *Jon* in the array, we would have to compare the target with the elements *Jim*, *Ron*, *Kay* and *Jon* (try it for yourself to check this).

This diagram looks a little bit like a family tree, and suggests that a **tree** is a suitable data structure. Tree structures are commonly used in computer science. The characteristic features of a tree are that each element may have several successors (or "**children**"), and every element except the topmost one has a unique predecessor (or "**parent**"). Tree structures are **hierarchical** rather than linear, (whereas a List is a linear structure). Examples of tree structures include computer file systems and the inheritance structure for Java classes.

Binary Trees and Binary Search Trees

Our diagram is a special kind of tree, called a **binary search tree**, which is ideal for storing data for efficient searching. The binary search tree is a hierarchical structure in which data access is similar to a binary search algorithm.

A binary search tree is itself a special kind of **binary tree**. A binary tree is a tree which is either empty or consists of a node called the root, together with two children called the **left subtree** and the **right subtree** of the root. Each of these children **is itself a binary tree.**

A binary search tree satisfies the following additional conditions:

- Each element has a key value which is used to order the elements
- The keys of all the elements (if there are any) in the left subtree of the root precede the key in the root
- The key in the root precedes all keys (if any) in the right subtree
- The left and right subtrees of the root are again search trees.

Look at the diagram above and check that the element *Jim* is a binary search tree.

Operations

The main primitive operations of a binary search tree are:

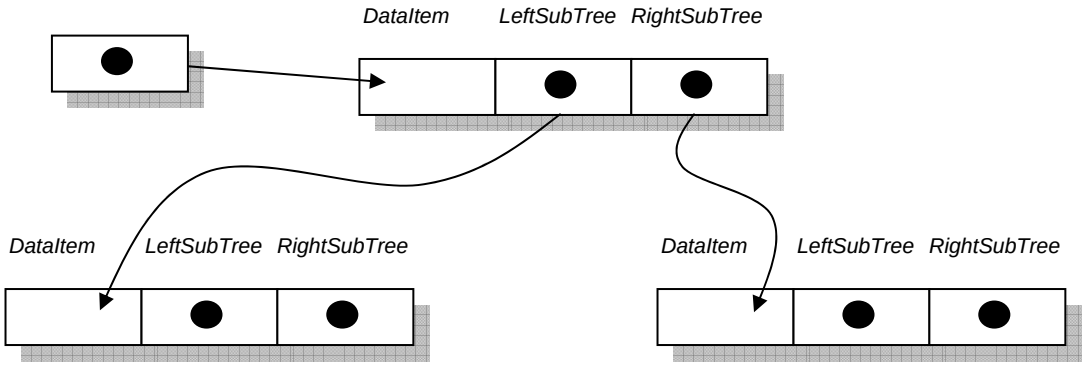
Add	adds a new node
Get	retrieves a specified node
Remove	removes a node
Traversal	moves through the structure

Additional primitives can be defined:

IsEmpty	reports whether the tree empty
IsFull	reports whether the tree is full
Initialise	creates/initialises the tree
Destroy	deletes the contents of the tree (may be implemented by re-initialising the tree)

Storing a binary search tree in a dynamic data structure

Each node contains data AND a references to the left and right subtrees. An empty subtree is represented by a NULL reference. Each subtree is itself a binary search tree.



For some purposes it is useful to include a reference to the parent tree, but for simplicity we will not do this here.

Traversal methods

Traversal is the facility to move through a structure visiting each of the nodes once. With a binary tree, there are three actions associated with a traversal:

V: visit the node (for example, to output the data stored in that node)

