Advanced C Programming

Trees

1

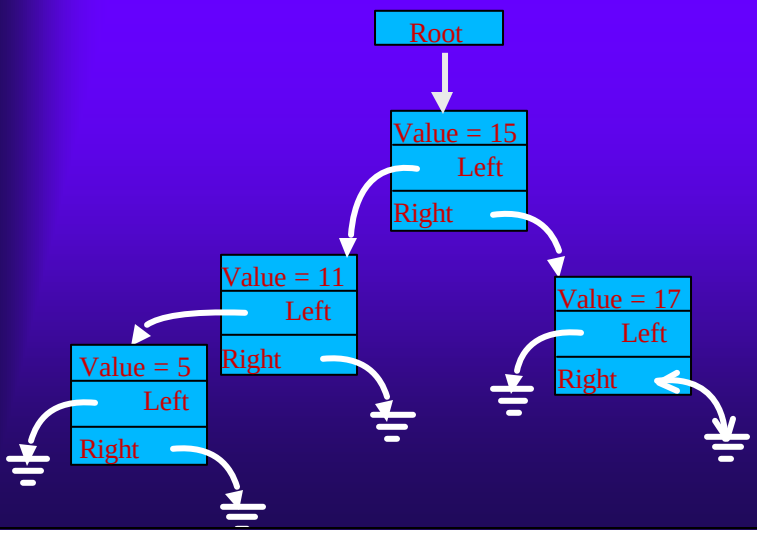
Trees

- Until now, we've only looked at lists that have only one "dimension": forward/backward or next/previous.
- Consider a structure that acts as a "parent" and has at most two "children" (a binary tree)

```
struct Node {
  int Value;
  struct Node *Left;
  struct Node *Right;
};
```

2

What does a tree look like? (Lots of children)



3

Interesting properties of trees

- Trees are fun to use because you can easily add more children to the existing children.
- With the trees we're working with, the left child always has a Value less than or equal to the parent's Value. The right child always has a Value greater than the parent's Value.
- You can always add a new child in the proper position (to the left or right of the parent).
- The tree is always fully sorted (how)?
- The tree is easily searchable.

4

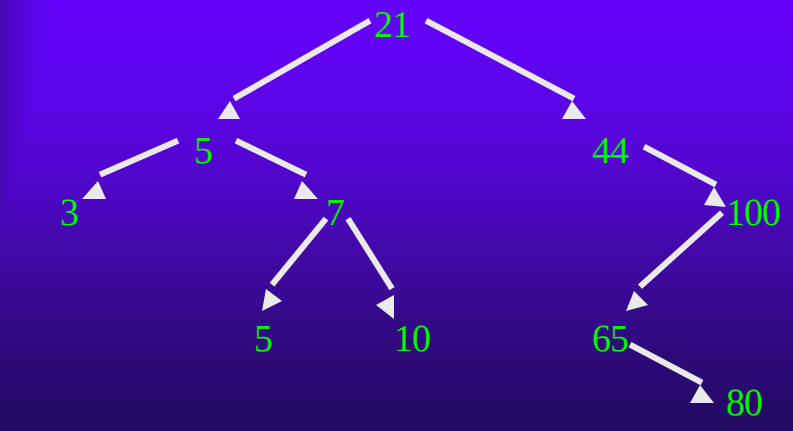
Creating an ordered binary tree

- Creating a binary ordered tree from a random data set: (also called binary search tree)
- First element stored in the root node
- Compare the second element with the root node, if the new value is less than the parent's value, then move left (e.g. Pointer = Pointer->Left). Otherwise go right.
- Keep on comparing new value and keep on moving (left or right) until you reach a NULL pointer. Append the new node at that location.

5

Creating an ordered binary tree

21 44 5 7 3 10 100 5 65 80



6

More about trees

- Searching an ordered binary tree is just as easy as inserting something in a tree:
  1. Set a Pointer to point at the root structure.
  2. If the value we're looking for == the Pointer value, return the Pointer.
  3. If the value we're looking for is < the Pointer value, go left. (e.g. Pointer = Pointer->Left) And goto (2)
  4. Otherwise go right and goto (2)
  5. If the Pointer is ever NULL, return NULL to indicate that the value was not found in the tree.

7

Tree processing and more trees

- Recursive methods for creating, searching, and traversing trees.
- Higher order trees. A node may have more than two children.
- For example: In a tertiary tree, maximum three children nodes per node

8

Recursive techniques for trees

- Most of the tree functions can be implemented using recursion.
- The code is easily readable and understandable.

