Binary Tree in C: Linked Representation & Traversals

Sept. 14, 2018 LINKED LIST (/blog/tag/linked-list/?tag=linked-list) C (/blog/tag/c/?tag=c) C++ (/blog/tag/cpp/?tag=cpp) TREE (/blog/tag/tree/?tag=tree) BINARY TREE (/blog/tag/binary-tree/?tag=binary-tree) BINARY SEARCH TREE (/blog/tag/binary-search-tree/?tag=binary-search-tree) DATA STRUCUTRE (/blog/tag/data-strucutre/?tag=data-strucutre) 1997

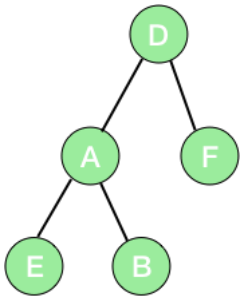
(https://play.google.com/store/apps/details?id=com.blogsdope&pcampaignid=MKT-Other-global-all-co-prtnr-py-PartBadge-Mar2515-1)

Previous:

- Trees in Computer Science (https://www.codesdope.com/blog/article/trees-in-computer-science)
- Binary Trees (https://www.codesdope.com/blog/article/binary-trees)

This post is about implementing a binary tree in C. You can visit Binary Trees (https://www.codesdope.com/blog/article/binary-trees) for the concepts behind binary trees. We will use linked representation to make a binary tree in C and then we will implement **inorder**, **preorder** and **postorder** traversals and then finish this post by **making a function to calculate the height of the tree**.

The binary tree (https://www.codesdope.com/blog/article/binary-trees) we will be using in this post is:



So, let's make a node using a structure in C.

```
struct node
{
    char data; //node will store character
    struct node *right_child; // right child
    struct node *left_child; // left child
};
```

`char data` – The data which we are going to store in this node is of character type.

`struct node *right_child` – 'right_child' is a pointer to the node (our defined structure). Thus, we will use this to link the current node with its right child.

`struct node *left_child` – Similarly, we will use this to link the current node with its left child.

Now, we have a structure of our node and we need a function to create new nodes to make a tree. So, let's write it.

```
struct node* new_node(char data)
{
    struct node *p; // node
    p = malloc(sizeof(struct node)); // allocating space to p
    p->data = data; // assinging data to p
    p->left_child = NULL; // making children NULL
    p->right_child = NULL;

    return(p); // returning the newly made node
}
```

`struct node *p` – '*p' is a node and 'p' is a pointer to the node. It means that p will store the address of the node. Please note that we have just declared a node here and no memory has been yet assigned to store data. Thus, the next task is to allocate the memory to the node and we will do it using the malloc function (https://www.codesdope.com/c-dynamic-memory/).

`p = malloc( sizeof (struct node))` – We are allocating a space in the memory to the node and storing the address of the memory in the variable 'p'.

`p->data = data` – Now, we have our node and we also have a space in memory allocated to it. Here, we are just storing data in it.

`p->left_child = NULL` – We are making the left child of the node null.

`p->right_child = NULL` – Similarly, we also made the right child null.

`return(p)` – We are just returning the address of the node we created.

Traversals in a Binary Tree

Now, we have made our node and a function to add new nodes. Thus, the next task is to make the tree described in the above picture and implement **inorder**, **postorder** and **preorder** traversals to it. So, let's first make the tree in the main function.

```
int main()
{
    struct node *root; //new structure
    root = new_node('D'); // making a new node
    /*
     ____
    |   D |
    |____|

    */
    root->left_child = new_node('A'); //left child of root

    /*
              ____
             |   D |
            /|____|
     ____ /
    |   A |
    |____|

    */

    root->right_child = new_node('F'); //right child of root

    /*
              ____
             |   D |
            /|____|\
     ____ /         _\___
    |   A |          |   F |
    |____|          |____|

    */

    root->left_child->left_child = new_node('E'); // new node

    /*
                       ____
                      |   D |
                     /|____|\
              ____ /         _\___
             |   A |          |   F |
            /|____|          |____|
      _____/
     |   E |
     |____|

    */

    root->left_child->right_child = new_node('B'); // new node

        /*
                       ____
                      |   D |
                     /|____|\
              ____ /         _\___
             |   A |          |   F |
            /|____|\         |____|
      _____/         _\___
     |   E |        |    B |
     |____|        |_____|

    */
}
```

You can learn the concepts behind the traversals from the post Binary Trees (https://www.codesdope.com/blog/article/binary-trees).

Preorder Traversal

```
void preorder(struct node *root)
{
    if(root!=NULL) // checking if the root is not null
    {
        printf(" %c ", root->data); // printing data at root
        preorder(root->left_child); // visiting left child
        preorder(root->right_child); // visiting right child
    }
}
```

In preorder traversal, we first visit the root and then the left subtree and lastly the right subtree. We are doing the same here.

`printf(" %c ", root->data)` – We are first visiting the root (of the main tree or subtree) or the current node then we will visit its left subtree and then the right subtree.

`preorder(root->left_child)` – Then we are visiting the left subtree.

`preorder(root->right_child)` – And lastly the right subtree.

Postorder Traversal

```
void postorder(struct node *root)
{
    if(root!=NULL) // checking if the root is not null
    {
        postorder(root->left_child); // visiting left child
        postorder(root->right_child); // visiting right child
        printf(" %c ", root->data); // printing data at root
    }
}
```

In postorder traversal, we first visit the left subtree and then the right and lastly the node.

