29. Trees

Herman Kamper

In the last two lectures we built linked lists, stacks and queues. All three are *dynamic data structures*: they grow and shrink as you add and remove data, so you do not have to decide the size up front the way you do with an array. In this lecture we look at the third and last of these: the tree. Trees are useful for storing data of a particular kind, and they let you store and search for it very efficiently.

Trees are also the one place in the course where recursion shows up everywhere. The structure of a tree and the structure of a recursive function match so well that the code is often only a few lines long. If you are comfortable with structures, pointers, `malloc` and recursion, trees bring all four together.

An informal introduction

Before looking at any code, let us look at what a tree is. The figure below is a tree that decides what kind of fruit you are holding.

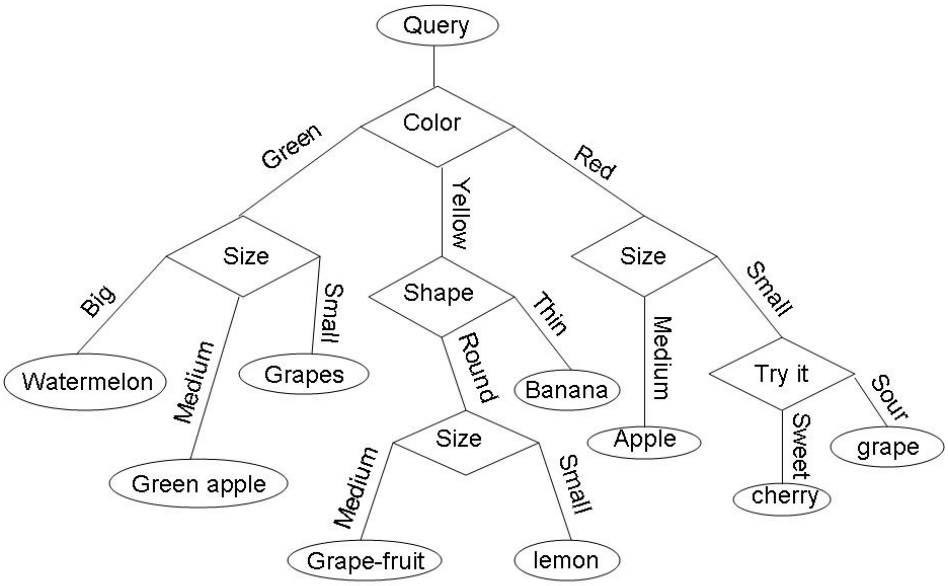


Figure 1: A tree for deciding which fruit you are holding. You start at the top and answer a question at each node, until you end up at a node with no questions left. Image from Wikimedia Commons.

You start at the top and ask a question: what colour is it? If it is red, you ask about the size. If it is small, you taste it: if it is sweet, it is a cherry; if it is sour, it is a grape. You keep on answering questions at each node until you reach a node at the bottom with no more questions, and that is what you were looking for.

Two things about this picture are worth noting. First, it looks upside down: the "root" is at the top and the tree grows downwards. That is how trees are always drawn. Second, this is a fairly complicated tree, because some questions can have three answers (green, yellow or red). In this course we will only look at *binary* trees, where every node has at most two ways out, like a yes or a no, or a left and a right.

Nodes, children and terminology

With a linked list, the node held some data and a single pointer to the next node. A tree node is exactly the same, except that it has *two* pointers, one to each of its two children:

```
struct treeNode {
    struct treeNode *leftPtr;
    int data;
    struct treeNode *rightPtr;
};

typedef struct treeNode TreeNode;
typedef TreeNode *TreeNodePtr;
```

The data here is an integer, but in practice it could be anything, such as a structure that holds a customer's details. The two `typedef` lines are shorthand. The first lets us write `TreeNode` instead of `struct treeNode`. The second makes `TreeNodePtr` mean "a pointer to a `TreeNode`", so that we do not have to write the `*` each time; you will often see this in code for dynamic data structures. The figure below shows a node, with the cross meaning `NULL`.

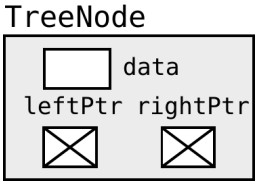


Figure 2: A tree node. It holds some data and two pointers, one to its left child and one to its right child. A new node has both pointers set to `NULL`.

There is some terminology that you need to know when you work with trees. It falls into place quickly. The figure below shows a small tree with the main words marked on it.

