#include "Gaddis\_17\_1\_YourOwnLL\_Specification.h"

#include <iostream>

#include <cstdlib>

namespace MyLL\_Ints

{

int MyLL::FindListLength(Node\* head)

{

int length = 0;

while (head != nullptr)

{

++length;

head = head->link;

}

return length;

}

bool MyLL::IsSortedUp(Node\* head)

{

if (head == nullptr || head->link == nullptr) // empty LL or 1-node LL

return true;

while (head->link != nullptr)

{

if (head->link->data < head->data)

return false;

head = head->link;

}

return true;

}

void MyLL::InsertAsHead(Node\*& head, int value)

{

Node\* newNodePtr = new Node; // create a new ptr

newNodePtr->data = value; // store value in newNodePtr

newNodePtr->link = head; // set next link to head? \*\*\*\*\*\*\*\*\*\*\*\*\*

head = newNodePtr; // set head to new ptr

}

void MyLL::InsertAsTail(Node\*& head, int value)

{

Node\* newNodePtr = new Node; // create a new ptr

newNodePtr->data = value; // set newNodePtr to value

newNodePtr->link = nullptr; // find last node

if (head == nullptr)

head = newNodePtr;

else

{

Node\* nodePtr = head;

while (nodePtr->link != nullptr) // while not at last node

nodePtr = nodePtr->link; // advance to the next link in the LL

nodePtr->link = newNodePtr;

}

}

void MyLL::InsertSortedUp(Node\*& head, int value)

{

Node \*pre = nullptr, \*nodePtr = head;

while (nodePtr != nullptr && nodePtr->data < value)

{

pre = nodePtr; // swap

nodePtr = nodePtr->link;

}

Node\* newNodePtr = new Node;

newNodePtr->data = value;

newNodePtr->link = nodePtr;

if (nodePtr == head)

head = newNodePtr;

else

pre->link = newNodePtr;

}

bool MyLL::DelFirstTargetNode(Node\*& head, int target) // FIX THIS!

{

Node \*pre = nullptr, \*nodePtr = head;

while (nodePtr != nullptr && nodePtr->data != target)

{

pre = nodePtr;

nodePtr = nodePtr->link;

}

if (nodePtr == nullptr)

{

std::cout << std::endl << "Target valeue " << target << " not found."

<< std::endl;

return false;

}

if (nodePtr == head) // OR: pre == nullptr

head = head->link;

else

pre->link = nodePtr->link;

delete nodePtr;

nodePtr = nullptr;

return true;

}

bool MyLL::DelNodeBefore1stMatch(Node\*& head, int target)

{

if (head == nullptr || head->link == nullptr || head->data == target)

return false;

Node \*nodePtr = head->link, \*pre = head, \*prepre = nullptr;

while (nodePtr != nullptr && nodePtr->data != target)

{

prepre = pre;

pre = nodePtr;

nodePtr = nodePtr->link;

}

if (nodePtr == nullptr) return false;

if (nodePtr == head->link)

{

head = nodePtr;

delete pre;

}

else

{

prepre->link = nodePtr;

delete pre;

}

return true;

}

void MyLL::ShowAll(std::ostream& outs, Node\* head)

{

while (head != nullptr)

{

outs << head->data << " ";

head = head->link;

}

outs << std::endl;

}

void MyLL::FindMinMax(Node\* head, int& minValue, int& maxValue)

// pass by reference (for some reason)

{

if (head == nullptr)

{

std::cerr << std::endl

<< "FindMinMax() attempted on empty list."

<< std::endl

<< "Minimum and maximum values not set."

<< std::endl;

}

else

{

minValue = maxValue = head->data;

while (head->link != nullptr)

{

head = head->link; // advance head to next link

if (head->data < minValue) // if head data is less than

// minValue, set new minValue

// to head data

minValue = head->data;

else if ( head->data > maxValue) // if head data is greater than

// maxValue, set new maxValue

// to head data

maxValue = head->data;

}

std::cout << minValue << " " << maxValue << std::endl;

}

}

double MyLL::FindAverage(Node\* head)

{

if (head == nullptr)

{

std::cerr << std::endl

<< "FindAverage() attempted on empty list."

<< std:: endl

<< "An arbitrary zero value is returned."

<< std::endl;

return 0.0;

}

else

{

int sum = 0, count = 0;

while (head != nullptr)

{

++count;

sum += head->data;

head = head->link;

}

return static\_cast<double>(sum) / count;

}

}

void MyLL::SortLinkedList(Node\* head)

{

Node \*nodePtr = head;

int tempData, count = 0;

while (nodePtr)

{

nodePtr = nodePtr->link;

++count;

}

nodePtr = head;

for (int i = 0; i < count; ++i)

{

while (nodePtr->link) //iterate through list until link is null

{

if (nodePtr->data > nodePtr->link->data)

{

tempData = nodePtr->data;

nodePtr->data = nodePtr->link->data;

nodePtr->link->data = tempData;

}

else

nodePtr = nodePtr->link; //increment node

}

nodePtr = head;

}

}

void MyLL::PromoteTarget(Node\*& head, int target)

{

bool targetIsThere = false;

Node \*newNode = new Node; // Create a new node, newNode, for when

// it's needed.

newNode->data = target; // Store target in newNode.

newNode->link = nullptr; // Make newNode point to the nullptr.

if (!head) // If there is not a headPtr, create one with target.

{

head = newNode;

return;

}

if (head->link == nullptr) // It there is a single node . . .

{

if (head->data != target) // . . . and it is not the target,

{ // append newNode containing target

// just after the first node.

Node\* nodePtr = head;

while (nodePtr->link) // nodePtr is an index pointer

// designed for traversing the LL.

nodePtr = nodePtr->link;

nodePtr->link = newNode; // Append a newNode to single node.

}

}

else // Otherwise, there is more than one node in the LL

{

Node \*previousNode = head;

Node \*nodePtr = new Node; // Index pointer

nodePtr = head->link;

while (nodePtr) // Search the LL until arrive at the nullptr.

{

if (previousNode->data == target) // If the first datum is

// target, target is there,

// and we're done.

targetIsThere = true;

if (nodePtr->data == target) // Move targets to the front

// of the LL.

{

previousNode->link = nodePtr->link;

nodePtr->link = head;

head = nodePtr;

nodePtr = previousNode->link;

targetIsThere = true;

}

else if (nodePtr->data != target) // Keep traveling through

// the nodes while not

// target.

{

previousNode = nodePtr;

nodePtr = nodePtr->link;

}

}

if (!targetIsThere) // Append node to the end of the LL IF

// target is NOT in the LL.

{

Node\* nodePtr = head;

while (nodePtr->link) // Traverse the LL until end.

nodePtr = nodePtr->link;

nodePtr->link = newNode; // Append newNode containing target

}

}

}

void MyLL::ListClear(Node\*& head, int noMsg)

{

int count = 0;

Node\* nodePtr = head;

while (head != nullptr)

{

head = head->link;

delete nodePtr;

nodePtr = head;

++count;

}

if (noMsg)

return;

std::clog << "Dynamic memory for " << count << " nodes freed."

<< std::endl;

}

}