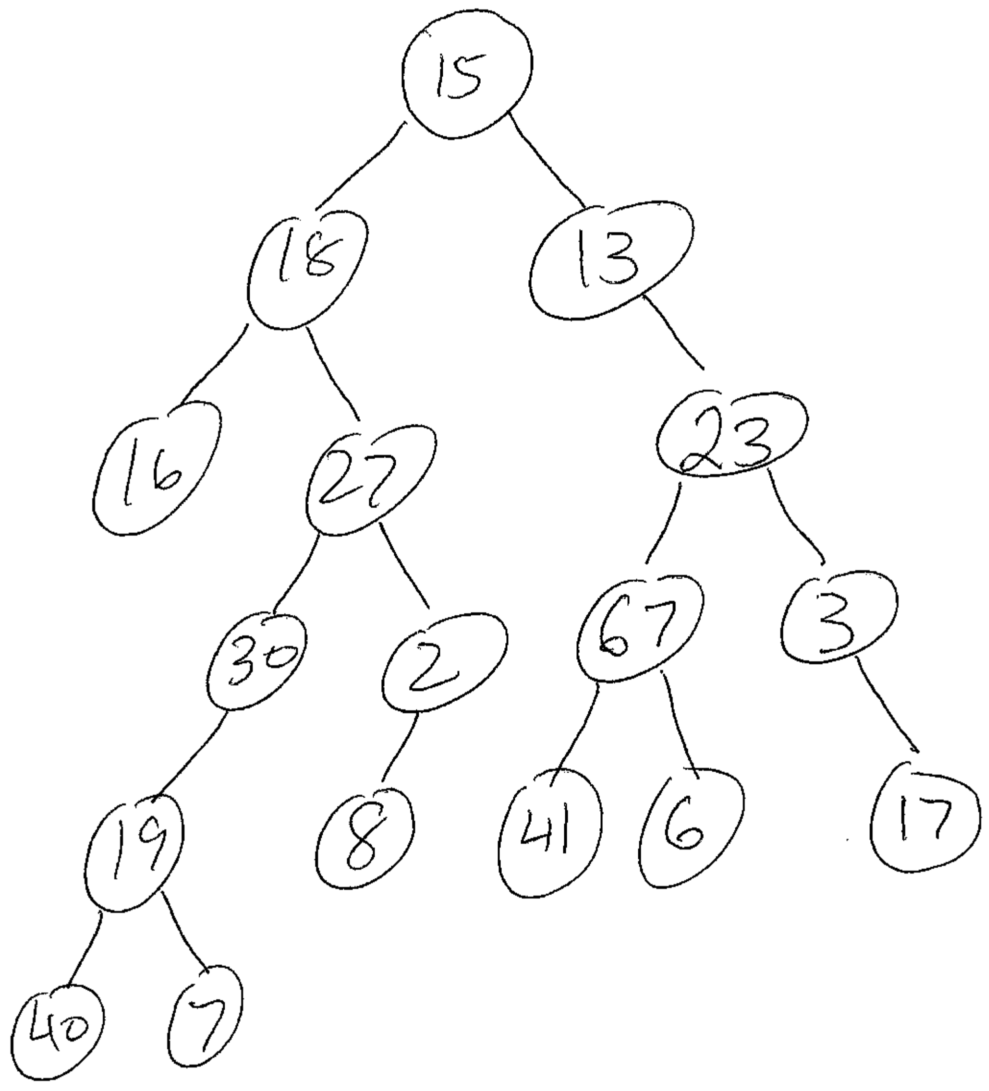
The Idea of a Tree

One of the problems we had with linked lists was that it took a long time to get to an arbitrary node of a linked list. It would be nice if we could have a linked structure were nodes were more accessible.

If you think of a tree with branches, and each point where branches intersect as a node, including leaves, you find a structure with a huge number of nodes, but one where the path to any one particular node is not too long.

An upside down picture of that sort of tree might look like this:



