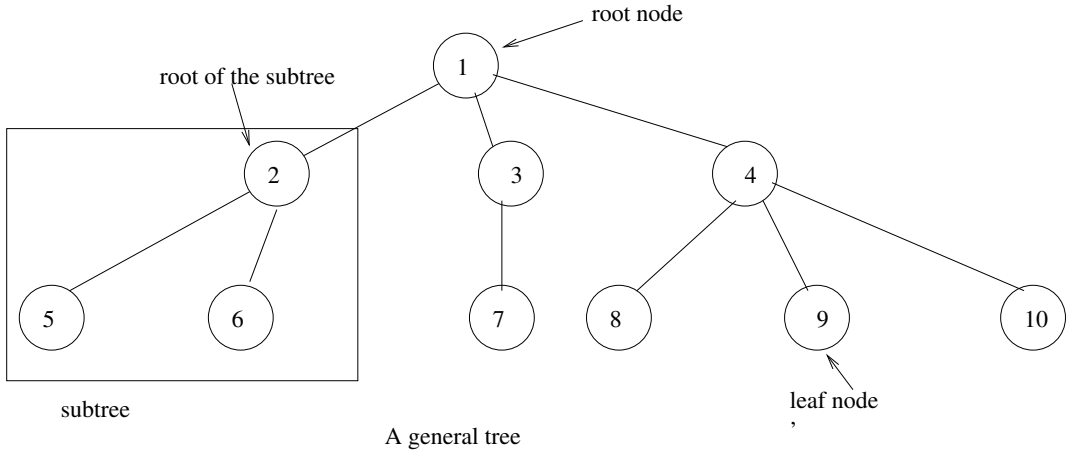
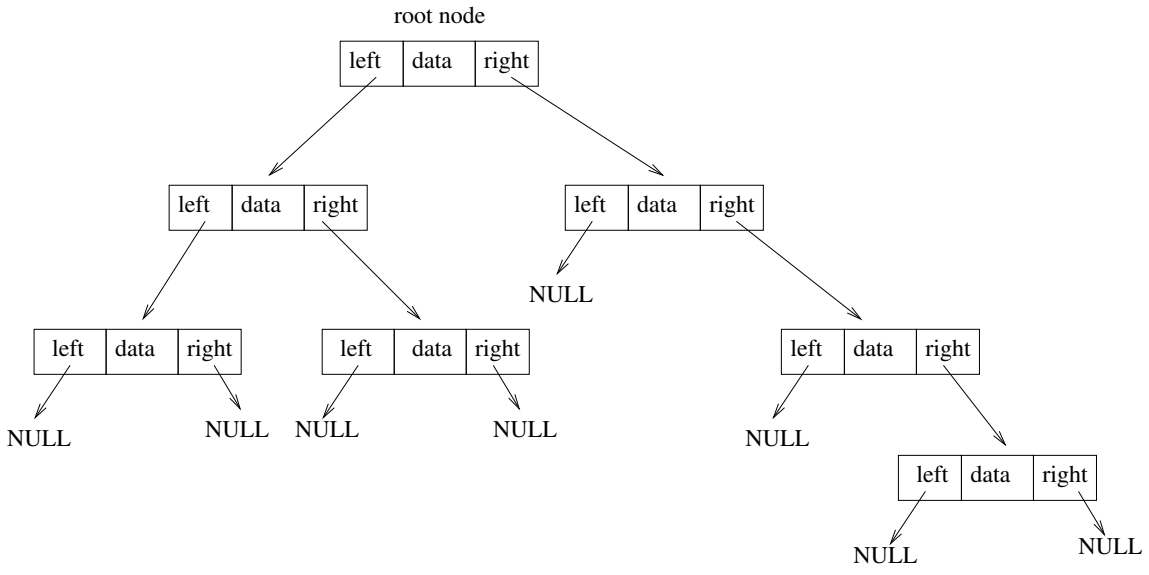
Binary trees

A tree consists of a finite set of elements called nodes. A tree has a unique node, called the root node, and the remaining nodes are a disjoint collection of subtrees. If a node r has T_1, T_2,$\cdots$, T_n as subtrees, then r_1, r_2, $\cdots$, r_n, the roots of these subtrees, are the children of r. A node with no children is called a leaf node. The diagram below shows a general tree.



