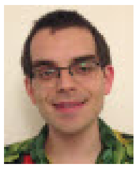
Binary Trees in C


By Alex Allain

The binary tree is a fundamental data structure used in computer science. The binary tree is a useful data structure for rapidly storing sorted data and rapidly retrieving stored data. A binary tree is composed of parent nodes, or leaves, each of which stores data and also links to up to two other child nodes (leaves) which can be visualized spatially as below the first node with one placed to the left and with one placed to the right. It is the relationship between the leaves linked to and the linking leaf, also known as the parent node, which makes the binary tree such an efficient data structure. It is the leaf on the left which has a lesser key value (i.e., the value used to search for a leaf in the tree), and it is the leaf on the right which has an equal or greater key value. As a result, the leaves on the farthest left of the tree have the lowest values, whereas the leaves on the right of the tree have the greatest values. More importantly, as each leaf connects to two other leaves, it is the beginning of a new, smaller, binary tree. Due to this nature, it is possible to easily access and insert data in a binary tree using search and insert functions recursively called on successive leaves.

The typical graphical representation of a binary tree is essentially that of an upside down tree. It begins with a root node, which contains the original key value. The root node has two child nodes; each child node might have its own child nodes. Ideally, the tree would be structured so that it is a perfectly balanced tree, with each node having the same number of child nodes to its left and to its right. A perfectly balanced tree allows for the fastest average insertion of data or retrieval of data. The worst case scenario is a tree in which each node only has one child node, so it becomes as if it were a linked list in terms of speed. The typical representation of a binary tree looks like the following:

```
        10
      /    \
     6      14
    / \    /  \
   5   8  11  18
```

The node storing the 10, represented here merely as 10, is the root node, linking to the left and right child nodes, with the left node storing a lower value than the parent node, and the node on the right storing a greater value than the parent node. Notice that if one removed the root node and the right child nodes, that the node storing the value 6 would be the equivalent a new, smaller, binary tree.
The structure of a binary tree makes the insertion and search functions simple to implement using recursion. In fact, the two insertion and search functions are also both very similar. To insert data into a binary tree involves a function searching for an unused node in the proper position in the tree in which to insert the key value. The insert function is generally a recursive function that continues moving down the levels of a binary tree until there is an unused leaf in a position which follows the rules of placing nodes. The rules are that a lower value should be to the left of the node, and a greater or equal value should be to the right. Following the rules, an insert function should check each node to see if it is empty, if so, it would insert the data to be stored along with the key value (in most implementations, an empty node will simply be a NULL pointer from a parent node, so the function would also have to create the node). If the node is filled already, the insert function should check to see if the key value to be inserted is less than the key value of the current node, and if so, the insert function should be recursively called on the left child node, or if the key value to be inserted is greater than or equal to the key value of the current node the insert function should be recursively called on the right child node. The search function works along a similar fashion. It should check to see if the key value of the current node is the value to be searched. If not, it should check to see if the value to be searched for is less than the value of the node, in which case it should be recursively called on the left child node, or if it is greater than the value of the node, it should be recursively called on the right child node. Of course, it is also necessary to check to ensure that the left or right child node actually exists before calling the function on the node.
Because binary trees have log (base 2) n layers, the average search time for a binary tree is log (base 2) n. To fill an entire binary tree, sorted, takes roughly log (base 2) n * n. Let's take a look at the necessary code for a simple implementation of a binary tree. First, it is necessary to have a struct, or class, defined as a node.

```
struct node
{
  int key_value;
  struct node *left;
  struct node *right;
};
```

The struct has the ability to store the key_value and contains the two child nodes which define the node as part of a tree. In fact, the node itself is very similar to the node in a linked list. A basic knowledge of the code for a linked list will be very helpful in understanding the techniques of binary trees. Essentially, pointers are necessary to allow the arbitrary creation of new nodes in the tree.

There are several important operations on binary trees, including inserting elements, searching for elements, removing elements, and deleting the tree. We'll look at three of those four operations in this tutorial, leaving removing elements for later.

We'll also need to keep track of the root node of the binary tree, which will give us access to the rest of the data:

```
struct node *root = 0;
```

It is necessary to initialize root to 0 for the other functions to be able to recognize that the tree does not yet exist. The destroy_tree shown below which will actually free all of the nodes of in the tree stored under the node leaf: tree.

```
void destroy_tree(struct node *leaf)
{
  if( leaf != 0 )
  {
      destroy_tree(leaf->left);
      destroy_tree(leaf->right);
      free( leaf );
  }
}
```

The function destroy_tree goes to the bottom of each part of the tree, that is, searching while there is a non-null node, deletes that leaf, and then it works its way back up. The function deletes the leftmost node, then the right child node from the leftmost node's parent node, then it deletes the parent node, then works its way back to deleting the other child node of the parent of the node it just deleted, and it continues this deletion working its way up to the node of the tree upon which delete_tree was originally called. In the example tree above, the order of deletion of nodes would be 5 8 6 11 18 14 10. Note that it is necessary to delete all the child nodes to avoid wasting memory.

The following insert function will create a new tree if necessary; it relies on pointers to pointers in order to handle the case of a non-existent tree (the root pointing to NULL). In particular, by taking a pointer to a pointer, it is possible to allocate memory if the root pointer is NULL.

```
insert(int key, struct node **leaf)
{
    if( *leaf == 0 )
    {
        *leaf = (struct node*) malloc( sizeof( struct node ) );
        (*leaf)->key_value = key;
        /* initialize the children to null */
        (*leaf)->left = 0;
        (*leaf)->right = 0;
    }
    else if(key < (*leaf)->key_value)
    {
        insert( key, &(*leaf)->left );
    }
    else if(key > (*leaf)->key_value)
    {
        insert( key, &(*leaf)->right );
    }
}
```

The insert function searches, moving down the tree of children nodes, following the prescribed rules, left for a lower value to be inserted and right for a greater value, until it reaches a NULL node--an empty node--which it allocates memory for and initializes with the key value while setting the new node's child node pointers to NULL. After creating the new node, the insert function will no longer call itself. Note, also, that if the element is already in the tree, it will not be added twice.

```
struct node *search(int key, struct node *leaf)
{
  if( leaf != 0 )
  {
      if(key==leaf->key_value)
      {
          return leaf;
      }
      else if(key<leaf->key_value)
      {
          return search(key, leaf->left);
      }
```

```
        else
        {
            return search(key, leaf->right);
        }
    }
    else return 0;
}
```

The search function shown above recursively moves down the tree until it either reaches a node with a key value equal to the value for which the function is searching or until the function reaches an uninitialized node, meaning that the value being searched for is not stored in the binary tree. It returns a pointer to the node to the previous instance of the function which called it.

Still not getting it? Ask an expert!
Previous: Variable argument lists to functions
Back to C Tutorial Index

Related Articles

Learn how binary trees are used in practice to compress data using the Huffman Encoding Algorithm