```
//*************************************     PROGRAM IDENTIFICATION   **********************************
//*                                                                                                   *
//*     PROGRAM FILE NAME:   Program7.cpp           ASSIGNMENT  #:  7           Grade: _______      *
//*                                                                                                   *
//*     PROGRAM AUTHOR:       ____________________________________                                    *
//*                                   Adrian Beloqui                                                  *
//*                                                                                                   *
//*     COURSE #:   CSC  36000  11                                  DUE DATE:    May 5, 2017        *
//*                                                                                                   *
//*****************************************************************************************************

//*************************************    PROGRAM DESCRIPTION   **************************************
//*                                                                                                   *
//*      PROCESS: This program is designed to read a file and store the records as items of an inventory  *
//*               into a binary threaded tree structure. It is to read commands and perform           *
//*               different actions depending on the command that is executed. It is to insert,       *
//*               delete, update and print the items from the inventory.                              *
//*                                                                                                   *
//*      USER DEFINED                                                                                 *
//*      MODULES      : main - Controlls the flow of the entire program, calling functions is the     *
//*                            right sequence and printing the labels into the output file.           *
//*                     Header - Prints a header in the output file.                                  *
//*                     Footer - Prints a footer in the output file.                                  *
//*                     PageBreak - Adds end lines to the output file.                                *
//*                     InventoryCLASS::InventoryCLASS - Initializes the private members of the class.    *
//*                     InventoryCLASS::ReadNode- Read an item from the input file.                   *
//*                     InventoryCLASS::Print - Prints all the items from the inventory or only one.  *
//*                     InventoryCLASS::InsertNode - Inserts a node into the binary tree.             *
//*                     InventoryCLASS::GetRoot - Gets the pointer of the root of the binary tree     *
//*                     InventoryCLASS::CheckExistance - Checks if a node exists in the binary tree and   *
//*                                                      returns it.                                  *
//*                     InventoryCLASS::DeleteNode - Deletes a node from the binary tree.             *
//*                     InventoryCLASS::UpdateNode - Updates a node depending on the command executed.    *
//*                                                                                                   *
//*****************************************************************************************************

    //Imports
#include <string>
#include <fstream>
#include <iomanip>

    //Definition of constants
#define NOT !
#define LinesPerPage 66

    //Definition of namespace
using namespace std;

    //Definition of a node structure
struct NodeType{
    char id[6];
    char name[21];
    int quantityOnHand;
    int quantiyOnOrder;

    NodeType *LPtr;
    NodeType *RPtr;
    int Thread;
};

    //Definition of classes
class InventoryCLASS{
public:
        // Constructor
    InventoryCLASS() { Root = NULL; };
```

```
        // Functions
    bool InsertNode(NodeType &);
    void ReadNode(ifstream &, NodeType &, char);
    bool Print(ofstream &, int &, NodeType *, char, NodeType *);
    void THIOT(ofstream &, int &);
    NodeType* GetRoot() { return Root; }
    NodeType* CheckExistance(NodeType *);
    bool DeleteNode(ofstream &, NodeType &);\
    bool UpdateNode(NodeType *, char);
private:
    NodeType *Root;
};

    //Function prototypes definitions
void Header(ofstream &);
void Footer(ofstream &);
void PageBreak(ofstream &, int &);

//************************************** FUNCTION MAIN  ********************************************
int main()
{
    ifstream InFile;
    ofstream OutFile;

        //Set initial variables
    int linesWritten = 0;
    bool endOfFile = false;
    char command, printingType;
    NodeType tempNode;
    InventoryCLASS inventory;

        // Open the input file
    InFile.open("thread_in.txt", ios::in);

        // Create the output file
    OutFile.open("output7.txt", ios::out);

        // Print the header in the output file.
