```
//::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
//                        CSCI112 Program 4 - Joel Cressy
//::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
//Fixed my stack functions! everything is using just plain functions and structs.

using namespace std;

//<tree declarations>
struct treeNode
{
    int data;
    treeNode *left;
    treeNode *right;
};

bool emptytree(treeNode *root) { return root == NULL; }
void iter_insert(treeNode *&, int);
void iter_inorder(treeNode *, ofstream &);
void iter_preorder(treeNode *, ofstream &);
void iter_postorder(treeNode *, ofstream &);
void preorder(treeNode *, ofstream &);
void inorder(treeNode *, ofstream &);
void postorder(treeNode *, ofstream &);
treeNode *ins(treeNode *, int);
treeNode *getmax(treeNode *);
treeNode *del(treeNode *, int);
//<tree declarations>

//<stack declarations>
struct stacktype {
    treeNode* data;
    stacktype* next;
};
stacktype* pushstack(stacktype* s,treeNode* data);
stacktype* popstack(stacktype*);
bool emptystack(stacktype*& top) { return top == NULL; }
//</stack declarations>

//::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
//                             Main Body of Program
//::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
void readem(treeNode *&root, treeNode *&t2, ifstream &inf)
{
    int temp;
    while (!inf.eof())
    {
        inf >> temp;
        iter_insert(root, temp); //insert iteratively
        t2 = ins(t2, temp); //insert recursively
    }
}

void printem(treeNode *&root, treeNode *&t2, ofstream &outf)
{
    outf << setfill(':') << setw(80) << "" << endl;
    outf << setfill(' ') << setw(60) << "Primary Assignment Using Iterative Insert" <<
 endl;
    outf << "Recursive inorder traversal:" << endl;
    inorder(root, outf);
    outf << endl << endl;
    root = del(root, 71);
    outf << "Deleted node 71, print using recursive postorder traversal:" << endl;
    postorder(root, outf);
    outf << endl << endl;
    root = del(root, 38);
```

```
    outf << "Deleted node 38, print using recursive preorder traversal:" << endl;
    preorder(root, outf);
    outf << endl << endl;
    outf << "iterative inorder traversal:" << endl;
    iter_inorder(root, outf);
    outf << endl << endl;
    outf << "iterative preorder traversal:" << endl;
    iter_preorder(root, outf);
    outf << endl << endl;
    outf << "iterative postorder traversal:" << endl;
    iter_postorder(root, outf);
    outf << endl << endl;
    outf << setfill(':') << setw(80) << "" << endl;
    outf << setfill(' ') << setw(55) << "This time, with recursive insert" << endl;
    outf << "Recursive Inorder: ";
    inorder(t2, outf);
    outf << endl << endl;
    outf << "Recursive preorder: ";
    preorder(t2,outf);
    outf << endl << endl;
    outf << "Recursive postorder: ";
    postorder(t2,outf);
    outf << endl << endl;
    outf << "Iterative inorder: ";
    iter_inorder(t2, outf);
    outf << endl << endl;
    outf << "Iterative preorder: ";
    iter_preorder(t2, outf);
    outf << endl << endl;
    outf << "Iterative postorder: ";
    iter_postorder(t2, outf);
    outf << endl << endl;
}

int main(int argc, char *argv[])
{
    string infile_name = "input.dat";
    //I would sometimes pass larger, alternate test files to the program
    if (argc > 1)
        infile_name = argv[1];
    treeNode *t1 = NULL;
    treeNode *t2 = NULL;
    ifstream inf(infile_name);
    ofstream outf("output.ot");
    readem(t1, t2, inf);
    printem(t1, t2, outf);
    return 0;
}

//:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
//                    Function Implementations - Iterative
//:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
void iter_insert(treeNode *&root, int data)
{
    treeNode *parent, *C;
    parent = root;
    C = root;
    if (!emptytree(root))
    {
        while (C != NULL)
        {
            parent = C;
            C = (data < C->data) ? C->left : C->right;
        }
        C = new treeNode({data,NULL,NULL});
        if (C->data < parent->data)
        {
            parent->left = C;
        }
```

```
        else
        {
            parent->right = C;
        }
    }
    else
    {
        root = new treeNode;
        root -> data = data;
    }
}

void iter_preorder(treeNode *root, ofstream &outf)
{
    treeNode *C;
    stacktype* s = NULL;
    s = pushstack(s, root);

    while (!emptystack(s))
    {
        C = s->data; //s is top
        outf << C->data << " ";
        s = popstack(s);
        if(C->right != NULL)
            s = pushstack(s,C->right);
        if(C->left != NULL)
            s = pushstack(s,C->left);
    }
}

void iter_inorder(treeNode *root, ofstream &outf)
{
    treeNode *C = root;