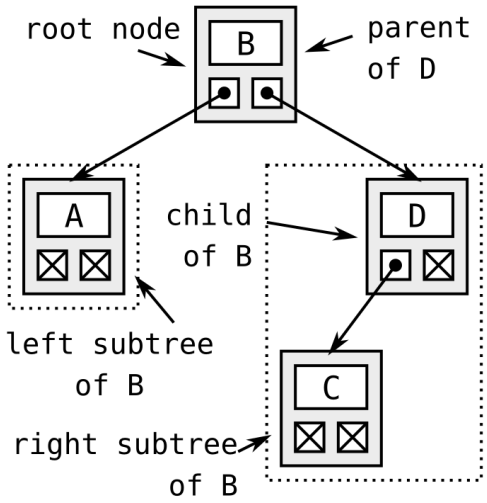


Figure 3: A small tree with its terminology. Node B is the root. A is the left child of B and D is its right child, so B is the parent of D. The nodes inside the dotted lines are the left and right subtrees of B.

- The node at the very top is the *root node*. The root is how you access the tree: a pointer to the root is the only thing that you need to keep.
- The nodes that a node's pointers point at are its *children*: the *left child* and the *right child*. Conversely, the node above is its *parent*.
- A node and everything below it is a *subtree*. Everything below the left child of a node, including the child itself, is the node's *left subtree*, and the same on the right is its *right subtree*.
- A node with no children (both pointers `NULL`) is a *leaf*.

A tree that is empty is represented by a `NULL` root pointer, in the same way that an empty list has a `NULL` start pointer.

Trees make sense for hierarchical data, such as the files and folders on a computer, or a family tree, but we will see that they are particularly good for storing numbers or records so that they can be found quickly.

Sorted binary trees

The trees we will work with are *sorted binary trees*. They follow one rule, which applies at every node:

- All the values in the left subtree of a node are smaller than the value in the node
- All the values in the right subtree of a node are larger than the value in the node

(If you allow duplicate values, you have to decide which side they go on, say the right, and use "larger or equal" there. We will assume there are no duplicates.)

The figure below shows a sorted binary tree. Check the rule at the root: everything to the left of the 5 is smaller than 5, and everything to the right is larger. Then check it at another node, such as the 2: the 1 and the 0 are to its left and smaller; the 4 and 3 are to its right and larger. It holds at every node.

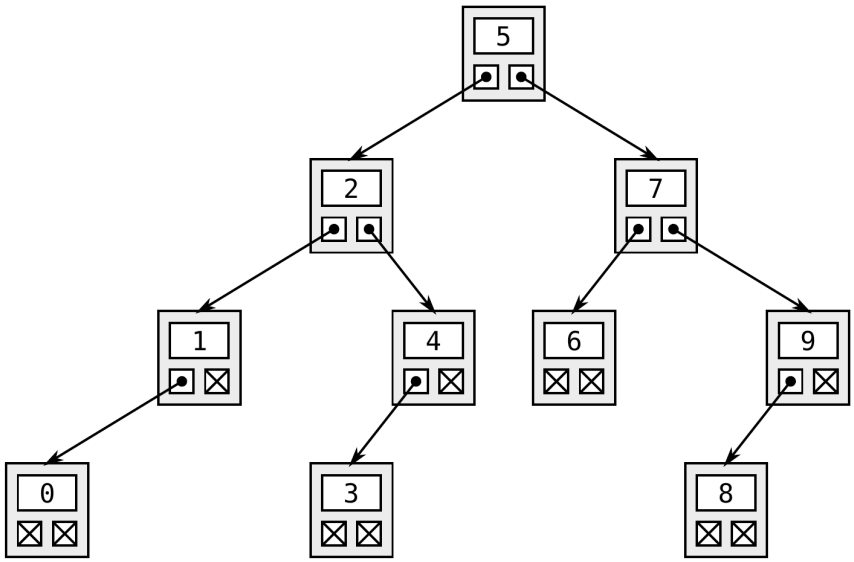


Figure 4: A sorted binary tree. The root holds 5. Everything in its left subtree (0, 1, 2, 3, 4) is smaller, and everything in its right subtree (6, 7, 8, 9) is larger. The same holds at every node.

Why organise data like this? Because it makes finding things fast. Suppose you are looking for the value 6. Start at the root: 6 is larger than 5, so it can only be in the right subtree. Go right. At the 7: 6 is smaller, so go left. There it is. At each node you made a single comparison and threw away a whole subtree. In a linked list you have to start at the beginning and walk along it until you find what you are looking for; with a tree you only follow one path from the root down.

Here is the search written as a recursive function. It returns a pointer to the node holding the value, or `NULL` if the value is not in the tree:

```
TreeNodePtr searchTree(TreeNodePtr treePtr, int value)
{
    if (treePtr == NULL || treePtr->data == value) {
        return treePtr;
    }
    if (value < treePtr->data) {
        return searchTree(treePtr->leftPtr, value);
    }
    return searchTree(treePtr->rightPtr, value);
}
```

Notice the pattern: at every node we do the same thing, either stop or repeat the same procedure

on a smaller tree (the left subtree or the right subtree). This is what a recursive function is. All tree code looks like this.