    Header(OutFile);
        // Add amount of lines written into the output file
    linesWritten += 3;
        // Print separator line
    OutFile << "================================================="
        << "============================================================" << endl << endl;
        // Add amount of lines written into the output file
    linesWritten += 2;

        // Read the input file
    do {
            // Read the command character
        InFile >> command;
            // Execute the right intructions depending on the command executed
        switch (command){
        case 'I':
                // Read a node
            inventory.ReadNode(InFile, tempNode, command);
                // Insert a node
            if (inventory.InsertNode(tempNode)){
                    // Print success message
                OutFile << "Item ID Number " << tempNode.id << " successfully entered into database."
                    << endl;
                OutFile << "------------------------------------------------------------" << endl;
                linesWritten += 2;
            }
            else {
                    // Print error duplicate message
```

```
            OutFile << "ERROR - Attempt to insert a duplicate item " << tempNode.id
                << " into the database." << endl;
            OutFile << "-------------------------------------------------------------" << endl;
            linesWritten += 2;
        }
        break;
    case 'D':
            // Read a node
        inventory.ReadNode(InFile, tempNode, command);
            // Delete a node
        if (inventory.DeleteNode(OutFile, tempNode)){
                // Print success message
            OutFile << "Item ID Number " << tempNode.id << " successfully deleted from database."
                << endl;
            OutFile << "-------------------------------------------------------------" << endl;
            linesWritten += 2;
        }
        else {
                // Print error message
            OutFile << "ERROR --- Attempt to delete an item " << tempNode.id
                << " not in the database list." << endl;
            OutFile << "-------------------------------------------------------------" << endl;
            linesWritten += 2;
        }
        break;
    case 'P':
            // Read printing type
        InFile >> printingType;
            // Print page break
        PageBreak(OutFile, linesWritten);
            // Print labels
        OutFile << setw(40) << "JAKE'S HARDWARE INVENTORY REPORT" << endl;
        OutFile << "Item" << setw(16) << "Item" << setw(26) << "Quantity"
            << setw(13) << "Quantity" << endl;
        OutFile << "ID Number" << setw(18) << "Description" << setw(19) << "On Hand"
            << setw(13) << "On Order" << endl;
        OutFile << "-------------------------------------------------------------" << endl;
        if (printingType == 'N'){
                // Read node
            inventory.ReadNode(InFile, tempNode, printingType);
                // Print node
            if (NOT inventory.Print(OutFile, linesWritten, inventory.GetRoot(),
                printingType, &tempNode)){
                    // Print error message
                OutFile << "Item " << tempNode.id << " not in database. Print failed."
                    << endl;
                OutFile << "-------------------------------------------------------------"
                    << endl;
                linesWritten += 2;
            }
        }
        else {
                // Print tree
            inventory.Print(OutFile, linesWritten, inventory.GetRoot(), printingType, NULL);
        }
        OutFile << endl << endl;
        linesWritten += 6;
        break;
    case 'S':
            // Read node
        inventory.ReadNode(InFile, tempNode, command);
            // Update node
        if (inventory.UpdateNode(&tempNode, command)){
                // Print success message
            OutFile << "Quantity on Hand for item " << tempNode.id
                << " successfully updated." << endl;
```

```
                OutFile << "------------------------------------------------------------" << endl;
                linesWritten += 2;
            }
            else {
                    // Print error message
                OutFile << "Item " << tempNode.id << " not in database. Data not updated." << endl;
                OutFile << "------------------------------------------------------------" << endl;
                linesWritten += 2;
            }
            break;
        case 'O':
                // Read Node
            inventory.ReadNode(InFile, tempNode, command);
                // Update Node
            if (inventory.UpdateNode(&tempNode, command)){
                    // Print success message
                OutFile << "Quantity on Order For item  " << tempNode.id
                    << " successfully updated." << endl;
                OutFile << "------------------------------------------------------------" << endl;
                linesWritten += 2;
            }
            else {
                    // Print error message
                OutFile << "Item " << tempNode.id << " not in database. Data not updated." << endl;
                OutFile << "------------------------------------------------------------" << endl;
                linesWritten += 2;
            }
            break;
        case 'R':
                // Read node
            inventory.ReadNode(InFile, tempNode, command);
                // Update node
            if (inventory.UpdateNode(&tempNode, command)){
                    // Print success message
                OutFile << "Quantity on Hand for item " << tempNode.id
                    << " successfully updated." << endl;
                OutFile << "------------------------------------------------------------" << endl;
                linesWritten += 2;
            }
            else {
                    // Print error message
                OutFile << "Item " << tempNode.id << " not in database. Data not updated." << endl;
                OutFile << "------------------------------------------------------------" << endl;
                linesWritten += 2;
            }
            break;
        case 'Q':
            endOfFile = true;
            break;
        }
    } while (NOT endOfFile);

        // Print page break
    PageBreak(OutFile, linesWritten);

        // Print the footer into the output file.