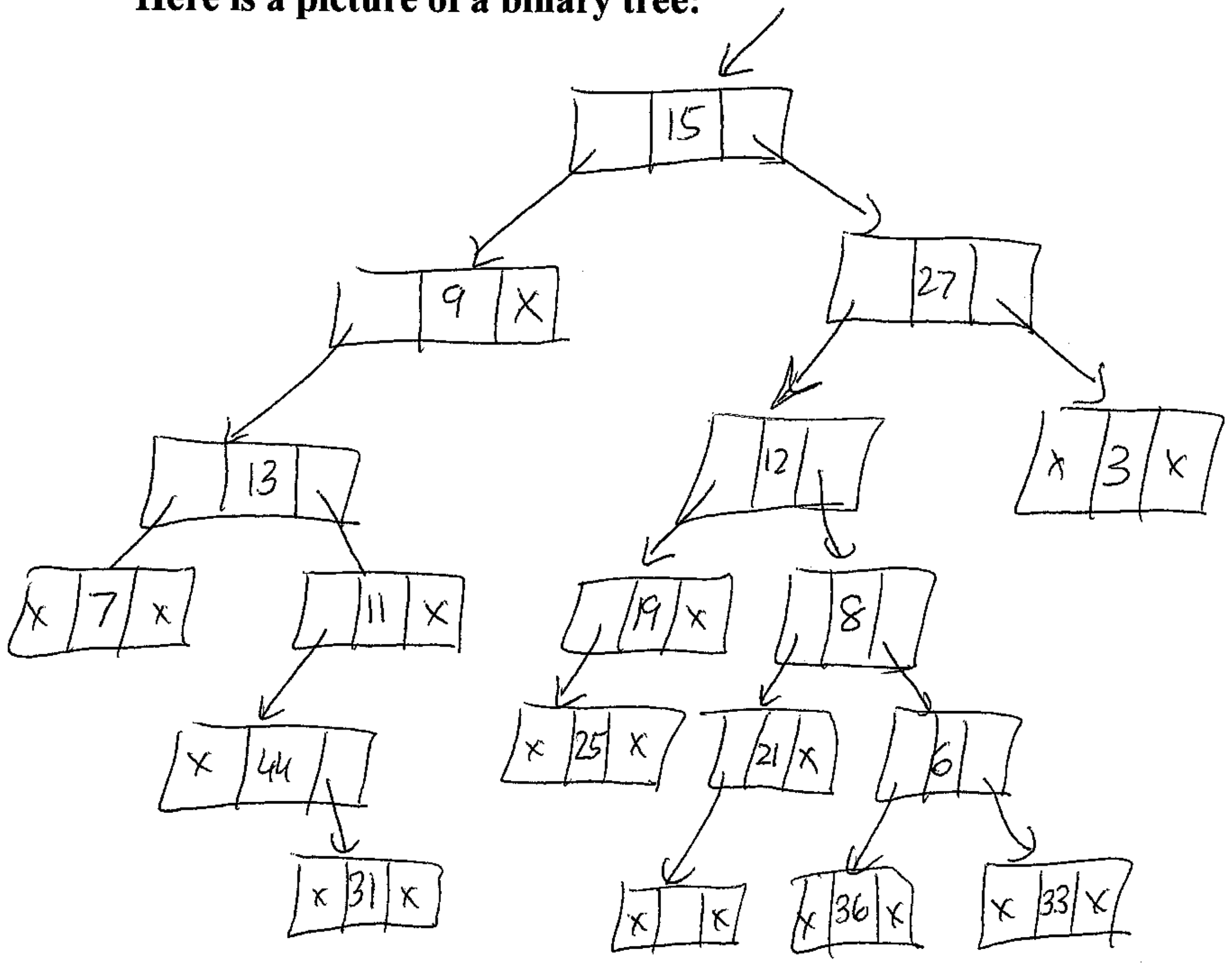
Binary Trees

Now, what we could do with that picture above is make each of the branches of the tree a pointer to a node, and each node contain a data value along with several pointers to other nodes.

To simplify things, we will limit the number of pointers to other nodes to be 2 or less, and also not allow any "circular" references. (These are when you follow a set of pointers from node to node and eventually get back to the node you started from.)

Any linked structure that fits these requirements is a binary tree.

Here is a picture of a binary tree:



Structure to Declare a Node in a Binary Tree

Pictorially, a node of a binary tree looks very similar to a node in a linked list, except instead of having 1 field as a pointer field, we should have 2 fields as pointer fields. Hence, we could declare a node in a binary tree as follows:

```
typedef struct tree_node {
    int data;
    struct tree_node *left;
    struct tree_node *right;
} tree_node;
```

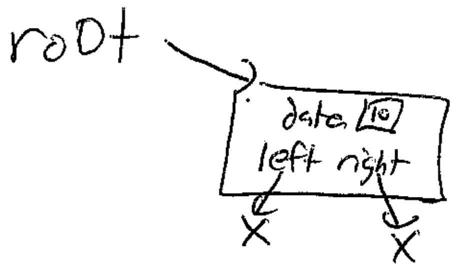
Now, to declare an empty binary tree, we simply declare a pointer to a Tree_Node as follows:

```
tree_node *root;
```

To add a single node to this tree, we could do something like:

```
root = (tree_node *)malloc(sizeof(tree_node));
root->data = 10;
root->left = NULL;
root-> = NULL;
```

Now, the picture looks like:



Traversal of a Binary Tree

To analyze traversing a binary tree, we will assume that we already have a binary tree that is filled. (Inserting nodes into a binary tree is no easy task.)