Inserting into a sorted binary tree

Where does a tree come from? We build it by inserting values one at a time. Suppose a new customer arrives with number 3, and we want to add them to the tree above, imagining that the 3 is not in it yet. The new node is shown at the top left in red. How do we know where it goes?

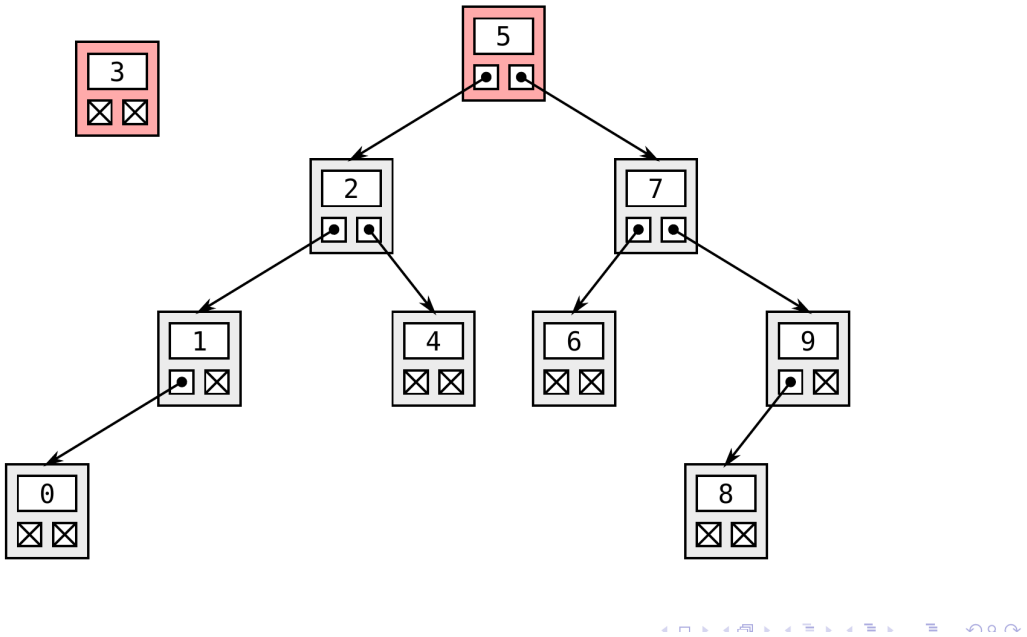


Figure 5: Inserting the value 3. The new node is at the top left. We start at the root, and compare at each node.

The rule is the same as for searching. At each node, if the new value is smaller, go left; if it is larger, go right. At the root: 3 is smaller than 5, so go left. At the 2: 3 is larger, so go right. At the 4: 3 is smaller, so go left. There is nothing there: the left pointer is `NULL`. This is where the new node belongs. We create a node (with `malloc`) holding 3, with both of its pointers set to `NULL`, and make the left pointer of the 4 point at it.

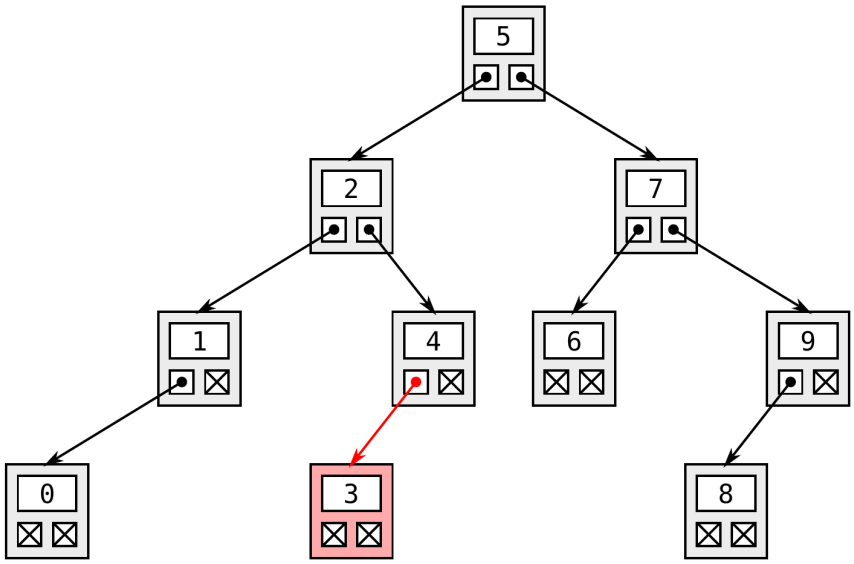


Figure 6: The tree after inserting 3. The red arrow is the pointer that was changed.