L: traverse the left subtree

R: traverse the right subtree

There are three commonly used ways of organizing the traversal

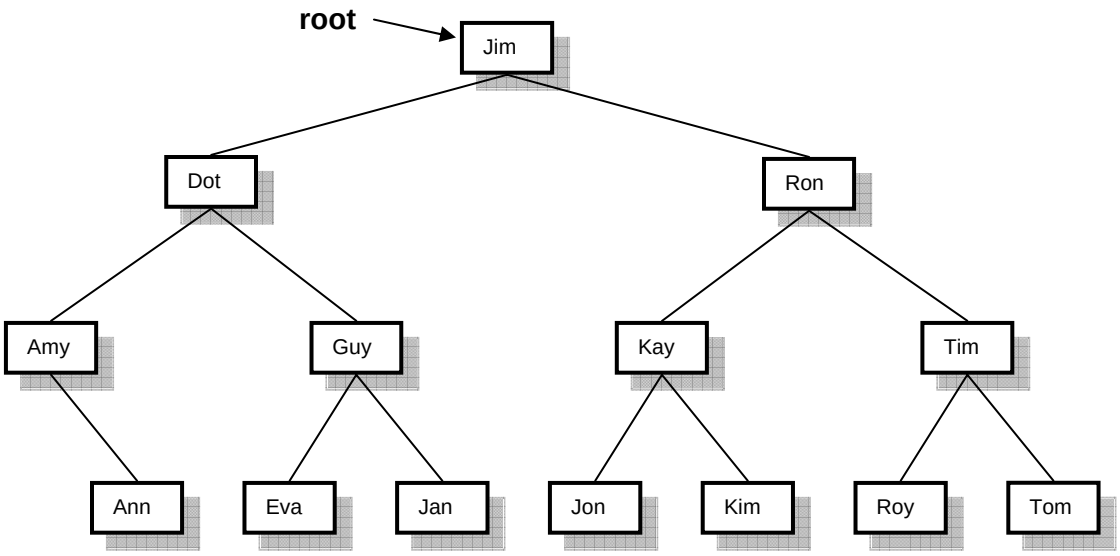
VLR PreOrder (i.e. visit the node then traverse the subtrees)

LVR InOrder (traverse the left subtree, visit the node then traverse the right subtree)

LRV PostOrder (traverse the subtrees then visit the node)

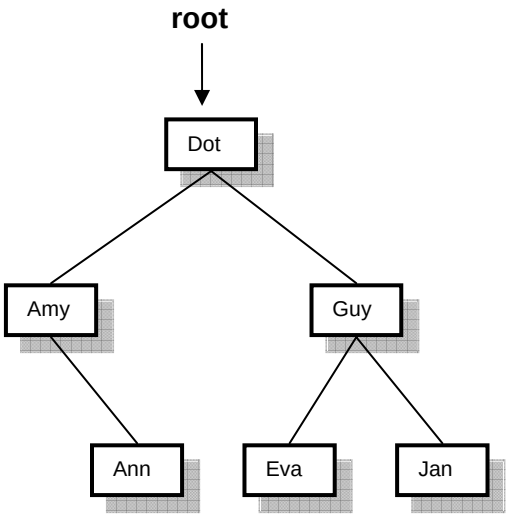
Example: PreOrder

Step 1. Root = *Jim*
Display *Jim* then traverse its left subtree (root = *Dot*) and then its right subtree (root = *Ron*)



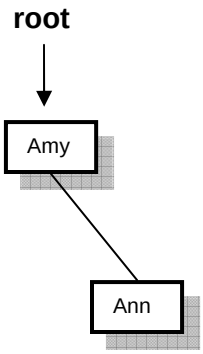
Display: *Jim*

Step 2. Root = *Dot* (*Jim*.LeftSubTree)
Display *Dot* then traverse its left subtree (root = *Amy*) and then its right subtree (root = *Guy*)



Display: *Jim Dot*

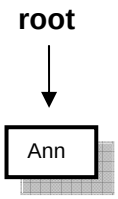
Step 3. Root = *Amy* (*Dot*.LeftSubTree)
Display *Amy* then traverse its left subtree (root = *NULL*) and then its right subtree (root = *Ann*)



Display: *Jim Dot Amy*

Since the right subtree of Amy is empty we then move onto the right subtree.

Step 4. Root = *Ann* (*Amy*.RightSubTree)
Display *Ann* then traverse its left subtree (root = *NULL*) and then its right subtree (root = *NULL*)



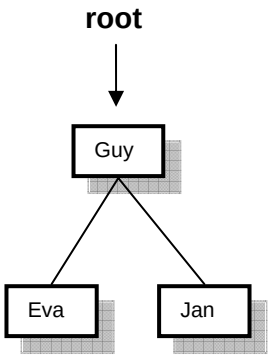
Display: *Jim Dot Amy Ann*

Since both of *Ann's* subtrees are empty we have finished traversing the tree with Root = *Ann*.

This completes the traversal of the right subtree of *Amy* and thus completes *Amy*,

The tree with root *Amy* is the left subtree of *Dot*, so we now continue with the right subtree of *Dot* (Root = *Guy*).

