// Austin Faulkner: May 24, 2020

#include <iostream>

#include <fstream>

#include <iomanip>

namespace My\_Templated\_LL

{

template<class T>

int LinkedList<T>::FindListLength(ListNode<T>\* head)

{

int length = 0;

while (head != nullptr)

{

++length;

head = head->next;

}

return length;

}

template <class T>

bool LinkedList<T>::IsSortedUp(ListNode<T>\* head)

{

if (head == nullptr || head->next == nullptr)

return true;

while (head->next != nullptr)

{

if (head->next->data.getGPA() < head->data.getGPA())

return false;

head = head->next;

}

return true;

}

template <class T>

void LinkedList<T>::InsertAsHead(ListNode<T>\*& head, T data)

{

ListNode<T>\* newNodePtr = new ListNode<T>(data);

newNodePtr->data = data;

newNodePtr->next = head;

head = newNodePtr;

}

template <class T>

void LinkedList<T>::InsertAsTail(ListNode<T>\*& head, T newValue)

{

ListNode<T> \*newNode; // To point to a new node

ListNode<T> \*nodePtr; // To move through the list

// Allocate a new node and store newValue there.

newNode = new ListNode<T>(newValue);

// If there are no nodes in the list

// make newNode the first node.

if (!head)

head = newNode;

else // Otherwise, insert newNode at end.

{

// Initialize nodePtr to head of list.

nodePtr = head;

// Find the last node in the list.

while (nodePtr->next)

nodePtr = nodePtr->next;

// Insert newNode as the last node.

nodePtr->next = newNode;

}

}

template <class T>

void LinkedList<T>::InsertSortedUp(ListNode<T>\*& head, T data, double value)

{

ListNode<T> \*previousNode = nullptr, \*nodePtr = head;

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

{

previousNode = nodePtr;

nodePtr = nodePtr->next;

}

ListNode<T>\* newNodePtr = new ListNode<T>(data);

newNodePtr->data.setGPA(value);

newNodePtr->next = nodePtr;

if (nodePtr == head)

head = newNodePtr;

else

previousNode->next = newNodePtr;

}

//------------------------------------------------------------------------------

// Delete Methods, DeleteFirstTargetNode() & DeleteNode(), Below

//------------------------------------------------------------------------------

template <class T>

bool LinkedList<T>::DeleteFirstTargetNode(ListNode<T>\*& head, T target)

{

ListNode<T> \*previousNode = nullptr, \*nodePtr = head;

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

{

previousNode = nodePtr;

nodePtr = nodePtr->next;

}

if (nodePtr == nullptr)

{

std::cout << std::endl

<< "Target value " << target << " not found."

<< std::endl;

return false;

}

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

head = head->next;

else

previousNode->next = nodePtr->next;

delete nodePtr;

nodePtr = nullptr;

return true;

}

template <class T>

void LinkedList<T>::DeleteNode(T target) // OR: Needs to be fixed

{

ListNode<T> \*nodePtr;

ListNode<T> \*previousNode;

// If the list is empty, do nothing.

if (!head)

{

std::cout << std::endl << "The roster is empty." << std::endl;

return;

}

if (head->data == target)

{

nodePtr = head->next;

delete head;

head = nodePtr;

}

else

{

nodePtr = head;

// Skip all nodes whose data member is

// not equal to target.

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

{

previousNode = nodePtr;

nodePtr = nodePtr->next;

}

if (nodePtr)

{

previousNode->next = nodePtr->next;

delete nodePtr;

}

}

}

//------------------------------------------------------------------------------

template <class T>

void LinkedList<T>::ShowAll(std::ostream& outs, ListNode<T>\* head)

{

while (head != nullptr)

{

std::cout << head->data << " ";

head = head->next;

}

outs << std::endl;

}

template <class T>

void LinkedList<T>::FindMinMax(ListNode<T>\* head, double& min, double& max)

{

if (head == nullptr)

{

std::cerr << std::endl

<< "Method FindMinMax() attempted on an empty linked list."

<< std::endl

<< "Minimum and Maximum GPAs not set."

<< std::endl;

}

else

{

min = max = head->data.gpa;

while (head->next != nullptr)

{

head = head->next;

if (head->data.gpa < min)

min = head->data.gpa;

else if (head->data.gpa > max)

max = head->data.gpa;

}

std::cout << "Minimum & Maximum GPA: "

<< min << " " << max << std::endl;

}

}

template <class T>

double LinkedList<T>::FindAverage(ListNode<T>\* head)

{

if (head == nullptr)

return 0.00;

else

{

double sum = 0;

int count = 0;

while (head != nullptr)

{

++count;

sum += head->data.gpa;

head = head->next;

}

return sum / count;

}

}

//------------------------------------------------------------------------------

// How can I better set SortLinkedList() so that all StudentData attributes

// can still be private?

//------------------------------------------------------------------------------

template <class T>

void LinkedList<T>::SortLinkedList(ListNode<T>\* head)

{

ListNode<T> \*nodePtr = head;

std::string tempFN,

tempMN,

tempLN,

tempID,

tempMaj;

double tempGPA;

std::string tempRes,

tempStNum,

tempStName,

tempDrStCt,

tempAptNum,

tempCity,

tempState;

int count = 0, tempZip;

while (nodePtr)

{

nodePtr = nodePtr->next;

++count;

}

nodePtr = head;

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

{

while (nodePtr->next)

{

if (nodePtr->data.getGPA() > nodePtr->next->data.getGPA())