    Footer(OutFile);

    return 0;
}
//***********************************  END OF FUNCTION MAIN  ****************************************

//************************************** FUNTION DELETENODE  ****************************************
bool InventoryCLASS::DeleteNode(ofstream &OutFile, NodeType &node)
{
        // Receives - The output file and a node
```

```
        // Task - Deletes a node from the tree
        // Returns - The output file, a node and a boolean

    bool found = false;       /* a flag to indicate a node is found  */
    NodeType  *delnode, *parnode, *node1, *node2, *node3;

    delnode = Root;
    parnode = NULL;
        // Find the node to be deleted
    while ((found == false) && (delnode != NULL) && (delnode->Thread != 1)) {
        if (strcmp(node.id, delnode->id) == 0){
            found = true;
        }
        else
        {
            parnode = delnode;
            if (strcmp(node.id, delnode->id) < 0) {
                delnode = delnode->LPtr;
            }
            else {
                delnode = delnode->RPtr;
            }
        }
    }
        // Check if the delete node is a thread
        // Previous while loop does not cover this case, we need to check it specifically
    if (found == false && delnode != NULL && delnode->Thread == 1){
        if (strcmp(node.id, delnode->id) == 0){
            found = true;
        }
    }

        // CASE 1 - NODE NOT FOUND
    if (found == false)
    {
        return found;
    }
    //=============================================
        //  CASE 2 -- NODE WITH NO CHILDREN
    if ((delnode->LPtr == NULL) && (((delnode->RPtr == NULL) || (delnode->Thread == 1))))
    {
        if (parnode == NULL)
            Root = NULL;
        else {
            if (parnode->LPtr == delnode)
                parnode->LPtr = NULL;
            else
            {
                parnode->RPtr = delnode->RPtr;
                parnode->Thread = 1;
            }
        }
        return found;
    }
    //=============================================
        //  CASE 3 -- NODE WITH ONE RIGHT CHILD
    if ((delnode->LPtr == NULL) && (((delnode->RPtr != NULL) || (delnode->Thread == 0)))) {
        if (parnode == NULL) {
            Root = delnode->RPtr;
        }
        else {
            if (parnode->LPtr == delnode) {         // Delete node is a LEFT CHILD
                parnode->LPtr = delnode->RPtr;
            }
            else {              // Delete node is a RIGHT CHILD
                parnode->RPtr = delnode->RPtr;
```

```
            }
        }
        return found;
    }
    //=============================================
        //  CASE 4 -- NODE WITH ONE LEFT CHILD
    if ((delnode->LPtr != NULL) && ((delnode->RPtr == NULL) || (delnode->Thread == 1))) {
        if (parnode == NULL) {
            Root = delnode->LPtr;
        }
        else {
            if (parnode->LPtr == delnode) { // Delete node is a LEFT CHILD
                parnode->LPtr = delnode->LPtr;
            }
            else {                          // Delete node is a RIGHT CHILD
                parnode->RPtr = delnode->LPtr;
            }
        }
        node1 = delnode->LPtr;
        while ((node1->RPtr != NULL) && (node1 -> Thread != 1)) {
            node1 = node1->RPtr;
        }
        node1->RPtr = delnode->RPtr;

        return found;
    }
    //=============================================
        // CASE 5 -- NODE WITH TWO CHILDREN
        // CASE 5 --- NODE WITH  2 CHILDREN
        // Find the rightmost child in the left sub-tree off the Delete node
    if ((delnode->LPtr != NULL) && (delnode->RPtr != NULL) && (delnode->Thread == 0)) {
        node1 = delnode;
        node2 = delnode->LPtr;
        node3 = delnode->LPtr;
        while ((node3 != NULL) && (node3->Thread != 1)) {
            node2 = node3;
            node3 = node3->RPtr;
        }
        //=========================================
        //  Six cases:
            //1.  Delete node is a  LEFT CHILD
            //2.  Delete node is a RIGHT CHILD
            //3.  Left child of delnode has a RIGHT subtree  but no LEFT subtree