Step 5. Root = *Guy* (*Dot.*RightSubTree)
Display *Guy* then traverse its left subtree (root = *Eva*) and then its right subtree (root = *Jan*)



Display: *Jim Dot Amy Ann Guy*

Remaining steps
We display Eva and Jan and this completes the right subtree of Dot, and thus the left subtree of Jim.

Display: *Jim Dot Amy Ann Guy Eva Jan*

We now traverse the right subtree of Jim in a similar way, giving a final output of

Display: *Jim Dot Amy Ann Guy Eva Jan Ron Kay Jon Kim Tim Roy Tom*

EXERCISE: Traversal

1. Write down the output for **InOrder** traversal of the example tree.

   What do you notice about the final output?

2. Write down the output for **PostOrder** traversal of the example tree.

Recursion and the binary search tree algorithms

The algorithms used to implement a binary tree can make use of **recursion**. A recursive operation can **call itself**. This can result in a lot of work being done by very little code. For example, the PreOrder traversal above uses the following algorithm:

```
PreOrder(Root)
If Root is not NULL
        Display Root.DataItem
        Call PreOrder(Root.LeftSubTree)
        Call PreOrder(Root.RightSubTree)
End If
```

Other operations can also make use of recursion:

Get

Searches for a specified target. The target is a key value, and the operation will return the data item with that key.

e.g. *Get(Root, Guy)*

Algorithm:

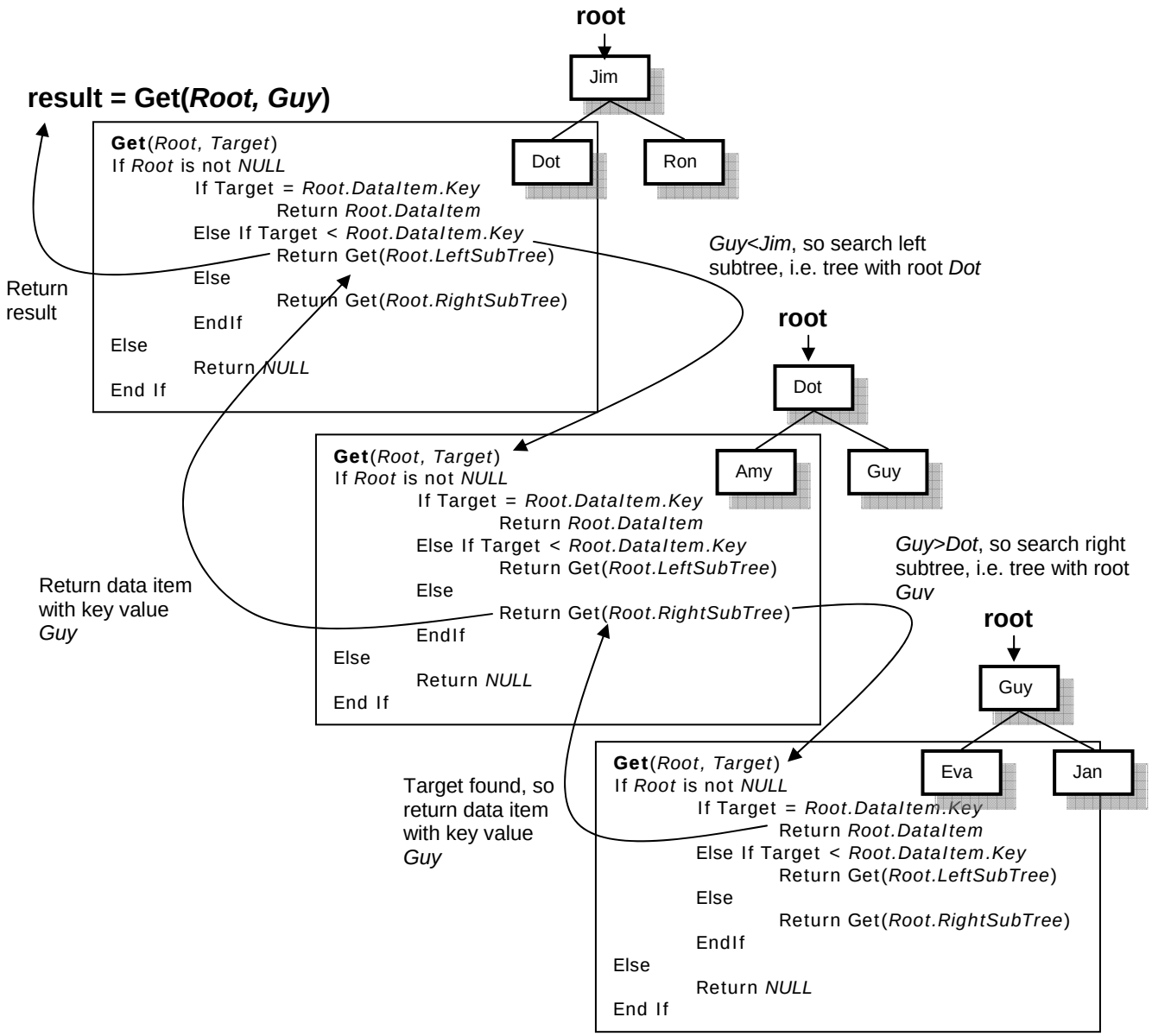
```
Get(Root, Target)
If Root is not NULL
        If Target = Root.DataItem.Key
                Return Root.DataItem
        Else If Target < Root.DataItem.Key
                Return Get(Root.LeftSubTree)
        Else
                Return Get(Root.RightSubTree)
        EndIf
Else
        Return NULL
End If
```