A binary tree is a tree whose elements have at most two children. A binary tree, as a data structure, is an object composed of elements that have two link fields, called the left child and the right child. Each link must point at a new object not already pointed at or be NULL. In this representation, a leaf node has both its left and right children set to NULL.



A binary tree

To describe a binary tree, we proceed as follows. Each node consists of three components: data, left link, and right link. The data could be a structure or a native variable type. The left and right links are pointer variables to the node. If we assume that the data is an integer, then a node is defined as follows:

```
typedef int data;
typedef struct node
{
data d;
struct node *left,*right;
} node;
```

We also define

```
typedef node *BTREE;
```

Hence, if we write `BTREE root`; then `root` is an address variable of the type `node`.

Note that a binary tree has a repetitive structure. The root node has two children: a left child and a right child. The left child is itself a binary subtree, and so is the right child. Thus, a function that processes the root node behaves the same way as one that starts at the left child node, and so on. Thus, recursive functions are best suited for a binary tree.

Binary tree traversal: There is only one way to traverse a linear list, namely from head to tail. However, there are several natural ways to visit the elements of a binary tree. The three commonest are

1. **Inorder traversal**: We traverse a tree such that for each node, we first traverse its left subtree, then process the node itself, and finally traverse its right subtree.
2. **Preorder traversal**: We traverse a tree such that for each node, we first process the node itself, then traverse its left subtree, and finally traverse its right subtree.
3. **Postorder traversal**: We traverse a tree such that for each node, we first traverse its left subtree, then its right subtree, and finally process the node itself.

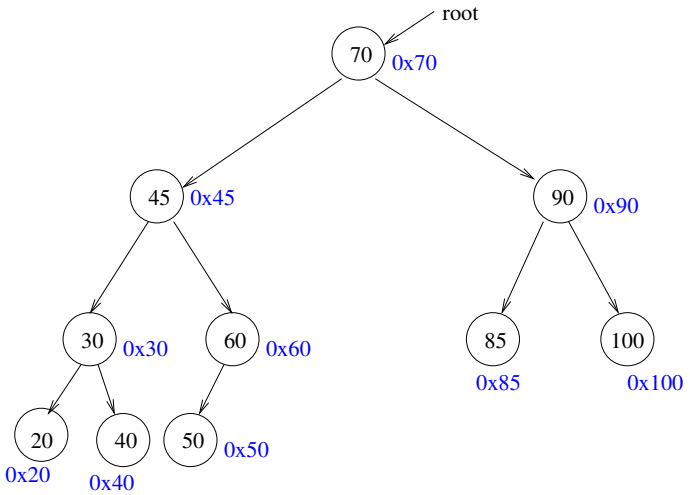
It is easy to write recursive functions that implement the three traversals described here. Suppose our job is to print the data (integers) in the nodes of a binary tree. The following implements the three traversals. For each function, we have sent the root node's address. Thus, initially, `t` (in each of the functions) has the address of the root node.

```
// Inorder binary tree traversal
void inorder(BTREE t)
{
if(t != NULL)
 {
 inorder(t->left);
 printf("%d  ",t->d);
 inorder(t->right);
```

```
 }
}

// Preorder binary tree traversal
void preorder(BTREE t)
{
if(t != NULL)
 {
 printf("%d  ",t->d);
 preorder(t->left);
 preorder(t->right);
 }
}

// Postorder binary tree traversal
void postorder(BTREE t)
{
if(t != NULL)
 {
 postorder(t->left);
 postorder(t->right);
 printf("%d  ",t->d);
 }
}
```



A binary tree with hypothetical addresses (in blue font) of the nodes.