            //4.  Left child of delnode has a LEFT subtree  but no RIGHT subtree
            //5.  Left child of delnode has BOTH a LEFT subtree  and a RIGHT subtree
        if (parnode == NULL) {
            Root = node3;
        }
        else {
            if (parnode->LPtr == delnode) {  //  Delete node is a  LEFT CHILD
                parnode->LPtr = node3;
            }
            else {                      // Delete node is a RIGHT CHILD
                parnode->RPtr = node3;
            }
        }

        if ((node3->LPtr != NULL) && (node2 != node3)) {
            node2->RPtr = node3->LPtr; // Left child of delnode has a RIGHT subtree but no LEFT subtree
            node3->LPtr = delnode->LPtr;
            node3->RPtr = delnode->RPtr;
            node3->Thread = 0;
        }

        if ((node3->LPtr != NULL) && (node2 != node3)) {
            node2->RPtr = node3->LPtr; // Left child of delnode has a RIGHT subtree but no LEFT subtree
```

```
            node3->LPtr = delnode->LPtr;
            node3->RPtr = delnode->RPtr;
            node3->Thread = 0;
        }
        else if (node2 == node3) {      // Left child of delnode has a LEFT subtree but no RIGHT subtree
            node3->RPtr = delnode->RPtr;
            node3->Thread = 0;
        }
        else if ((node2->RPtr != NULL)) {
            node2->RPtr = node3;        // Left child of delnode has BOTH a LEFT subtree and
            node2->Thread = 1;               // a RIGHT subtree
            node3->LPtr = delnode->LPtr;
            node3->RPtr = delnode->RPtr;
            node3->Thread = 0;
        }
    }
    return found;
}
//***********************************  END OF FUNCTION DELETENODE  ************************************

//*************************************** FUNTION UPDATENODE  ***************************************
bool InventoryCLASS::UpdateNode(NodeType *node, char command) {

        // Receives - A node and a character indicating the command executed
        // Task - Updates the members of a node depending on the command executed
        // Returns - A boolean

        // Check if the node exists
    NodeType *nodeToUpdate = CheckExistance(node);
    if (nodeToUpdate == NULL){
        return false;
    }

    if (node != NULL)
    {
            // Update a node depending on the command executed
        switch (command){
        case 'S':
            if (strcmp(node->id, nodeToUpdate->id) == 0){
                nodeToUpdate->quantityOnHand -= node->quantityOnHand;
                return true;
            }
            break;
        case 'O':
            if (strcmp(node->id, nodeToUpdate->id) == 0){
                nodeToUpdate->quantiyOnOrder += node->quantityOnHand;
                return true;
            }
            break;
        case 'R':
            if (strcmp(node->id, nodeToUpdate->id) == 0){
                nodeToUpdate->quantityOnHand += node->quantityOnHand;
                nodeToUpdate->quantiyOnOrder -= node->quantityOnHand;
                return true;
            }
            break;
        }
    }

    return false;
}
//***********************************  END OF FUNCTION UPDATENODE  ************************************

//*************************************** FUNTION THIOT  ********************************************
void InventoryCLASS::THIOT(ofstream &OutFile, int &linesWritten) {
```

```
        // Receives - The output file, the amount of lines written
        // Task - Print the entire tree in order
        // Returns - The output file and the lines written

    int  RightThread;
    NodeType  *CurrPtr;

    CurrPtr = Root;

    while (CurrPtr->LPtr != NULL) {
        CurrPtr = CurrPtr->LPtr;
    }

    while (CurrPtr != NULL) {
        //   PROCESS THE NODE HERE

        OutFile << setw(6) << CurrPtr->id;
        OutFile << setw(30) << CurrPtr->name;
        OutFile << setw(10) << CurrPtr->quantityOnHand;
        OutFile << setw(10) << CurrPtr->quantiyOnOrder << endl;
        linesWritten++;

        RightThread = CurrPtr->Thread;
        CurrPtr = CurrPtr-> RPtr;
        if ((CurrPtr != NULL) && (RightThread == 0)) {
            while (CurrPtr->LPtr != NULL) {
