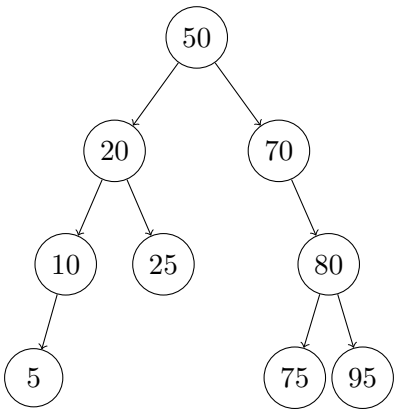

Figure 1: An example binary tree. The node containing 50 is the root; the nodes containing 50, 20 and 80 have two children, the nodes with 25, 75 and 95 have no descendents, the node with 10 only has a left child, the one with 70 only has a right child.

```
1 struct node
2 {
3   int entry;
4   struct node *left;
5   struct node *right;
6 };
```

Table 1: A node, it has a variable to store the entry and pointers to the left and right children.

IV.2 Data structures - binary trees

A binary tree is a data structure somewhat similar to a linked list except each node leads on to up to two other node, rather than just one. With the restriction that there are no loops, it can be thought of as looking something like a tree. The equivalent of the head, the node that has no other node pointing to it, is called the *root*, but, confusingly, it is very common to put it at the top in diagrams. Another confusing thing is that a family tree metaphor is often used, so you talk about a node's *children* or its *descendents*, but the botanical expressions root and *leaf*, a leaf is a childless node. An example binary tree is shown in Fig. 1 and code for a node and for making a node, including the root, is given in Table 1 and Table 2.

The most common application for a binary tree as a data structure is the binary search tree. In a binary search tree the nodes are arranged so that a nodes left child has a smaller value and its right child a greater one. This makes searching easy, you start at the root and at each node go left of right according to whether the value you are search for is bigger or smaller than the node value. For this to work the nodes must be added the same way, so that a left child is always smaller and a right child bigger; doing this is easy, basically you act as if you were searching for the entry you wish to add and if the next place you want to look is a NULL, you add the node there. There is code for adding a node at Table 3 and for finding a node at Table 4.

```
1  struct node * make_node(int new_entry)
2  {
3    struct node * a_node=(struct node *)malloc(sizeof(struct node));
4    a_node->entry=new_entry;
5
6    return a_node;
7  }
```

Table 2: Functions for making a node.

```
1  struct node * add_node(struct node * here,int new_entry)
2  {
3
4    if(here==NULL)
5      return make_node(new_entry);
6
7    if(new_entry<here->entry)
8        here->left = add_node(here->left,new_entry);
9    else
10       here->right = add_node(here->right,new_entry);
11   return here;
12
13 }
```

Table 3: Adds a node. This is done recursively, searching to the left or right until it gets to NULL and making a new node there. See how cleverly it does the recursion, it links the current node to the return value and then returns itself unless it is NULL, in which case it returns the new node.

```
1  struct node * find_entry(struct node * root, int a_entry)
2  {
3    if(root==NULL)
4      return root;
5    else if(root->entry==a_entry)
6      return root;
7
8    if(root->entry>a_entry)
9      return find_entry(root->left,a_entry);
10   return find_entry(root->right,a_entry);
11 }
```

Table 4: Finds a node. This is done recursively down the tree and returns a pointer to the node containing the entry, or a NULL pointer if the entry isn't found.

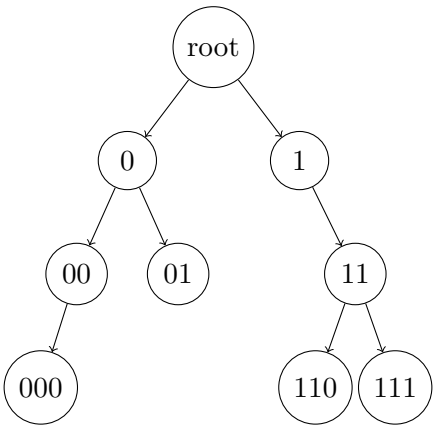

Figure 2: A route map. Here the nodes are labelled in a way that describes there position, starting with the root a 0 indicates a left turn and a 1 a right turn.

As for computational complexity; this depends on how balanced the tree is. Imagine the tree is balanced, so at each node the number of left descendents is roughly equal to the number of right descendents. Now, consider finding; the run time is

$$T_n = c + T_{n/2} \tag{1}$$

since at each node you either go left or right, roughly halving the number of nodes still under consideration. Hence

$$T_n \in O(\log n) \tag{2}$$

Making a node is the same, to make a node you first find its position and then add it, adding it is $O(1)$ so making is $O(\log n)$.

One way to think about this is to note that when finding you only pass through each level once, where a level is made up of all the nodes the same number of nodes away from the root. How many levels are there?

One way to think about this is to imagine labelling all the nodes with a binary number describing its position, so you have a node numbered x its left node is $x0$ and its right node is $x1$. To get to the node you would start at the root and look at the first bit, if it zero you go to the left node, if it is one, you go to the right node. An example is given in Fig. 2. Now, if all the nodes are packed in together so there are no wasted labels then the longest words are about $\log(n+1)/2$ bits long. Roughly speaking, this is because about half the nodes are on the lowest level. Thus the number of levels, and hence the number of algorithmic steps goes like $\log n$.

Of course, all this depends on the tree being balanced. This is an important issue. In Fig. 3 we see the tree corresponding to 10, 20, 25, 30, 35, 40, 45, 50 and the tree corresponding to the same sequence added in a different order 30, 20, 40, 10, 25, 35, 45. Obviously this would have a big impact on search times. In fact, we'll see there is a way to balance the trees as the nodes are added.

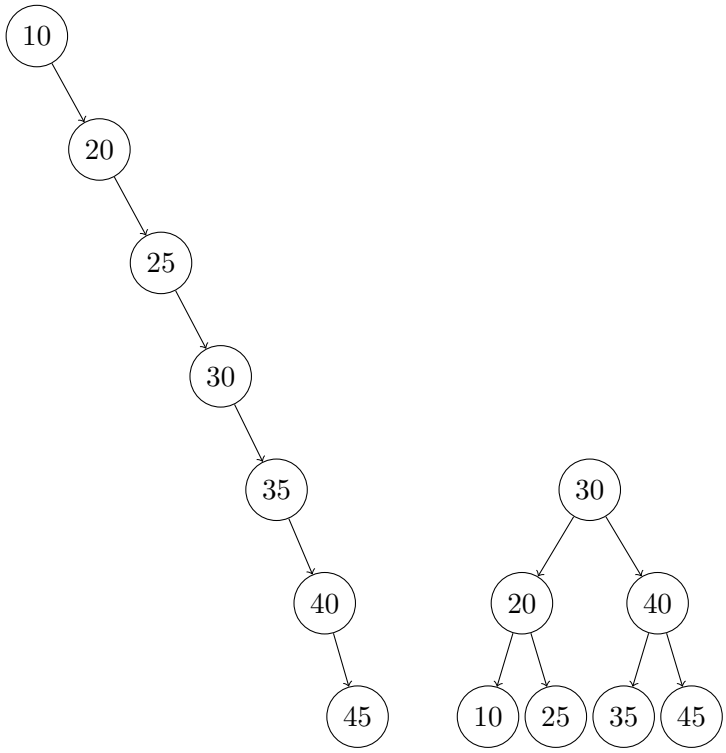

Figure 3: Trees for the same data. LEFT: Unbalanced tree, seven levels. RIGHT: Balanced tree, three levels.