The new node is always added as a leaf. We never need to move existing nodes. Written as a recursive function, the insertion looks like this:

```
void insertNode(TreeNodePtr *treePtr, int value)
{
    if (*treePtr == NULL) {
        *treePtr = malloc(sizeof(TreeNode));

        if (*treePtr != NULL) {
            (*treePtr)->data = value;
            (*treePtr)->leftPtr = NULL;
            (*treePtr)->rightPtr = NULL;
        }
    }
    else if (value < (*treePtr)->data) {
        insertNode(&((*treePtr)->leftPtr), value);
    }
    else if (value > (*treePtr)->data) {
        insertNode(&((*treePtr)->rightPtr), value);
    }
}
```

The first parameter is a pointer to a `TreeNodePtr`, the double-pointer idea from the last lecture. The function must be able to change a pointer: either the root pointer (when the tree is empty) or the left or right pointer of a node (when we reach the empty spot). To change a pointer inside

a function, we pass the address of that pointer, so inside the function `*treePtr` is the pointer that we may need to change. The recursive calls pass the address of the left or right pointer of the current node.

Starting with an empty tree, `TreeNodePtr root = NULL;` and calling `insertNode(&root, 5)`, then 2, 7, 1, 4, 6 and 9, builds the tree that we will use for the rest of this lecture.

Traversing a tree

Often you want to visit every node in the tree: perhaps to print every value in it. There is more than one order in which this can be done, and each order is a short recipe. There are three of them, called *inorder*, *preorder* and *postorder*:

- **Inorder.** At each node: go to the left subtree, then process the node's value, then go to the right subtree.
- **Preorder.** At each node: process the node's value, then go to the left subtree, then go to the right subtree.
- **Postorder.** At each node: go to the left subtree, then go to the right subtree, then process the node's value.

The only difference is when the node's own value is processed: before (pre), between (in) or after (post) the two subtrees. "Process" here means whatever we want to do with the value, such as printing it. These recipes are easy to implement with recursive functions, as we will see. They are also easy to confuse, so learn the three of them.

We will use the following tree for all three.

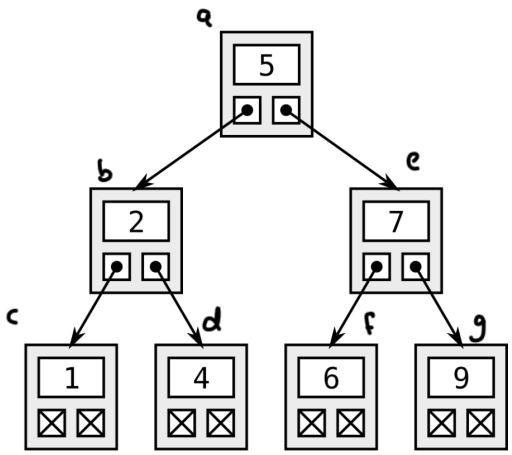


Figure 7: The tree used in the traversal examples.

Inorder traversal

Here is the code for inorder traversal:

```
void inOrder(TreeNodePtr treePtr)
{
    if (treePtr != NULL) {
        inOrder(treePtr->leftPtr);
        printf("%3d", treePtr->data);
        inOrder(treePtr->rightPtr);
    }
}
```

The three lines inside the `if` are the three steps of the recipe. The function is called in `main` on the root of the tree, `inOrder(root);`.

Before looking at how the code works, let us do the recipe by hand. Label the nodes a to g, with a (5) the root, b (2) and e (7) its children, and c (1), d (4), f (6) and g (9) the leaves. At each node we do three steps: step 1 is go left, step 2 is print, step 3 is go right.

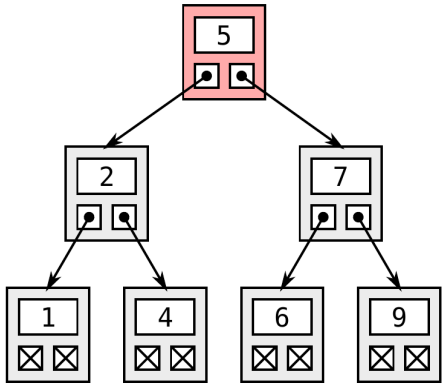
We start at a. Step 1 is go left, so we are at b. At b, step 1 is again to go left, so we move to c. At c, step 1 is go left, but there is nothing there, so nothing happens. Step 2 is print: the screen shows 1. Step 3 is go right, and again there is nothing. We have finished c.