                CurrPtr = CurrPtr->LPtr;
            }
        }
    }
    return;
}
//************************************  END OF FUNCTION THIOT  ****************************************

//************************************** FUNTION PRINT  ********************************************
bool InventoryCLASS::Print(ofstream &OutFile, int &linesWritten, NodeType *root, char printingType,
    NodeType *node) {

        // Receives - The output file, the amount of lines written, the root node, the printing
        //            character, and a node
        // Task - Print the entire tree or just one node
        // Returns - The output file, the lines written and a boolean

        // Check what type of printing is
    switch (printingType){
    case 'E':
        if (root != NULL)
        {
                // Print node
            THIOT(OutFile, linesWritten);
        }
        break;
    case 'N':
        if (root != NULL)
        {
                // Check if the node exists
            NodeType *tempNode = CheckExistance(node);

                // Check if the node exists
            if (tempNode == NULL){
                return false;
            }

                // Print the node
            OutFile << setw(6) << tempNode->id;
            OutFile << setw(30) << tempNode->name;
```

```
            OutFile << setw(10) << tempNode->quantityOnHand;
            OutFile << setw(10) << tempNode->quantiyOnOrder << endl;
            linesWritten++;
            return true;
        }
        break;
    }

    if (printingType == 'E'){
        return true;
    }
    else {
        return false;
    }

}
//***********************************  END OF FUNCTION PRINT  ****************************************

//*************************************** FUNTION INSERTNODE  ****************************************
bool InventoryCLASS::InsertNode(NodeType &node)
{
        // Receives - A node
        // Task - Inserts a node into a tree with an "in order" format
        // Returns - A node and a boolean

    NodeType *tempNode = CheckExistance(&node);
    if (tempNode != NULL){
        return false;
    }

    int Inserted = 0;
    NodeType  *ParentNode, *newPtr;
    newPtr = new NodeType();
    ParentNode = Root;

        // Copy the node into a new node to be inserted
    strcpy_s(newPtr->id, node.id);
    strcpy_s(newPtr->name, node.name);
    newPtr->quantityOnHand = node.quantityOnHand;
    newPtr->quantiyOnOrder = node.quantiyOnOrder;
    newPtr->LPtr = NULL;
    newPtr->RPtr = NULL;
    newPtr->Thread = node.Thread;

    if (ParentNode == NULL) {
        Root = newPtr;
        return true;
    }
    while (Inserted != 1)   {
        if (strcmp(node.id, ParentNode->id) <= 0) {
            if (ParentNode->LPtr != NULL) {
                ParentNode = ParentNode->LPtr;
            }
            else  {
                ParentNode->LPtr = newPtr;
                newPtr->RPtr = ParentNode;
                newPtr->Thread = 1;
                Inserted = 1;
            }
        }
        else {
            if (ParentNode->RPtr != NULL && ParentNode->Thread != 1){
                ParentNode = ParentNode->RPtr;
            }
            else  {
                newPtr->RPtr = ParentNode->RPtr;
```

```
                ParentNode->RPtr = newPtr;
                ParentNode->Thread = 0;
                newPtr->Thread = 1;
                Inserted = 1;
            }
        }
    }  /* end while loop  */
    return (Inserted > 0);


}
//***********************************  END OF FUNCTION INSERTNODE  ************************************

//*************************************** FUNTION CHECKEXISTANCE  ************************************
NodeType* InventoryCLASS::CheckExistance(NodeType *mainNode)
{
        // Receives - A node
        // Task - Checks if a node already exists and returns the node if it exists
        // Returns - A node

    int  RightThread;
    NodeType  *CurrPtr;
    NodeType *returningNode = new NodeType();

    CurrPtr = Root;
    if (CurrPtr != NULL) {
        while (CurrPtr->LPtr != NULL) {
            CurrPtr = CurrPtr->LPtr;
        }