    stacktype* s = NULL;
    bool done = false;
    while(!done) {
        if(C != NULL)
        {
            s = pushstack(s,C);
            C = C->left;
        }
        else {
            if (emptystack(s))
            {
                done = true;
            }
            else {
                C = s->data;
                s = popstack(s);
                if (C != NULL) outf << C->data << " ";
                C = C->right;
            }
        }
    }
}

void iter_postorder(treeNode *C, ofstream &outf)
{
    treeNode *parent = NULL;
    stacktype* s = NULL;
    do {
        while (C!=NULL)
        {
            s = pushstack(s,C);
            C = C->left;
        }
        while (C == NULL && !emptystack(s))
        {
            C = s->data;
```

```
            if (C->right != NULL && C->right != parent)
            {
                C = C->right;
            }
            else
            {
                outf << C->data << " ";
                s = popstack(s);
                parent = C;
                C = NULL;
            }
        }
    }
    while (!emptystack(s));
}
//<stack implementations>
stacktype* pushstack(stacktype* top, treeNode* data)
{
    if(!emptystack(top))
    {
        stacktype *pushed = new stacktype;
        pushed->data = top->data;
        pushed->next = top->next;
        top->data = data;
        top->next = pushed;
    }
    else
    {
        top = new stacktype;
        top->data = data;
        top->next = NULL;
    }
    return top;
}
stacktype* popstack(stacktype* top)
{
    stacktype *newtop = new stacktype;
    if (top->next != NULL)  {
        newtop = top->next;
        top->data = newtop->data;
        top->next = newtop->next;
    }
    else
    {
        top = NULL;
    }
    delete newtop;
    return top;
}
//::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
//                            Recursive Functions
//::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
void preorder(treeNode *C, ofstream &outf)
{
    if (C != NULL)
    {
        outf << C->data << " ";
        preorder(C->left, outf);
        preorder(C->right, outf);
    }
}

void inorder(treeNode *C, ofstream &outf)
{
    if (C != NULL)
    {
        inorder(C->left, outf);
        outf << C->data << " ";
        inorder(C->right, outf);
```

```
    }
}

void postorder(treeNode *C, ofstream &outf)
{
    if (C != NULL)
    {
        postorder(C->left, outf);
        postorder(C->right, outf);
        outf << C->data << " ";
    }
}


treeNode *ins(treeNode *C, int data)
{
    if(C != NULL)
    {
        if(data < C->data)
            C->left = ins(C->left, data);
        if(data >= C->data)
            C->right = ins(C->right, data);
    }
    else
    {
        C = new treeNode;
        C->data = data;
    }
    return C; //C is returned to parent so that their link gets updated
}

treeNode *getmax(treeNode *C)
{
    if (C == NULL)
        return NULL;
    //The right children will always be the highest number in a BST
    while (C->right != NULL)
        C = C->right;
    return C;
}

treeNode *del(treeNode *C, int data)
{
    if (C != NULL)
    {
        //find the node
        if (data < C->data)
        {
            C->left = del(C->left, data);
        }
        else if (data > C->data)
        {
            C->right = del(C->right, data);
        }
        else
        {
            //no child
            if (C->right == NULL && C->left == NULL)
            {
                delete C;
                C = NULL; //return NULL to the parent so they can update their links
            }
            else
            {
                if (C->right == NULL)
                {
                    treeNode *temp = C; //temp acts as the trash can
                    C = C->left; //Since we're returning C, send C's left to the paren
t.
```

```
                    delete temp; //dump the trash
                }
                if (C->left == NULL)
                {
                    treeNode *temp = C;
                    C = C->right; //same thing, except we're returning the left node.
                    delete temp;
                }
                //two child
                if (C->left != NULL && C->right != NULL)
                {
                    //get the largest node in our left subtree
                    treeNode *temp = getmax(C->left);
                    //copy the found node's data into the current node'
                    C->data = temp->data;
                    //delete the largest node in the left subtree and store that treeN
ode in the left link
                    C->left = del(C->left, temp->data);
                }
            }
        }
    }
    return C;
}
```