So we go back up to b, where we had done step 1. Step 2 at b: print 2. Step 3: go right, to d. At d there is nothing to the left, so we print 4 and there is nothing to the right. We finish d, and so we have finished b. Back up at a, step 1 is done, and step 2 is to print 5. Step 3 is to go right, to e, and the same thing happens in the right half: 6, then 7, then 9. The output is:

```
1 2 4 5 6 7 9
```

The figure below shows the code and its output for this tree.

```c
void inOrder( TreeNodePtr treePtr )
{
  if ( treePtr != NULL ) {
    inOrder( treePtr->leftPtr );
    printf( "%3d", treePtr->data );
    inOrder( treePtr->rightPtr );
  }
}
```



Output

```
1 2 4 5 6 7 9
```

Figure 8: Inorder traversal of the tree. The output is 1 2 4 5 6 7 9.

Notice that the values come out in ascending order. This is no accident: it follows from the sorting rule. At each node, everything smaller is on the left and gets printed first, then the node, then everything larger on the right. So inorder traversal of a sorted binary tree prints the values in sorted order. That is the reason for the name.

The instruction "go left" really means "go left, and do everything that there is to do on that side, all the way down, before coming back". This is what makes it confusing at first, and it is the right way to think about the recursion.

How the recursion works

The part of this code that people find strange is that `inOrder` calls itself, twice. How does the function know where to go back to? A way of thinking about this is to imagine that every time a function is called, it spawns a fresh copy of itself, with its own `treePtr`, and when that copy finishes, execution returns to the line after the call that created it. It is like the film *Inception*: a dream inside a dream inside a dream, and you wake up one level at a time.

Here is how that looks on our tree. In `main`, we call `inOrder` with the address of a. That starts a copy of the function with `treePtr` pointing at a.

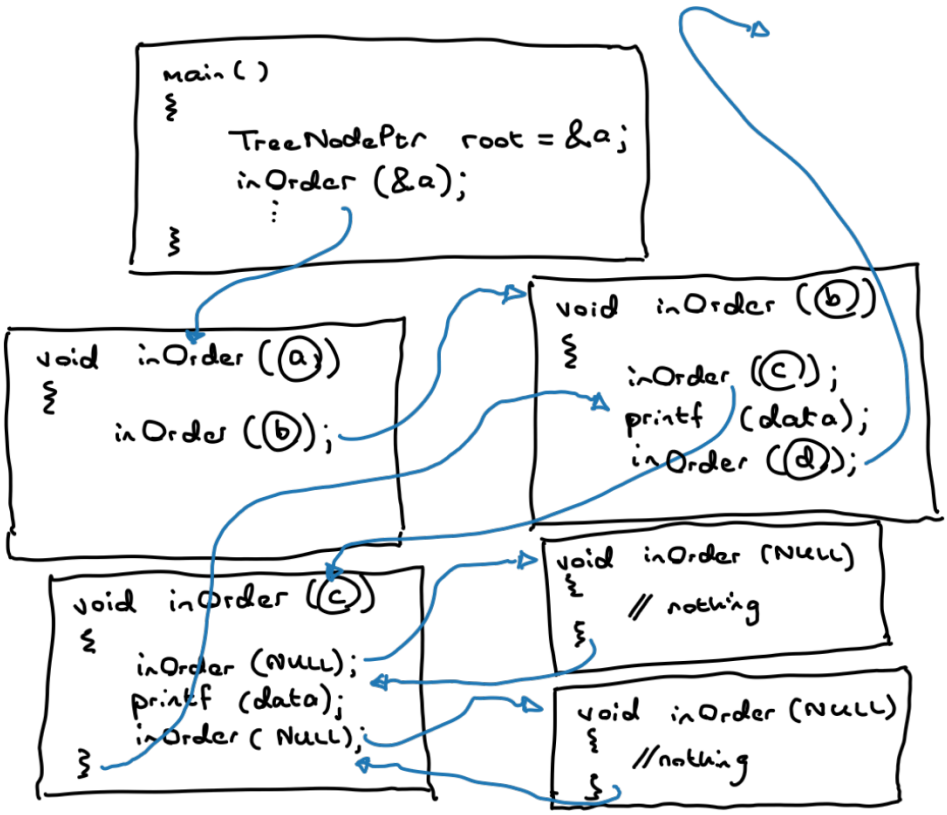


Figure 9: The calls made by `inOrder` as it works down the left side of the tree. Each box is a separate copy of the function. a calls b, b calls c, and c calls `inOrder` on its `NULL` children, which does nothing and returns.

In the copy for a, `treePtr` is not `NULL`, so we run the three lines. The first one calls `inOrder` on a's left pointer, which is b. This starts a new copy for b. In b's copy the first line calls `inOrder` on c, starting a copy for c. In c's copy, the first line calls `inOrder` on c's left pointer, which is `NULL`. That starts a copy that finds `treePtr` is `NULL`, so the `if` is false and nothing happens: the function just ends. It is the most boring dream there is.