        while (CurrPtr != NULL) {
            //   PROCESS THE NODE HERE
            if (strcmp(mainNode->id, CurrPtr->id) == 0) {
                return CurrPtr;
            }

            RightThread = CurrPtr->Thread;
            CurrPtr = CurrPtr->RPtr;
            if ((CurrPtr != NULL) && (RightThread == 0)) {
                while (CurrPtr->LPtr != NULL) {
                    CurrPtr = CurrPtr->LPtr;
                }
            }
        }
    }
    return NULL;
}
//***********************************  END OF FUNCTION CHECKEXISTANCE  ********************************

//*************************************** FUNTION READNODE  ******************************************
void InventoryCLASS::ReadNode(ifstream &InFile, NodeType &node, char command)
{
        // Receives - The input file, the node, and the command character
        // Task - Reads the input data into the node depending on the command character
        // Returns - The input file and the node

        // Read input data depending on the command character
    switch (command){
    case 'I':
            // Read input data
        InFile >> ws;
        InFile.getline(node.id, 6);
        InFile.getline(node.name, 21);
        InFile >> node.quantityOnHand;
        InFile >> node.quantiyOnOrder;
        node.LPtr = NULL;
```

```
        node.RPtr = NULL;
        node.Thread = 0;
        break;
    case 'D':
            // Read input data
        InFile >> ws;
        InFile.getline(node.id, 6);
        InFile.getline(node.name, 21);
        break;
    case 'N':
            // Read input data
        InFile >> ws;
        InFile.getline(node.id, 6);
        break;
    case 'S': case 'O': case 'R':
            // Read input data
        InFile >> ws;
        InFile.getline(node.id, 6);
        InFile >> node.quantityOnHand;
        break;
    }


}
//***********************************  END OF FUNCTION READNODE  ***************************************

//*************************************** FUNTION PAGEBREAK  ***************************************
void PageBreak(ofstream &Outfile, int &limit)
{
        // Receives - The output file and the amount of lines written in the current page.
        // Task - Add end lines to the output file.
        // Returns -  The output file and the amount of lines written in the current page.

        // Calculate amount of blank lines needed for new page
    limit = LinesPerPage - limit;

        // Print blank lines
    for (int i = 0; i < limit; i++){
        Outfile << endl;
    }
        // Reset amount of lines writen in one page
    limit = 0;
}
//***********************************  END OF FUNCTION PAGEBREAK  ***************************************

//*************************************** FUNCTION HEADER  ***************************************
void Header(ofstream &Outfile)
{
        // Receives - The output file
        // Task - Prints the output preamble
        // Returns - The output file

    Outfile << setw(45) << "Adrian Beloqui ";
    Outfile << setw(15) << "CSC 36000";
    Outfile << setw(15) << "Section 11" << endl;
    Outfile << setw(50) << "Spring 2017";
    Outfile << setw(20) << "Assignment #7" << endl;
    Outfile << setw(35) << "-----------------------------------------------";
    Outfile << setw(35) << "-------------------------------------------------------" << endl;
    return;
}
//***********************************  END OF FUNCTION HEADER  ***************************************

//*************************************** FUNCTION FOOTER  ***************************************
void Footer(ofstream &Outfile)
{
```

```
        // Receives - The output file
        // Task - Prints the output salutation
        // Returns - The output file

    Outfile << endl;
    Outfile << setw(35) << "-----------------------------------------------" << endl;
    Outfile << setw(35) << "|               END OF PROGRAM OUTPUT          |" << endl;
    Outfile << setw(35) << "-----------------------------------------------" << endl;
    return;
}
//**********************************  END OF FUNCTION FOOTER  ****************************************
```