Inorder Traversal

```
void inorder(struct node *root)
{
    if(root!=NULL) // checking if the root is not null
    {
        inorder(root->left_child); // visiting left child
        printf(" %c ", root->data); // printing data at root
        inorder(root->right_child);// visiting right child
    }
}
```

We first visit the left subtree and then root and lastly the right subtree in inorder traversal.

Height of a Node or Binary Tree

Height of a node is 1+ greater heights among the left subtree and the right subtree. Also, the height of a leaf node or a null node is 0. Thus, we first write a function to identify a leaf node.

Function to Identify Leaves in Binary Tree

```
int is_leaf(struct node *a)
{
    if(a->right_child==NULL && a->left_child==NULL)
        return 1;
    return 0;
}
```

Checking for a leaf node is simple. If both the children of a node are null then it is a leaf node. We are checking the same with – `if(a->right_child==NULL && a->left_child==NULL)`.

Now, we are ready to write a function to get the height of any node of tree.

```
// function to return maximum of two numbers
int get_max(int a, int b)
{
    return (a>b) ? a : b;
}

//function to get the height of a tree or node
int get_height(struct node *a)
{
    if(a==NULL || is_leaf(a)) // height will be 0 if the node is leaf or nul
        return 0;
    //height of a node will be 1+ greater among height of right subtree and
    return(get_max(get_height(a->left_child), get_height(a->right_child)) + 1
}
```

'get_max' is a function to determine the greater number of the two numbers passed to it.

'get_height' is the function to calculate the height of the tree. We are first checking for a null node or leaf node with `if(a==NULL || is_leaf(a))`. In both cases, the height will be 0. Else, the height will be 1+maximum among the heights of left and the right subtrees – `get_max(get_height(a->left_child), get_height(a->right_child)) + 1`.

Let's implement the above concepts and see the result.

```c
#include <stdio.h>
#include <stdlib.h>
/*

            D
           / \
          /   \
         /     \
        A       F
       / \
      /   \
     E     B
*/

struct node
{
    char data; //node will store character
    struct node *right_child; // right child
    struct node *left_child; // left child
};

struct node* new_node(char data)
{
    struct node *p; // node
    p = malloc(sizeof(struct node)); // allocating space to p
    p->data = data; // assinging data to p
    p->left_child = NULL; // making children NULL
    p->right_child = NULL;

    return(p); // returning the newly made node
}

void preorder(struct node *root)
{
    if(root!=NULL) // checking if the root is not null
    {
        printf(" %c ", root->data); // printing data at root
        preorder(root->left_child); // visiting left child
        preorder(root->right_child); // visiting right child
    }
}

void postorder(struct node *root)
{
    if(root!=NULL) // checking if the root is not null
    {
        postorder(root->left_child); // visiting left child
        postorder(root->right_child); // visiting right child
        printf(" %c ", root->data); // printing data at root
    }
}

void inorder(struct node *root)
{
    if(root!=NULL) // checking if the root is not null
    {
        inorder(root->left_child); // visiting left child
        printf(" %c ", root->data); // printing data at root
        inorder(root->right_child);// visiting right child
    }
}

int is_leaf(struct node *a)
{
    if(a->right_child==NULL && a->left_child==NULL)
        return 1;
    return 0;
}

// function to return maximum of two numbers
int get_max(int a, int b)
{
    return (a>b) ? a : b;
```

```
}

//function to get the height of a tree or node
int get_height(struct node *a)
{
    if(a==NULL || is_leaf(a)) // height will be 0 if the node is leaf or nul
        return 0;
    //height of a node will be 1+ greater among height of right subtree and
    return(get_max(get_height(a->left_child), get_height(a->right_child)) + 1
}

int main()
{
    struct node *root; //new structure
    root = new_node('D'); // making a new node
    /*
     ____
    |  D |
    |____|

    */
    root->left_child = new_node('A'); //left child of root

    /*
              ____
             |  D |
            /|____|
     ____ /
    |  A |
    |____|

    */

    root->right_child = new_node('F'); //right child of root

    /*
              ____
             |  D |
            /|____|\
     ____ /        _\___
    |  A |         |  F |
    |____|         |____|

    */

    root->left_child->left_child = new_node('E'); // new node

    /*
                       ____
                      |  D |
                     /|____|\
              ____ /        _\___
             |  A |         |  F |
            /|____|         |____|
      _____/
     |  E |
     |____|

    */

    root->left_child->right_child = new_node('B'); // new node

        /*
                       ____
                      |  D |
                     /|____|\
              ____ /        _\___
             |  A |         |  F |
            /|____|\        |____|
      _____/        _\___
     |  E |       |   B |
     |____|       |_____|

    */
```

```
    preorder(root);
    printf("\n");
    postorder(root);
    printf("\n");
    inorder(root);
    printf("\n");

    printf("%d\n", get_height(root));

    return 0;
}
```

Output:

```
 D   A   E   B   F

 E   B   A   F   D

 E   A   B   D   F

2
```

Next:

1. Binary Search Tree (https://www.codesdope.com/blog/article/binary-search-tree)
2. Binary Search Tree in C (https://www.codesdope.com/blog/article/binary-search-tree-in-c)