When that copy ends, execution goes back to the line after the call that created it. We are in c's copy again, and we are at the second line: `printf` prints 1. Then the third line calls `inOrder` on c's right pointer, which is `NULL`: another boring copy that does nothing. Now c's copy reaches the closing brace and ends, and execution returns to the line after the call in b's copy, which is the `printf`. That prints 2. Then b calls `inOrder` on d, which prints 4, and returns; b's copy finishes; and we are back in a's copy, at its `printf`, which prints 5. The right side then works the same way.

That is how the function "knows" where to go back to: each copy resumes at the line after the one that called it. The empty subtrees (`NULL` pointers) are the *base case* of the recursion: they stop it from going on for ever. The check `treePtr != NULL` is also what protects the code from dereferencing a null pointer.

It is a lot to take in the first time. The way to make sense of it is to draw it for yourself, with a small tree of three or four nodes.

Preorder traversal

In preorder traversal, the node's value is processed first. The code differs from inorder in one thing: the order of the three lines.

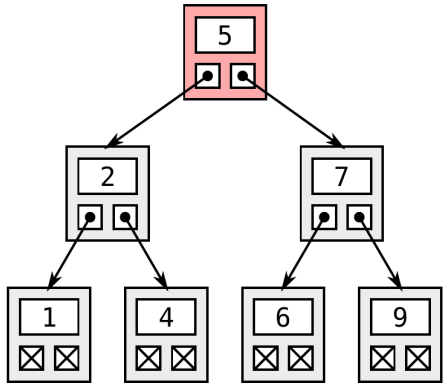
```
void preOrder(TreeNodePtr treePtr)
{
    if (treePtr != NULL) {
        printf("%3d", treePtr->data);
        preOrder(treePtr->leftPtr);
        preOrder(treePtr->rightPtr);
    }
}
```

We start at the root and print 5. Then we go left to the 2 and print it. Then left again to the 1: print 1. Both of its children are `NULL`, so we are done with it and go back up to the 2, which has been printed and has gone left, so it now goes right: print 4. Back at 5, which has printed and has gone left, we go right to the 7: print 7, then go left to print 6, and right to print 9. The output is:

```
5 2 1 4 7 6 9
```

Which order you want depends on the application. Preorder visits every node before any of its descendants, so it is the natural order for, say, copying a tree: you need to create a parent before you can attach its children.

```
void preOrder( TreeNodePtr treePtr )
{
  if ( treePtr != NULL ) {
    printf( "%3d", treePtr->data );
    preOrder( treePtr->leftPtr );
    preOrder( treePtr->rightPtr );
  }
}
```




```
Output
 5 2 1 4 7 6 9
```

Figure 10: Preorder traversal of the tree. The output is 5 2 1 4 7 6 9.

Postorder traversal

In postorder traversal, both subtrees are done before the node is processed:

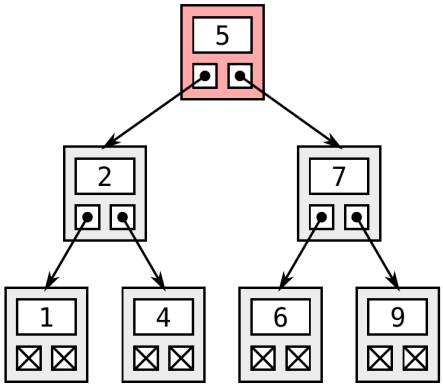
```
void postOrder(TreeNodePtr treePtr)
{
    if (treePtr != NULL) {
        postOrder(treePtr->leftPtr);
        postOrder(treePtr->rightPtr);
        printf("%3d", treePtr->data);
    }
}
```

From the root we go left to the 2, then left to the 1. There is nothing to the left or right of it, so we print 1. We are back at the 2, which has gone left but not yet right: go right to 4, which prints 4. Now the 2 has done both sides, so it prints 2. Then we are back at the 5, which has been all the way left and must now go right, and the same happens: 6, then 9, then 7. The value of the root is printed last of all, because we had to go all the way left and all the way right first. The output is:

```
1 4 2 6 9 7 5
```

Postorder is the natural order when you need to deal with the children before the parent: for example, to free a whole tree, you have to free the children before you free the parent, or you lose the pointers to them.

```
void postOrder( TreeNodePtr treePtr )
{
  if ( treePtr != NULL ) {
    postOrder( treePtr->leftPtr );
    postOrder( treePtr->rightPtr );
    printf( "%3d", treePtr->data );
  }
}
```




```
Output
 1 4 2 6 9 7 5
```

Figure 11: Postorder traversal of the tree. The output is 1 4 2 6 9 7 5. The root is always the last value to be printed.