Consider the binary tree shown in the picture. The function `inorder()` produces the following output:

```
20   30   40   45   50   60   70   85   90   100
```

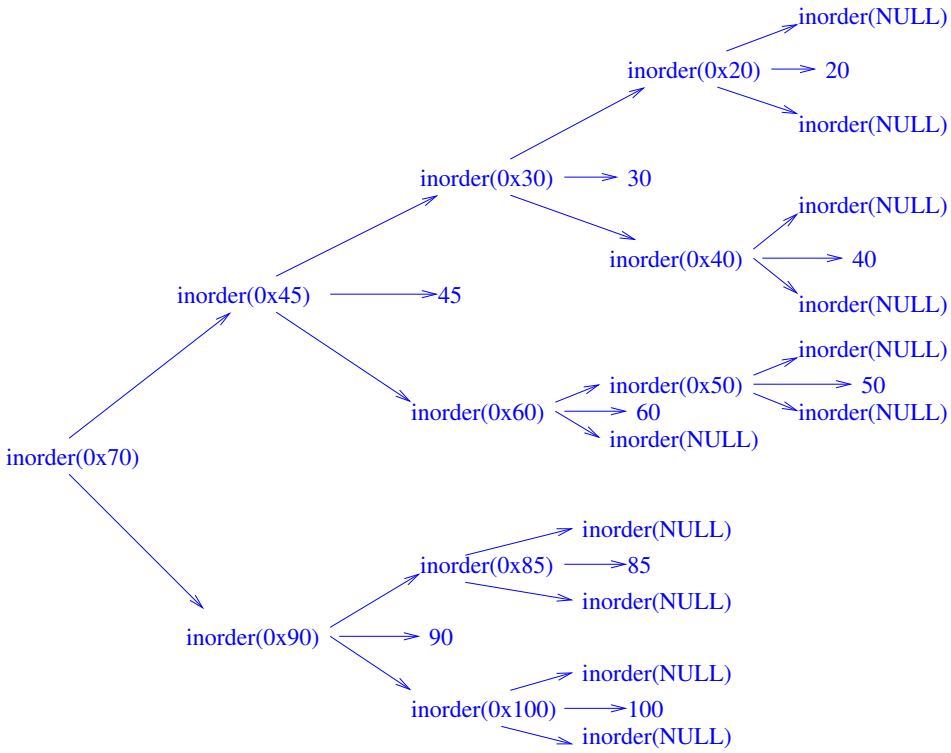
On the other hand, preorder traversal leads to the following output:

```
70  45  30  20  40  60  50  90  85  100
```

Similarly, the postorder traversal generates the following output:

```
20  40  30  50  60  45  85  100  90  70
```

Let us analyze the inorder traversal. It is called with `inorder(0x70)`. The diagram below explains how the function `inorder()` works.



Inorder traversal in a binary tree

Note that three statements are associated with every non-NULL node. Since `0x70` is non-NULL, it executes the first statement `inorder(0x45)`. Once this execution finishes, then it will execute the next two statements. Since `0x45` is non-NULL, three statements are associated with it. The first statement executed is `inorder(0x30)`. Again, because it is non-NULL, the first statement in the function`inorder(0x30)` is `inorder(0x20)`. Here, it is non-NULL again, so three statements are associated with it. The first statement calls its left child, which is NULL, and hence it does nothing. The next `printf()` statement prints `20` and the next statement calls its right child, which is NULL, and hence it does nothing. Thus, the first output is `20`. The first statement of the call associated with `inorder(0x30)` is now finished. The next statement prints `30` and the final statement calls `inorder(0x40)`. Thus, the second output is `30`. Next, we can trace `inorder(0x40)` using the arguments just described. In this way, we can explain the output of our initial call `inorder(0x70)`. The numbers printed in the output follow from the top of the diagram to the bottom.

Now we create a binary tree from the data values stored in an array `A[size]`. Note that the indices of the array run from `0` to `size-1`. There is a nice mapping from the indices of a

linear array into the nodes of a binary tree. We do this by taking the value `A[i]` and letting it have as child values `A[2*i+1]` and `A[2*i+2]`. Thus, we map `A[0]` into the unique root node of the resulting binary tree. Its left child will be `A[1]` and its right child will be `A[2]`. Similarly, left and right children of `A[1]` will be `A[3]` and `A[4]`. And, left and right child of `A[2]` will be `A[5]` and `A[6]`. This will continue until `A[size-1]`.