{

tempFN = nodePtr->data.studentFirstName;

nodePtr->data.studentFirstName =

nodePtr->next->data.studentFirstName;

nodePtr->next->data.studentFirstName = tempFN;

tempMN = nodePtr->data.studentMiddleName;

nodePtr->data.studentMiddleName =

nodePtr->next->data.studentMiddleName;

nodePtr->next->data.studentMiddleName = tempMN;

tempLN = nodePtr->data.studentLastName;

nodePtr->data.studentLastName =

nodePtr->next->data.studentLastName;

nodePtr->next->data.studentLastName = tempLN;

tempID = nodePtr->data.studentId;

nodePtr->data.studentId = nodePtr->next->data.studentId;

nodePtr->next->data.studentId = tempID;

tempMaj = nodePtr->data.studentMajor;

nodePtr->data.studentMajor = nodePtr->next->data.studentMajor;

nodePtr->next->data.studentMajor = tempMaj;

tempGPA = nodePtr->data.gpa;

nodePtr->data.gpa = nodePtr->next->data.gpa;

nodePtr->next->data.gpa = tempGPA;

tempRes = nodePtr->data.residency;

nodePtr->data.residency = nodePtr->next->data.residency;

nodePtr->next->data.residency= tempRes;

tempStNum = nodePtr->data.streetNumber;

nodePtr->data.streetNumber = nodePtr->next->data.streetNumber;

nodePtr->next->data.streetNumber = tempStNum;

tempStName = nodePtr->data.streetName;

nodePtr->data.streetName = nodePtr->next->data.streetName;

nodePtr->next->data.streetName = tempStName;

tempDrStCt = nodePtr->data.driveStreetCourt;

nodePtr->data.driveStreetCourt =

nodePtr->next->data.driveStreetCourt;

nodePtr->next->data.driveStreetCourt = tempDrStCt;

tempAptNum = nodePtr->data.apartmentNum;

nodePtr->data.apartmentNum = nodePtr->next->data.apartmentNum;

nodePtr->next->data.apartmentNum = tempAptNum;

tempCity = nodePtr->data.city;

nodePtr->data.city = nodePtr->next->data.city;

nodePtr->next->data.city = tempCity;

tempState = nodePtr->data.state;

nodePtr->data.state = nodePtr->next->data.state;

nodePtr->next->data.state = tempState;

tempZip = nodePtr->data.zipcode;

nodePtr->data.zipcode = nodePtr->next->data.zipcode;

nodePtr->next->data.zipcode = tempZip;

}

else

nodePtr = nodePtr->next;

}

nodePtr = head;

}

}

//------------------------------------------------------------------------------

template <class T>

void LinkedList<T>::PromoteTarget(ListNode<T>\*& head, T target)

{

bool targetIsThere = false;

ListNode<T>\* newNode = new ListNode<T>(target);

newNode->data = target;

newNode->next = nullptr;

if (!head) // If there is not a head pointer, create one with newNode

{

head = newNode;

return;

}

if (head->next == nullptr) // If there is a single node . . .

{

if (head->data != target)

{

ListNode<T>\* nodePtr = head;

while (nodePtr->next)

nodePtr = nodePtr->next;

nodePtr->next = newNode;

}

}

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

{

ListNode<T>\* previousNode = head;

ListNode<T>\* nodePtr = new ListNode<T>(target);

nodePtr = head->next;

while (nodePtr)

{

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

{ // target, target is there

// and do nothing.

targetIsThere = true;

return;

}

if (nodePtr->data == target)

{

previousNode->next = nodePtr->next;

nodePtr->next = head;

head = nodePtr;

nodePtr = previousNode->next;

targetIsThere = true;

}

else if (nodePtr->data != target)

{

previousNode = nodePtr;

nodePtr = nodePtr->next;

}

}

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

// target is NOT in the LL;

{

ListNode<T>\* nodePtr = head;

// <--------- WORK ON THIS PART

while (nodePtr->next)

nodePtr = nodePtr->next;

nodePtr->next = newNode;

}

}

}

template <class T>

void LinkedList<T>::ListClear(ListNode<T>\*& head, int noMsg)

{

int count = 0;

ListNode<T>\* nodePtr = head;

while (head != nullptr)

{

head = head->next;

delete nodePtr;

nodePtr = head;

++count;

}

if (noMsg)

return;

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

<< std::endl;

}

//------------------------------------------------------------------------------

// File-Read-Into-LL Method, ReadStudentRecords(), Below

//------------------------------------------------------------------------------

template <class T>

void LinkedList<T>::ReadStudentRecords(std::ifstream& fin,

std::istream& ins, T data)

{

std::string file\_name;

std::cout << "Which student file would you like to read in? ";

ins >> file\_name;

std::cout << std::endl;

fin.open(file\_name, std::ifstream::in);

std::string str = " ";

if (!fin)

{

std::cerr << "Cannot find file."

<< std::endl;

}

std::cout << "Student records read from "

<< "\'" << file\_name << "\'."

<< std::endl << std::endl << std::endl;

while (std::getline(fin, str)) // WILL PROBABALY PRESENT A PROBLEM

// WHEN TRYING TO SORT LL BY GPAs.

{

std::cout << str << std::endl;

}

fin.close();

}

//------------------------------------------------------------------------------

template <class T>

void LinkedList<T>::PrintStudentRecords(std::ofstream& fout,

ListNode<T>\* head)

{

while (head != nullptr)

{

fout << head->data << " ";

head = head->next;

}

fout << std::endl;

}

template <class T>

LinkedList<T>::~LinkedList()

{

ListNode<T> \*nodePtr; // To traverse the list

ListNode<T> \*nextNode; // To point to the next node

// Position nodePtr at the head of the list.

nodePtr = head;

// While nodePtr is not at the end of the list...

while (nodePtr != nullptr)

{

// Save a pointer to the next node.

nextNode = nodePtr->next;

// Delete the current node.

delete nodePtr;

// Position nodePtr at the next node.

nodePtr = nextNode;

}

}

}