The shape of a tree

Is the root always the middle of the sorted values? No. The tree we have used happens to be *balanced*: each level is full, and it is about as short as a tree with these values can be. But the shape of the tree depends entirely on the order in which the values were inserted.

If you insert the values 1, 2, 3, 4, 5 in ascending order, then each new value is larger than everything in the tree, so it goes to the right every time. You end up with a tree that has only right children, which is a linked list in disguise, and searching it is no faster than searching a list. This is called an *unbalanced* tree. A tree with n nodes that is balanced has about $\log_2 n$ levels, so a search needs only that many comparisons; an unbalanced tree can have n levels, and a search can take n comparisons. (There are algorithms that keep a tree balanced automatically as values are inserted, but that is beyond this course.)

A search in a sorted binary tree is similar to binary search in a sorted array: at each step, you compare with the middle and throw away half. The difference is the data structure: binary search needs an array, with its fixed size and with the problem that inserting a value means moving everything after it. A tree grows and shrinks with the data and finds its elements in the same way.

Where trees are used

Trees are used in many places, such as file systems, the compiler's representation of an expression, and in searching and sorting. Here are three examples from machine learning, where one of the most popular models before neural networks took over was the *decision tree*.

A classic data set in machine learning is about irises, a type of flower. For each flower we have the length and width of its petals and the length and width of its sepals (the outer leaves),

in centimetres, as in the figure below. The task is to write a program that, given these four measurements, tells you which type of iris it is.

Figure 12: An iris. The petal and sepal are marked, with the length and width measured for each.

One way is a decision tree, just like the fruit tree that we started with. Each node asks a question about one of the measurements. The tree in the figure below was not written by hand; a machine learning algorithm worked out which questions to ask, and in what order, from a set of labelled flowers.

