AVL Tree

#include&lt;stdio.h&gt;

#include&lt;stdlib.h&gt;

// structure of the tree node

struct node

{

    int data;

    struct node\* left;

    struct node\* right;

    int ht;

};

// global initialization of root node

struct node\* root = NULL;

// function prototyping

struct node\* create(int);

struct node\* insert(struct node\*, int);

struct node\* delete(struct node\*, int);

struct node\* search(struct node\*, int);

struct node\* rotate\_left(struct node\*);

struct node\* rotate\_right(struct node\*);

int balance\_factor(struct node\*);

int height(struct node\*);

void inorder(struct node\*);

void preorder(struct node\*);

void postorder(struct node\*);

int main()

{

    int user\_choice, data;

    char user\_continue = &#39;y&#39;;

    struct node\* result = NULL;

    while (user\_continue == &#39;y&#39; || user\_continue == &#39;Y&#39;)

    {

        printf(&quot;\n\n------- AVL TREE --------\n&quot;);

        printf(&quot;\n1. Insert&quot;);

        printf(&quot;\n2. Delete&quot;);

        printf(&quot;\n3. Search&quot;);

        printf(&quot;\n4. Inorder&quot;);

        printf(&quot;\n5. Preorder&quot;);

        printf(&quot;\n6. Postorder&quot;);

        printf(&quot;\n7. EXIT&quot;);

        printf(&quot;\n\nEnter Your Choice: &quot;);

        scanf(&quot;%d&quot;, &amp;user\_choice);

        switch(user\_choice)

        {

            case 1:

                printf(&quot;\nEnter data: &quot;);

                scanf(&quot;%d&quot;, &amp;data);

                root = insert(root, data);

                break;

            case 2:

                printf(&quot;\nEnter data: &quot;);

                scanf(&quot;%d&quot;, &amp;data);

                root = delete(root, data);

                break;

            case 3:

                printf(&quot;\nEnter data: &quot;);

                scanf(&quot;%d&quot;, &amp;data);

                result = search(root, data);

                if (result == NULL)

                {

                    printf(&quot;\nNode not found!&quot;);

                }

                else

                {

                    printf(&quot;\n Node found&quot;);

                }

                break;

            case 4:

                inorder(root);

                break;

            case 5:

                preorder(root);

                break;

            case 6:

                postorder(root);

                break;

            case 7:

                printf(&quot;\n\tProgram Terminated\n&quot;);

                return 1;

            default:

                printf(&quot;\n\tInvalid Choice\n&quot;);

        }

        printf(&quot;\n\nDo you want to continue? &quot;);

        scanf(&quot; %c&quot;, &amp;user\_continue);

    }

    return 0;

}

// creates a new tree node

struct node\* create(int data)

{

    struct node\* new\_node = (struct node\*) malloc (sizeof(struct node));

    // if a memory error has occurred

    if (new\_node == NULL)

    {

        printf(&quot;\nMemory can&#39;t be allocated\n&quot;);

        return NULL;

    }

    new\_node-&gt;data = data;

    new\_node-&gt;left = NULL;

    new\_node-&gt;right = NULL;

    return new\_node;

}

// rotates to the left

struct node\* rotate\_left(struct node\* root)

{

    struct node\* right\_child = root-&gt;right;

    root-&gt;right = right\_child-&gt;left;

    right\_child-&gt;left = root;

    // update the heights of the nodes

    root-&gt;ht = height(root);

    right\_child-&gt;ht = height(right\_child);

    // return the new node after rotation

    return right\_child;

}

// rotates to the right

struct node\* rotate\_right(struct node\* root)

{

    struct node\* left\_child = root-&gt;left;

    root-&gt;left = left\_child-&gt;right;

    left\_child-&gt;right = root;

    // update the heights of the nodes

    root-&gt;ht = height(root);

    left\_child-&gt;ht = height(left\_child);

    // return the new node after rotation

    return left\_child;

}

// calculates the balance factor of a node

int balance\_factor(struct node\* root)

{

    int lh, rh;

    if (root == NULL)

        return 0;

    if (root-&gt;left == NULL)

        lh = 0;

    else

        lh = 1 + root-&gt;left-&gt;ht;

    if (root-&gt;right == NULL)

        rh = 0;

    else

        rh = 1 + root-&gt;right-&gt;ht;

    return lh - rh;