Example: Target = Guy

The operation must start at the root Jim and then go through the following stages:

1. ***Guy < Jim*** go to left subtree of *Jim (*root is *Dot)*
2. ***Guy > Dot*** go to right subtree of *Dot (*root is *Guy)*
3. ***Guy = root*** target found, return data item of node *Guy*

The target data item is passed by return statements back to the original operation call, as shown in the diagram below:



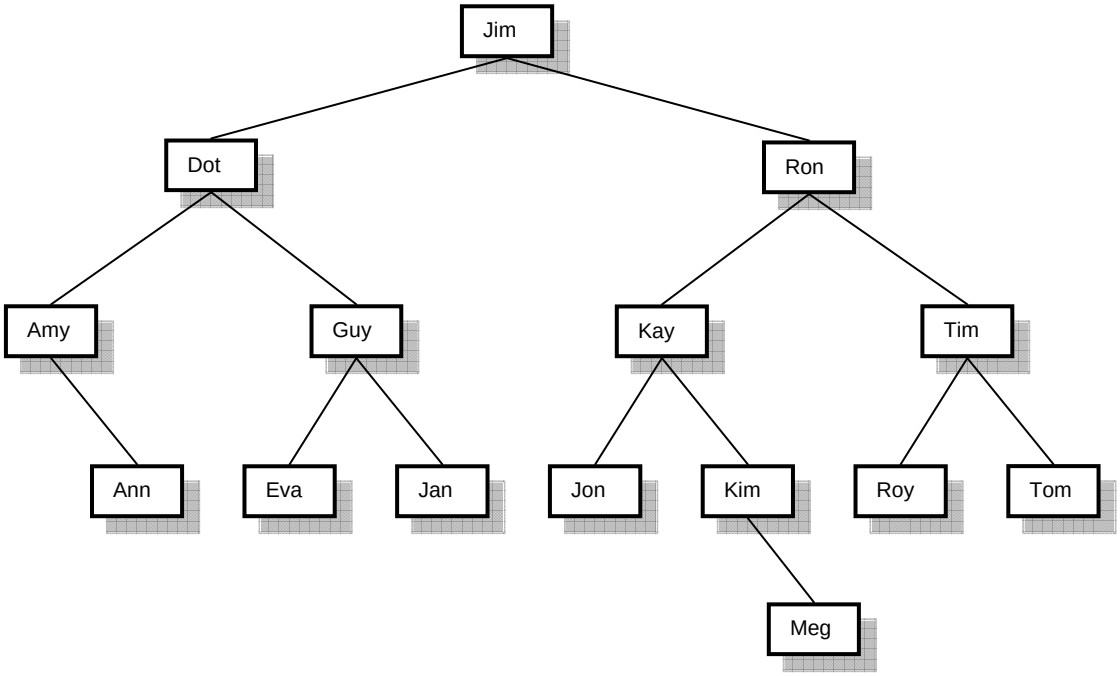
Add

Adds a new node to the tree.

e.g. *Add(Root, Meg)*

The Add operation is similar to the Get operation in that you have to recursively descend the tree until you find the appropriate place to add the new node. For example, if you want to add a new node with key Meg, the operation must start at the root Jim and then go through the following stages:

1. ***Meg > Jim*** go to right subtree of *Jim*
2. ***Meg <Ron*** go to left subtree of *Ron*
3. ***Meg* >Kay** go to right subtree of *Kay*
4. ***Meg* >Kim** go to right subtree of *Kim* which is NULL, therefore add Meg as a right child of *Kim*

