```cpp
#include "SymbolTable.h"

LinkedList::LinkedList()
{
    ptrToFirst = NULL;
}

void LinkedList::insert(const Data& insertedData, int scope)
{
    Node *index = ptrToFirst; // node to look for any previous declarations of that variable
    Node *node = new Node(insertedData, scope, NULL); // the new node to be inserted
    if (index == NULL) // skips the loop if there is no nodes in the list
        ptrToFirst = node;
    else
    {
        while (index->getNext() != NULL && !(index->getData().name == insertedData.name && index->getScope() == scope)) // looks for a node identical to the one to be inserted
            index = index->getNext();
        if (!(index->getData().name == insertedData.name && index->getScope() == scope)) // if there are no identical nodes, insert the new one in the front
        {
            node->setNext(ptrToFirst);
            ptrToFirst = node;
        }
    }
}

Node* LinkedList::search(const string& searchTerm) // returns the node of the searched term
{
    if (ptrToFirst == NULL)
        return NULL;
    else
    {
        Node *node = ptrToFirst;
        while (node != NULL && node->getData().name != searchTerm)
            node = node->getNext();
        return node;
    }
}

bool LinkedList::empty() // returns if empty
{
    return (ptrToFirst == NULL);
}

void LinkedList::print() const // couts any information in the list
{
    Node *node = new Node();
    // do some printing function in the style of whatever way your professor asks...
    if (ptrToFirst != NULL)
    {
        node = ptrToFirst;
        cout << "( " << node->getData().name << ", " << node->getScope() << " ) ";
        while (node->getNext() != NULL)
        {
            node = node->getNext();
            cout << "( " << node->getData().name << ", " << node->getScope() << " ) ";
        }
    }
}

void LinkedList::del(int scope) // deletes a node from the list and adds it to a temporary stack that will be added to a larger stack later
{
    Node *node = ptrToFirst;
    Node *trailingNode = new Node();
    while (node != NULL)
    {
        while (node != NULL && node->getScope() != scope)
        {
            trailingNode = node;
            node = node->getNext();
        }

        if (node != NULL)
        {
            //deletes the item from the list and pushes it onto the stack
            if (node == ptrToFirst && node->getNext() == NULL)
            {
                ptrToFirst = NULL;
                tempStack.push(node->getData());
                node = NULL;
            }
            else if (node == ptrToFirst && node->getNext() != NULL)
            {
                ptrToFirst = node->getNext();
                tempStack.push(node->getData());
                node = ptrToFirst;
            }
            else if (node != ptrToFirst && node->getNext() == NULL)
            {
                trailingNode->setNext(NULL);
                tempStack.push(node->getData());
                node = NULL;
            }
            else if (node != ptrToFirst && node->getNext() != NULL)
            {
                node = node->getNext();
                trailingNode->setNext(node);
                tempStack.push(node->getData());
            }
        }
        //one item has been deleted or there was nothing to delete in the list
    }
    //there's nothing else to delete
}



HashTable::HashTable() // default consturctor
{
    for (int i = 0; i < 23; i++)
        table[i] = new LinkedList();
    localScope = 0;
}

int HashTable::convertString(const string& strToConvert) // makes a hash value out of the string
{
    int sum = 0;
    int size = strToConvert.size();
    for (int i = 0; i < size; i++)
        sum += strToConvert[i];

    int key = (sum + 255) % 23;
    return key;
}

void HashTable::insert(const Data& data) // inserts the string into the appropriate list
{
    table[convertString(data.name)]->insert(data, localScope);
}

void HashTable::print() // prints all the contents of the stack
{
    ofstream table;
    table.open("symboltable.txt");
    while (localScope > 0)
        closeScope();
    while (!obsoleteVariables.empty())
    {
        table << obsoleteVariables.top().type << " " << obsoleteVariables.top().name << " " << obsoleteVariables.top().address << ", ";
        obsoleteVariables.pop();
    }
    table.close();
}
```

```cpp

Data* HashTable::find(const string& searchString) // returns the string searched for if found otherwise returns "Not Found"
{
    int a = convertString(searchString);
    if (table[a]->search(searchString) != NULL)
    {
        string type = table[a]->search(searchString)->getData().type;
        string name = table[a]->search(searchString)->getData().name;
        int addr = table[a]->search(searchString)->getData().address;
        Data *data = new Data({type, name, addr});
        return data;
    }
    else
    {
        Data *data = new Data({ "NULL", "Not Found", -1 });
        return data;
    }
}

void HashTable::newScope() // increments localScope
{
    localScope++;
}

void HashTable::closeScope() // decriments localScope and adds all the temporary stacks onto the big stack
{
    if (localScope > 0)
    {
        for (int i = 0; i < 23; i++)
        {
            table[i]->del(localScope);
            while (!table[i]->getStack().empty())
            {
                obsoleteVariables.push(table[i]->getStack().top());
                table[i]->remove();
            }
        }
        localScope--;
    }

}
```