Also, traversing a binary tree itself isn’t trivial. In a linked list it was clear that the nodes were in some sort of order and we could go to each node in that order. With a binary tree, that order is suddenly not so clear.

Once again, we will have to think about this problem recursively. (Even though you can do linked list traversals iteratively without much pain, the same can not be said of traversing a binary tree, take my word for it!)

Consider the three components of a binary tree:

1) A node, (the root node)
2) A left subtree
3) A right subtree

We can traverse these 3 components in any order that we want. (Typically however, the left subtree is traversed before the right subtree, which leaves us three options to traverse a binary tree.)

Inorder Binary Tree Traversal

An inorder tree traversal visits the three parts of the tree in this order:

1) left subtree
2) root node
3) right subtree

This traversal is the most common because it is typically used to go through a sorted list in order stored in a binary tree.

Here is a function that prints out all the data values stored in a binary tree, using an inorder traversal. The code is going to look simple, but what is going on is quite complex.

```
// Performs inorder tree traversal on binary tree, and prints
// out data values in each node in that order.
void inorder(tree_node *current_ptr)

      if (current_ptr != NULL) {
            inorder(current_ptr->left);
            printf("%d ", current_ptr->data);
            inorder(current_ptr->right);
      }
}
```

Preorder Binary Tree Traversal

```
// Performs preorder tree traversal on binary tree, and prints
// out data values in each node in that order.
void preorder(tree_node *current_ptr)

     if (current_ptr != NULL) {
          printf("%d ", current_ptr->data);
          preorder(current_ptr->left);
          prerder(current_ptr->right);
     }
}
```

Postorder Binary Tree Traversal

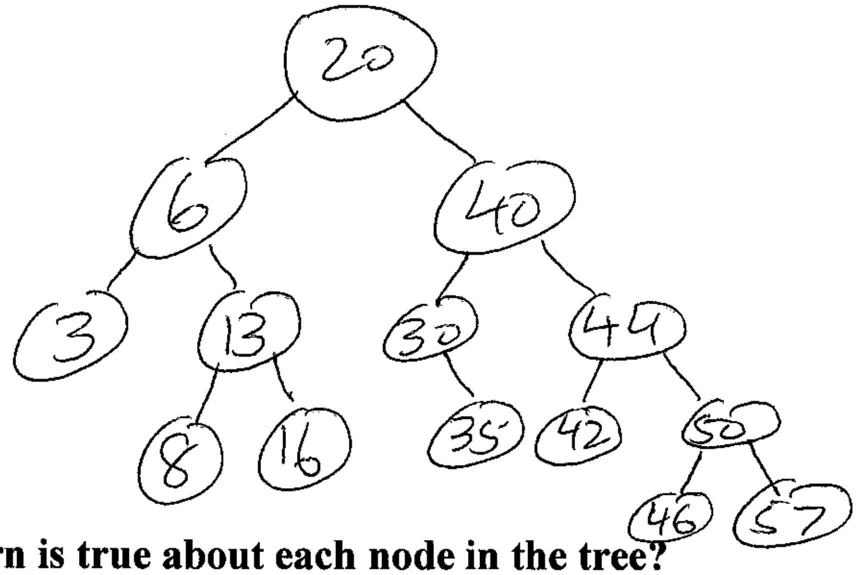
```
// Performs inorder tree traversal on binary tree, and prints
// out data values in each node in that order.
void postorder(tree_node *current_ptr)

     if (current_ptr != NULL) {
          postorder(current_ptr->left);
          postorder(current_ptr->right);
          printf("%d ", current_ptr->data);
     }
}
```

Binary Search Tree

Though it's been nice showing how to traverse a binary tree, it's not quite clear how this new type of data structure can help us. However, what if we added a restriction to a binary tree?

Consider the following binary tree:



What pattern is true about each node in the tree?

For each node N, all the values stored in the left subtree of N are LESS than the value stored in N.

Also, all the values stored in the right subtree of N are GREATER than the value stored in N.

Why might this property be a desireable one?

Clearly, searching for a value could be easier if this property is satisfied for a tree! (This a tree that adheres to the property is known as a Binary Search Tree.)

Rather than "looking" both directions after inspecting a node, we know EXACTLY which direction to go!

Searching a valid Binary Search Tree

Here's the algorithm:

1) If the tree is NULL, return false.
2) Check the root node. If the value is there, return true!
3) If not, if the value is less than that stored in the root node, recursively search in the left subtree.
4) Otherwise, recursively search in the right subtree.

Here is the corresponding code:

```
int search(struct tree_node *current_ptr,
int val) {

  if (current_ptr == NULL) return 0;

  if (val < current_prt->data)
    return Find(current_ptr->left, val);
  else if (val > current_ptr->data)
    return Find(current_ptr->right, val);

  return 1;
}
```