The function `cr_tr()` incorporates the above mapping. The function `cr_tr()` takes the data array `A`, an integer index `i`, and an integer `size` that denotes the size of the data array `A`. This function creates a node of the binary tree with data `A[i]` with links to its left and right children. It returns the address of the node with data `A[i]`. If we have `i>=size`, then the function `cr_tr()` returns `NULL`. To create links to its left child, we use `cr_tr()` function with the data array `A`, an integer index `2*i+1`, and the integer `size`. Similarly, for the right child, we use `cr_tr()` function with the data array `A`, an integer index `2*i+2`, and the integer `size`. The code below implements a program that creates a binary tree using the `cr_tr()` function and prints the node values using inorder traversal.

```
#include <stdio.h>
#include <stdlib.h>
typedef int data;
typedef struct node
{
data d;
struct node *left,*right;
} node;
typedef node *BTREE;
BTREE cr_tr(data [], int, int);
void inorder(BTREE);
int main()
{
 BTREE root;
 data A[]={70,45,90,30,60,85,100,20,40,50};
 root=cr_tr(A,0,10);           // array, i, size
 inorder(root);
 printf("\n");
 return 0;
}
BTREE cr_tr(data A[], int i, int size)
{
 BTREE t;
 if(i>=size)
```

```
 {
  return NULL;
 }
 else
 {
 t=(node *)calloc(1,sizeof(node));
 t->d=A[i];
 t->left=cr_tr(A,2*i+1,size);
 t->right=cr_tr(A,2*i+2,size);
 return t;
 }
}
void inorder(BTREE t)
{
if(t != NULL)
      {
 inorder(t->left);
 printf("%d  ",t->d);
 inorder(t->right);
       }
}
```

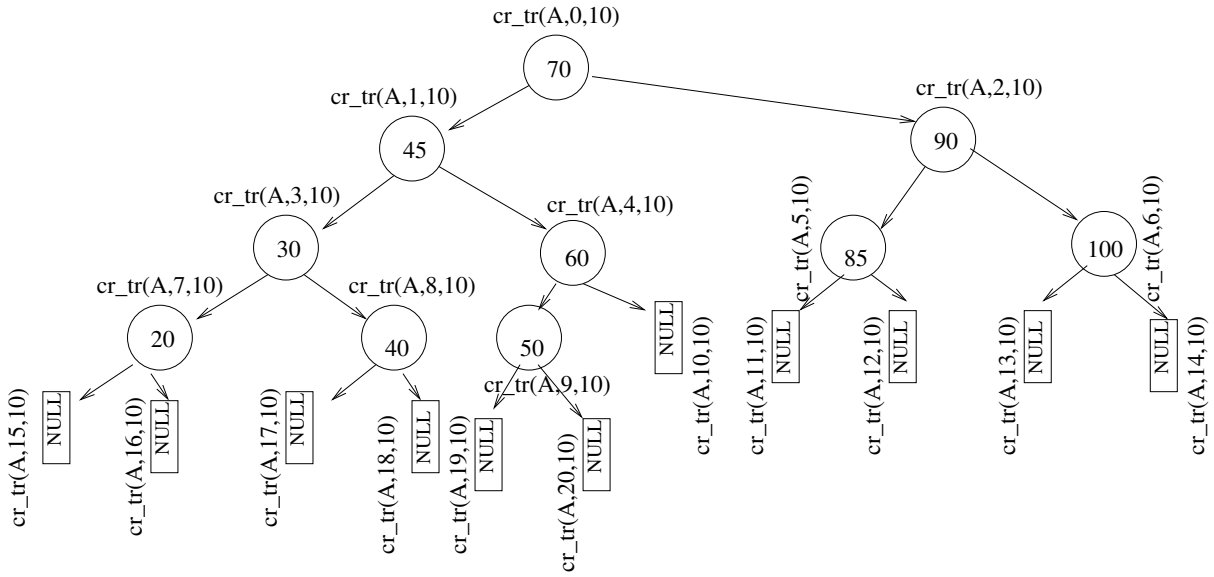
Listing 80: C Program "btree.c"