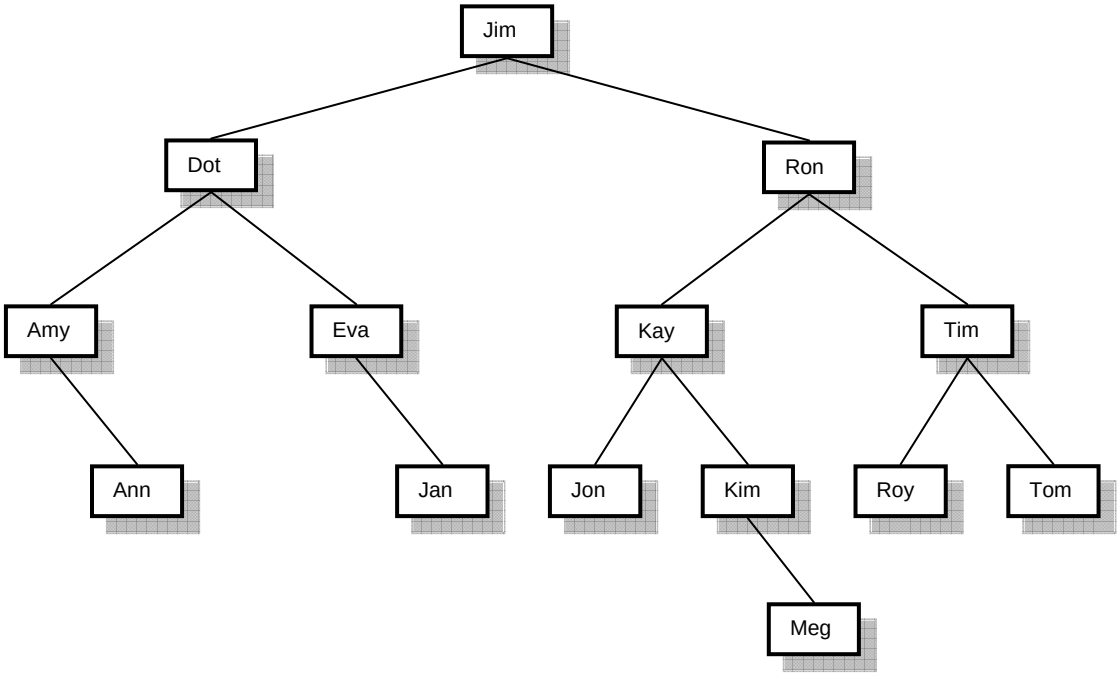


Remove

Removes a node from the tree

e.g. *Remove(Root, Guy)*

The remove operation can be rather involved, as it may be necessary to rearrange nodes so that the remaining structure is still a valid binary search tree. For example, if *Guy* is removed, a possible new structure would be:



Note that:

- *Eva* is now the right subtree of *Dot*, rather than *Guy*
- *Jan* is now the right subtree of *Eva,* rather than *Guy*

Removing a node with empty subtrees, known as a **leaf node** (e.g. *Meg*) is straightforward as no rearrangement is required.

Algorithms to remove a node and change the attachments of other nodes as required are quite complex, and it can be useful to have a *parent* reference in each node.

EXERCISE: Algorithms

1. Describe the steps required to search for

   (a) Roy (b) Ian

2. Describe the steps required to add

   (a) Abi (b) Ken (c) Rik

3. Draw a diagram of a possible tree structure after removing:

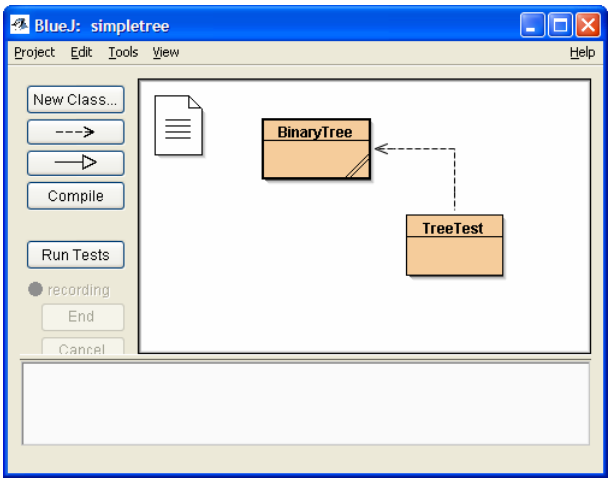
   (a) Ann (b) Ron

EXERCISE: Looking at a Binary Tree

In this example you will look at a Java implementation of a binary tree. For simplicity, this version stores *Strings* rather than *Objects*.