9

Tree Functions (create)

```
struct Node *Create_Node(int Value)
{
  struct Node *Ptr = NULL;

  Ptr = malloc(sizeof(struct Node));
  assert(Ptr != NULL);

  Ptr->Left = NULL;
  Ptr->Right = NULL;
  Ptr->Value = Value;
}
```

10

Tree functions (Insert) (Iterative Version)

```
void Insert_Node(struct Node *Root,   struct Node *New)
{   while(1) {
    if (New->Value <= Root->Value)
      if (Root->Left == NULL)
      {         Root->Left = New;
          return;
      }
      else     Root = Root->Left;
    else
      if (Root->Right == NULL)
       {   Root->Right = New;
           return;
       }
      else   Root = Root->Right;
  }
}
```

11

Tree Functions (Insert) (Recursive Version)

```
void Insert_Node(struct Node *Root,    struct Node *New)
{
    if (New->Value <= Root->Value)
      if (Root->Left == NULL) {
        Root->Left = New; return;
      }
      else
        Insert_Node(Root->Left, New);
    else
      if (Root->Right == NULL) {
        Root->Right = New; return;
      }
      else
        Insert_Node(Root->Right, New);
}
```

12

More about trees

- Searching an ordered binary tree is just as easy as inserting something in a tree:
  1. Set a Pointer to point at the root structure.
  2. If the value we're looking for == the Pointer value, return the Pointer.
  3. If the value we're looking for is < the Pointer value, go left. (e.g. Pointer = Pointer->Left) And goto (2)
  4. Otherwise go right and goto (2)
  5. If the Pointer is ever NULL, return NULL to indicate that the value was not found in the tree.

13

Recursive version of Tree_Find

```
struct Node *Tree_Find(
  struct Node *Root,
  int Value)
{
  if (Root == NULL)
    return NULL;  /* Not found */
  if (Value == Root->Value)
    return Root;  /* Found it */
  if (Value < Root->Value)  /* Go left */
    return Tree_Find(Root->Left, Value);

  return Tree_Find(Root->Right, Value);
}
```

14

How do we get at the sorted content of a tree?

- We know that an ordered binary tree is fully sorted. We'd like to take advantage of that.
- The "least" element in the tree is at the far left.
- The "greatest" element is at the far right.
- Our tree nodes do not point back to their parents.
- How can we start at the far left and go through each node in order???

15

Tree Traversal

- Accessing each of the nodes of a tree in order is often called Tree Traversal or Iterating over a Tree. We can do this in several ways.
- Least to greatest:
  For each node, access the left node recursively, then the node itself, then the right node recursively. (Abbreviated L-N-R)
- Greatest to least: Same way except R-N-L.
- Prefix: N-L-R
- Postfix: L-R-N

16

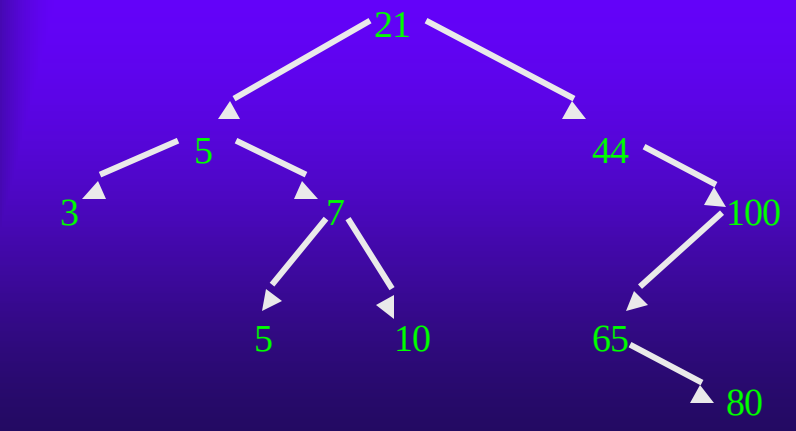
Example of ordered printing

```
void Print_Tree(struct Node *Ptr)
{
  if (Ptr == NULL)
    return;

  Print_Tree(Ptr->Left);  /* Go left */

  printf("%d\n", Ptr->Value); /* Node */

  Print_Tree(Ptr->Right); /* Go right */
}
```

17

Example of tree traversal

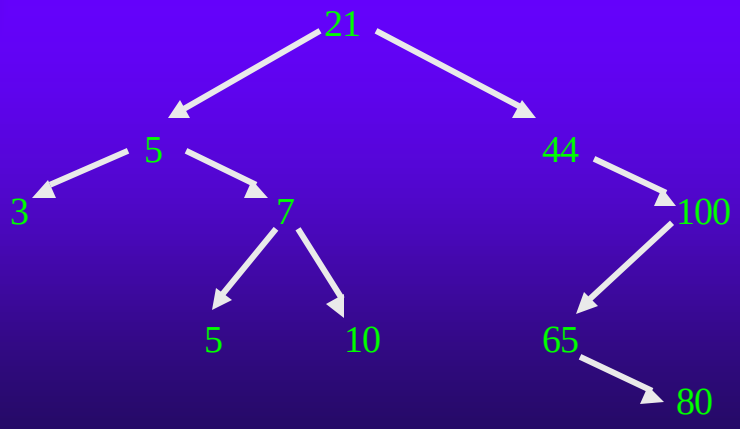
21 44 5 7 3 10 100 5 65 80



18

Example of tree traversal

LNR output: 3 5 5 7 10 21 44 65 80 100



RNL output: 100 80 65 44 21 10 7 5 5 3

19