}

// calculate the height of the node

int height(struct node\* root)

{

    int lh, rh;

    if (root == NULL)

    {

        return 0;

    }

    if (root-&gt;left == NULL)

        lh = 0;

    else

        lh = 1 + root-&gt;left-&gt;ht;

    if (root-&gt;right == NULL)

        rh = 0;

    else

        rh = 1 + root-&gt;right-&gt;ht;

    if (lh &gt; rh)

        return (lh);

return (rh);

}

// inserts a new node in the AVL tree

struct node\* insert(struct node\* root, int data)

{

    if (root == NULL)

    {

        struct node\* new\_node = create(data);

        if (new\_node == NULL)

        {

            return NULL;

        }

        root = new\_node;

    }

    else if (data &gt; root-&gt;data)

    {

        // insert the new node to the right

        root-&gt;right = insert(root-&gt;right, data);

        // tree is unbalanced, then rotate it

        if (balance\_factor(root) == -2)

        {

            if (data &gt; root-&gt;right-&gt;data)

            {

                root = rotate\_left(root);

            }

            else

            {

                root-&gt;right = rotate\_right(root-&gt;right);

                root = rotate\_left(root);

            }

        }

    }

    else

    {

        // insert the new node to the left

        root-&gt;left = insert(root-&gt;left, data);

        // tree is unbalanced, then rotate it

        if (balance\_factor(root) == 2)

        {

            if (data &lt; root-&gt;left-&gt;data)

            {

                root = rotate\_right(root);

            }

            else

            {

                root-&gt;left = rotate\_left(root-&gt;left);

                root = rotate\_right(root);

            }

        }

    }

    // update the heights of the nodes

    root-&gt;ht = height(root);

    return root;

}

// deletes a node from the AVL tree

struct node \* delete(struct node \*root, int x)

{

    struct node \* temp = NULL;

    if (root == NULL)

    {

        return NULL;

    }

    if (x &gt; root-&gt;data)

    {

        root-&gt;right = delete(root-&gt;right, x);

        if (balance\_factor(root) == 2)

        {

            if (balance\_factor(root-&gt;left) &gt;= 0)

            {

                root = rotate\_right(root);

            }

            else

            {

                root-&gt;left = rotate\_left(root-&gt;left);

                root = rotate\_right(root);

            }

        }

    }

    else if (x &lt; root-&gt;data)

    {

root-&gt;left = delete(root-&gt;left, x);

        if (balance\_factor(root) == -2)

        {

            if (balance\_factor(root-&gt;right) &lt;= 0)

            {

                root = rotate\_left(root);

            }

            else

            {

                root-&gt;right = rotate\_right(root-&gt;right);

                root = rotate\_left(root);

            }

        }

    }

    else

    {

        if (root-&gt;right != NULL)

        {

            temp = root-&gt;right;

            while (temp-&gt;left != NULL)

                temp = temp-&gt;left;

            root-&gt;data = temp-&gt;data;

            root-&gt;right = delete(root-&gt;right, temp-&gt;data);

            if (balance\_factor(root) == 2)

            {

                if (balance\_factor(root-&gt;left) &gt;= 0)

                {

                    root = rotate\_right(root);

                }

                else

                {

                    root-&gt;left = rotate\_left(root-&gt;left);

                    root = rotate\_right(root);

                }

            }

        }

        else

        {

            return (root-&gt;left);

        }

    }

    root-&gt;ht = height(root);

    return (root);

}

// search a node in the AVL tree

struct node\* search(struct node\* root, int key)

{

    if (root == NULL)

    {

        return NULL;

    }

    if(root-&gt;data == key)

    {

        return root;

    }

    if(key &gt; root-&gt;data)

    {

        search(root-&gt;right, key);

    }

    else

    {

        search(root-&gt;left, key);

    }

}

// inorder traversal of the tree

void inorder(struct node\* root)

{

    if (root == NULL)

    {

        return;

    }

    inorder(root-&gt;left);

    printf(&quot;%d &quot;, root-&gt;data);

    inorder(root-&gt;right);

}

// postorder traversal of the tree

void postorder(struct node\* root)

{

    if (root == NULL)

    {

        return;

    }

    postorder(root-&gt;left);

    postorder(root-&gt;right);

    printf(&quot;%d &quot;, root-&gt;data);

}