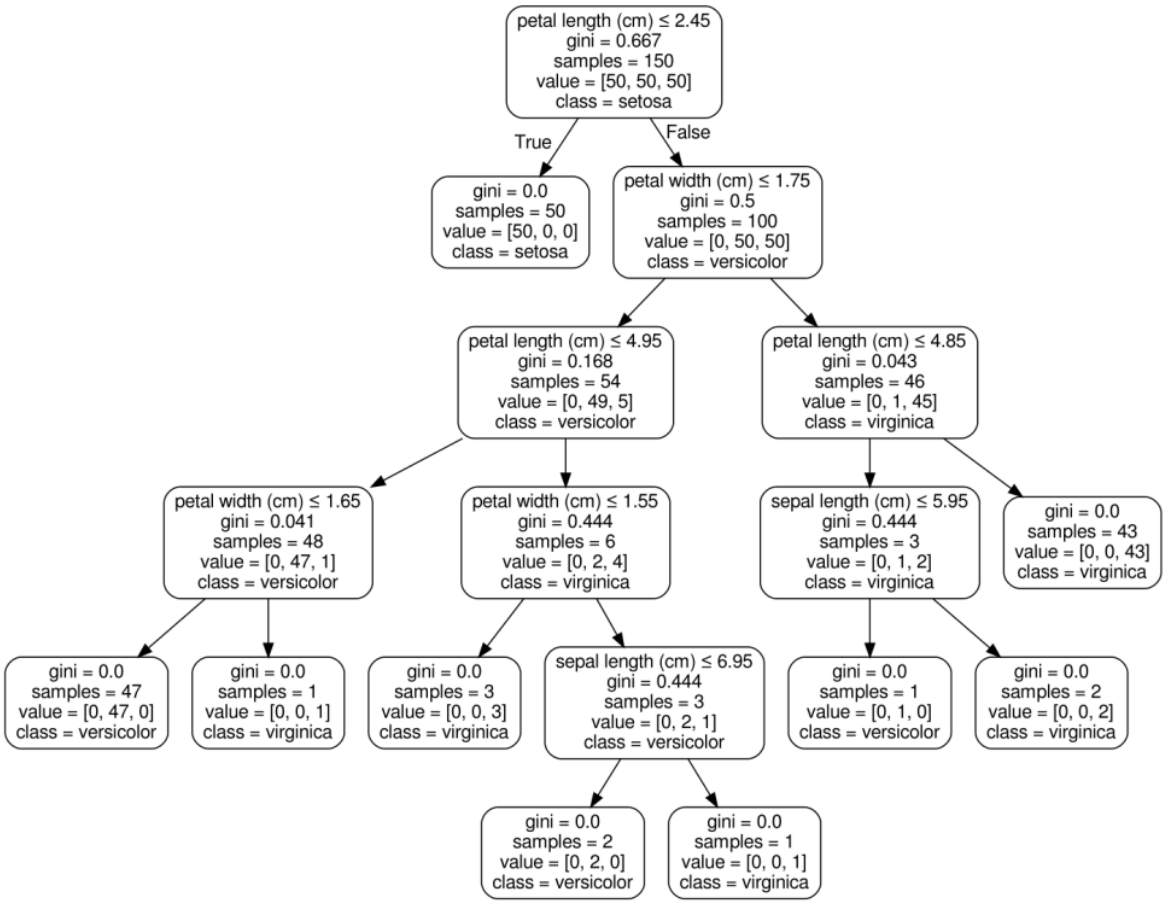


Figure 13: A decision tree for classifying irises. At the top: is the petal length at most 2.45 cm? If true, it is a setosa and we are done. If false, the next question is about the petal width, and so on, down to a leaf that names the type of iris.

The second example is speech recognition for different accents of South African English. In this case a decision tree was used to decide, for each speech sound, whether data from the different accents could be shared, or whether a separate model was needed for each accent. The questions ask about the accent and about the sounds on either side of the one being modelled.

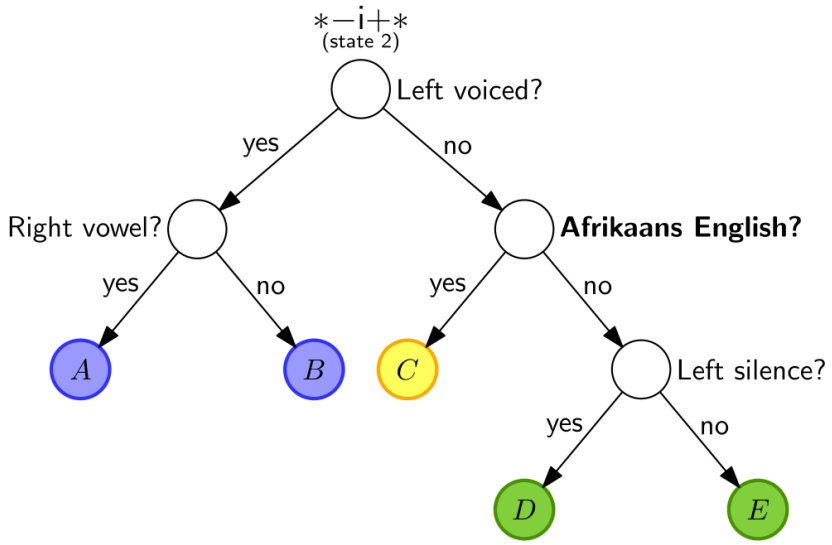


Figure 14: A decision tree used for speech recognition with several accents. The questions ask, for example, whether the sound on the left is voiced, or whether the accent is Afrikaans English. This tree is not balanced.

In all of these, the tree is the same structure that we built in this lecture: nodes with a left and a right pointer, and a rule for which way to go at each node.

Summary

A tree is a dynamic data structure in which each node can point at more than one other node. In a binary tree each node has a left and a right pointer, and the top of the tree is the root node. The children of a node are the nodes its pointers point at, and a node and everything below it is a subtree. A leaf has `NULL` for both pointers.

In a sorted binary tree, every value in a node's left subtree is smaller than the node's value, and every value in the right subtree is larger. This lets us search by following one path from the root, and insert by finding the empty place at the end of that path and attaching a new node there.

There are three ways to visit all the nodes of a tree, each a three-step recursive recipe:

- Inorder: left, node, right (a sorted tree is visited in ascending order)
- Preorder: node, left, right
- Postorder: left, right, node

Each of these is a short recursive function whose base case is a `NULL` pointer. Trees bring together structures, pointers, `malloc` and recursion, and the shape of the tree depends on the order of insertion, which determines how quickly it can be searched.

Exercises

1. Draw the sorted binary tree that results from inserting the values 8, 3, 10, 1, 6, 14, 4, 7, 13 in that order into an empty tree.
2. For the tree in the previous exercise, write the output of an inorder, a preorder and a postorder traversal.
3. Insert 1, 2, 3, 4, 5 into an empty tree in that order. What does it look like? What would you change about the order to make a balanced tree?
4. Write down the three tree-traversal recipes from memory. Which of them prints a sorted tree in ascending order?
5. Why does `insertNode` take a `TreeNodePtr *` rather than a `TreeNodePtr`?
6. Write a function `countNodes` that returns the number of nodes in a tree, using recursion.
7. Write a function `treeHeight` that returns the number of levels in a tree. Use the result `0` for an empty tree.
8. Write a function that frees every node of a tree. Which of the three traversal orders do you need, and why?