The Java Collections Framework includes a tree class *TreeMap*. The tree we use here is much simpler than this.

Download *simpletree.exe* from your course web site and extract its contents. You should now have a BlueJ project called *simpletree*. Open the project. The BlueJ window should look like this:



You should look at the code for each of these classes. Notice that the *BinaryTree* class is quite short – recursive code does a lot of work with a few lines of code.

Create a new instance of *BinaryTree.*

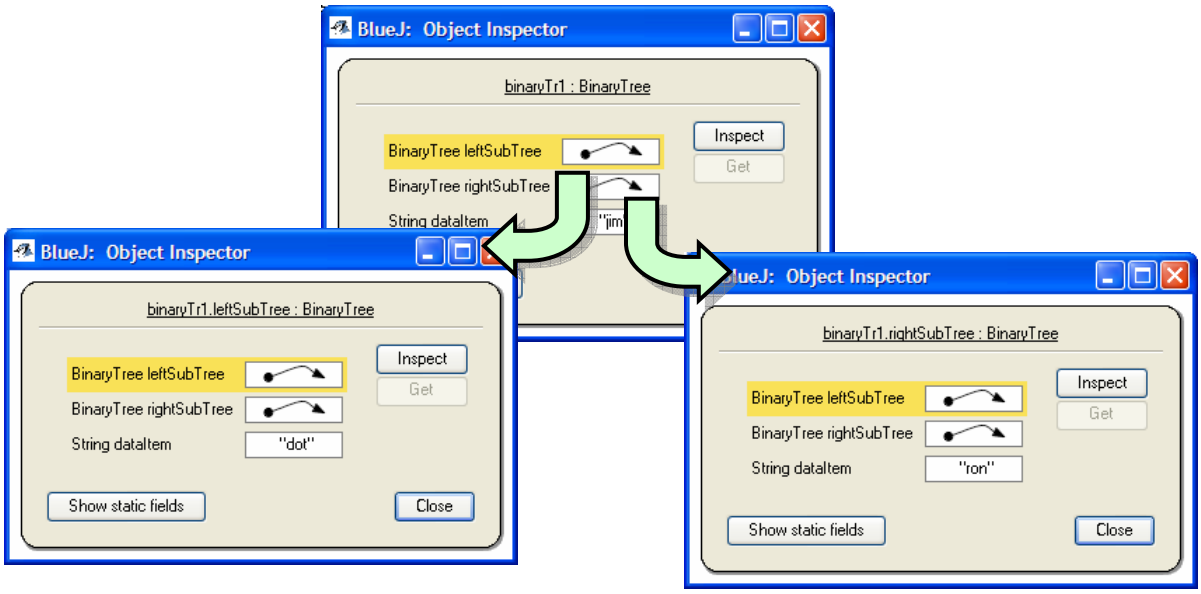
Create a new instance of *TreeTest* and select your *BinaryTree* instance in the object bench as the parameter in the constructor. This means that you will be testing the *BinaryTree* you created in the previous step.

Call the *populateTree1* method of your *TreeTest* instance. This adds some strings to the tree so that it contains the names in the diagram on page 3.

Inspect the *BinaryTree*. Its attributes are a left subtree, a right subtree and a data item.

What is the type of the subtree attributes?

What do you expect the data items for the two subtrees to be (look at the diagram on page 3)?



By inspecting subtrees, find the node “Kim”.

What nodes did you have to inspect to find the target?

By inspecting all the possible subtrees, explore the tree **and draw a diagram of the nodes**. Compare this to the diagram on page 3.

Create another new instance of *BinaryTree.* Create a new instance of *TreeTest* and select your new *BinaryTree* as the parameter in the constructor.

Call the *populateTree21* method of your *TreeTest* instance. This adds some strings to the tree so that it contains a different set of names.

By inspecting subtrees, find the node “Kent”.

What nodes did you have to inspect to find the target?

By inspecting all the possible subtrees, explore the tree **and draw a diagram of the nodes**.

Add a new node *“lara”*

By inspecting subtrees, find where the new node was added **and add this to your diagram.**

Further Exercise

The *BinaryTree* class does not currently have any traversal methods. As a further exercise, add and test suitable traversal methods