The binary tree constructed from the data array `A` is identical to the one shown earlier, with `70` as the root node. Hence, the output is as expected, shown below:

```
20  30  40  45  50  60  70  85  90  100
```

Note that `cr_tr(A,0,10)` is the first call to the function. Since `i=0<10=size`, the function `cr_tr()` first creates space to hold the first element `A[0]` of the data array and the links to the left and right children. This is the root node of the binary tree, whose address is returned to `main()` when execution transfers back to `main()`. After creating space, it assigns the data element `A[0]=70` and links to the left and right children. To assign the left link, we call `cr_tr(A,1,10)`. Once the call `cr_tr(A,1,10)` finishes, we call `cr_tr(A,2,10)` to assign a link to the right child. Note that the call `cr_tr(A,1,10)` creates a node with data `45` with links to its left and right children. Due to the recursive nature of the call, the call after `cr_tr(A,1,10)` is `cr_tr(A,3,10)`. Next call is `cr_tr(A,7,10)` (whose data value is `20`) followed by `cr_tr(A,15,10)`. Since `i=15>= 10=size`, the left child of the node with data `20` is `NULL`. The next call will be `cr_tr(A,16,10)` which also returns `NULL`. Thus, both the left

and right children of the node with data `20` are `NULL`. The next call `cr_tr(A,8,10)` will be executed now. The diagram below shows the trace of the function call `cr_tr(A,0,10)`.

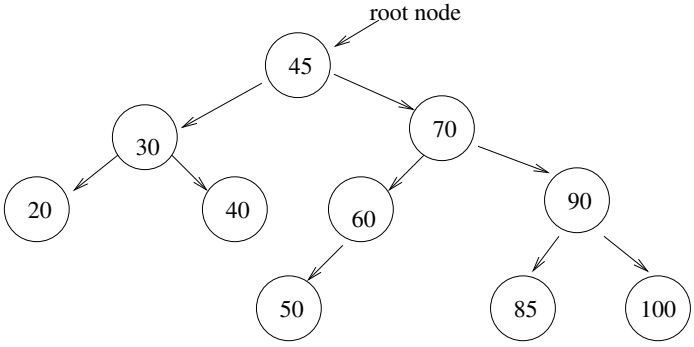


Starting from a node, we follow the left path. When the left path finishes, then we follow the right path from the node. Suppose we print the value of `i` of the function call `cr_tr(A,i,size)`. If the first call is `cr_tr(A,0,10)`, then the following will be printed

```
0 1 3 7 15 16 8 17 18 4 9 19 20 10 2 5 11 12 6 13 14
```

These numbers show the sequence of calls to the recursive function `cr_tr(A,i,size)`. Thus, the sequence of calls is `cr_tr(A,0,10)`, `cr_tr(A,1,10)`, `cr_tr(A,3,10)`, `cr_tr(A,7,10)`,.....

Note that due to the particular arrangement of integers in the data array `A`, every node's left child has a lower value, and every node's right child has a higher value. This type of binary tree is called the binary search tree. Starting from an arbitrary data array, we can create a binary search tree as follows. The first element `A[0]` is made the root node. The next element `A[1]` is compared with the element in the root node. If it is less, then we make it the left child; otherwise, we make it the right child. Now for the next data element `A[2]`, we compare with the root node. If it is less, then we compare with the left child and so on. For example, suppose the data array is `data B[]={45,70,90,30,60,85,100,20,40,50};` where we have interchanged the first two elements of the array `A` used in Listing 80. If we follow the rules just described, then we get the following binary search tree:



First element `45` is made the root node. The next element `70` is larger, and hence it is made the right child of the root. The third element `90` is compared with the root, and we travel

right since it is higher than the root. Now we compared with the right child `70` of the root. Since it is higher, we travel to the right of the node with value `70`. Since the right of `70` is empty, we made `100` the right child of the node with `70`. This process gives rise to the binary search tree shown in the diagram. This can be easily implemented in a program as well. However, the resulting binary search tree may not be well-balanced. Since a binary search tree has many useful applications, there are techniques by which a binary search tree can be made well-balanced. However, we will not pursue this further.

Return to Lecture Topics