|  |  |  |
| --- | --- | --- |
| **American University of Sharjah**  **School of Engineering**  Computer Engineering Department  P. O. Box 26666 Sharjah, UAE |  | **Instructor:** Dr. Ghassan Z. Qadah  **Lab Instructor:** Ms.Praveena Kolli  **Office**: EB2-126  **Phone**: 971-6-515-2352  **e-mail**: pkolli@aus.edu  **Semester**: Spring 2019 |

**CMP305L Data Structures and Algorithms**

Objectives:

* To understand and implement sorted linear and linked list
* To understand processing of sorted linear and linked list

**Exercise 1**

Abstract classes for **StudentType** and **SortedType** are given. Implement the **StudentType** and **SortedType** classes. **SortedType** list **must** be implemented using **linked structure**.

enum RelationType { LESS, EQUAL, GREATER } ;

class StudentType // declares class data type

{

public :

RelationType ComparedTo (StudentType ) const ;

void Print ( ) const ;

void Initialize ( int, string, string, string ) ;

int Getid();

private :

int id ;

string firstName;

string lastName;

string grade;

} ;

// SPECIFICATION FILE ( unsorted.h )

#include “StudentType.h”

enum ERROR\_CODE { SUCCESS, FAIL, UNDERFLOW, OVERFLOW, NOT\_FOUND } ;

class SortedType *// declares a class data type*

{

public : *// 8 public member functions*

SortedType ( );

void MakeEmpty ( );

bool IsFull ( ) const ;

int LengthIs ( ) const ; *// returns length of list*

ERROR\_CODE RetrieveItem ( Student& item, bool& found ) ;

ERROR\_CODE InsertItem ( Student item ) ;

ERROR\_CODE DeleteItem ( Student item ) ;

void ResetList ( );

ERROR\_CODE GetNextItem ( Student& item ) ;

private : *// 3 private data members*

int length ;

NodeType\* ListData ;

NodeType\* currentPos ;

} ;

**Exercise 2.a**

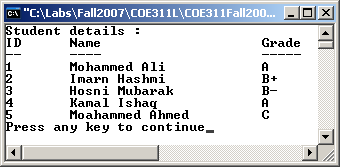
Use StudentType class to create an instance for each student whose details are given in a file by name **student.txt.**

Initialize a **SortedType** list with the instances that you have created for all the students. Implement and print all the students’ details.

**Sample Input:**



**Sample Output:**



**Exercise 2.b**

Modify the program code for **exercise 2.a** to list all the students that have grade **A.**

**Sample output:**



#include<iostream>

#include<fstream>

using namespace std;

enum RelationType { LESS, EQUAL, GREATER } ;

class StudentType // declares class data type

{

public :

RelationType ComparedTo (StudentType s) const {

return id == s.Getid() ? EQUAL : id < s.Getid() ? LESS : GREATER;

}

void Print ( ) const {cout << id << "\t" << firstName << " " << lastName << "\t" << grade;}

void Initialize ( int a, string b, string c , string d){

id = a; firstName = b; lastName = c; grade = d;

}

int Getid(){return id;} //I hate the way this is capitalized

string getGrade(){return grade;}

private :

int id ;

string firstName;

string lastName;

string grade;

};

#include"StudentType.cpp"

enum ERROR\_CODE { SUCCESS, FAIL, UNDERFLOW, OVERFLOW, NOT\_FOUND } ; //I hate ERROR\_CODE

struct NodeType{

StudentType info;

NodeType\* next;

};

class SortedType // declares a class data type

{

public : // 8 public member functions

SortedType ( );

SortedType(const SortedType& original);

SortedType& operator=(const SortedType& original);

~SortedType( );

void MakeEmpty ( );

bool IsEmpty() const; //I hate ThisNamingConvention

bool IsFull ( ) const ;

int LengthIs ( ) const ; // returns length of list

ERROR\_CODE RetrieveItem ( StudentType& item, bool& found ) ; //This is just stupid

ERROR\_CODE InsertItem ( StudentType item ) ;

ERROR\_CODE DeleteItem ( StudentType item ) ;

void ResetList ( );

ERROR\_CODE GetNextItem ( StudentType& item ) ;

private : // 3 private data members

int length ;

NodeType\* listData ;

NodeType\* currentPos ;

} ;

SortedType::SortedType ( ){

length = 0 ;

listData = nullptr; //prefer nullptr over null in newer code

currentPos = new NodeType;

//I want to use smart pointers but am afraid I'll get penalized :(

}

SortedType::~SortedType(){

MakeEmpty();

}

bool SortedType::IsEmpty() const {return listData == nullptr;}

SortedType& SortedType::operator=(const SortedType& original){

if(this->listData != original.listData){

MakeEmpty();

listData = nullptr;

length = 0;

currentPos = new NodeType;

NodeType\* tmp = original.listData;

while(tmp){

InsertItem(tmp->info);

tmp = tmp->next;

}

}

return \*this;

}

SortedType::SortedType(const SortedType& original){

listData = nullptr;

length = 0;

NodeType\* tmp = original.listData;

currentPos = new NodeType;

while(tmp){

InsertItem(tmp->info);

tmp = tmp->next;

}

}

void SortedType::MakeEmpty ( ){

if(!IsEmpty()){

while(listData){

DeleteItem(listData->info);

}

delete currentPos;

}

}

bool SortedType::IsFull ( ) const {

NodeType\* location;

try{

location = new NodeType;

delete location;

return false;

}catch(std::bad\_alloc exception){

return true;

}

}

int SortedType::LengthIs ( ) const {return length;} // returns length of list

ERROR\_CODE SortedType::RetrieveItem ( StudentType& item, bool& found ) {

//Normally I would take advantage of the fact that the list is sorted and do a binary search. However, you can't index into arbitrary elements in lists. Sooooo, here's linear search.

NodeType\* location = listData ;

found = false ;

while (location && !found ){

if (item.ComparedTo(location->info) == EQUAL){

found = true; // match found

item = location->info;

}

else // advance pointer

location = location->next;

}

return NOT\_FOUND;

}

ERROR\_CODE SortedType::InsertItem ( StudentType item ) {

if(IsFull()){return OVERFLOW;}

NodeType\* location = listData;

NodeType\* predLoc = nullptr;

NodeType\* newNode;

//find insertion point.

while (location && item.ComparedTo(location->info) == GREATER) {

predLoc = location;

location = location->next;

}

// create a node

newNode = new NodeType;

newNode -> info = item;

// insert node into list:

if (predLoc == nullptr){

listData = newNode;

}

else {

predLoc -> next = newNode;

}

newNode -> next=location;

length++;

return SUCCESS;

}

ERROR\_CODE SortedType::DeleteItem ( StudentType item ) {

if(IsEmpty()){return UNDERFLOW;}

NodeType\* location = listData;

NodeType\* tmp;

// locate node to be deleted

if (item.ComparedTo(listData -> info) == EQUAL){

tmp = listData;

listData = listData ->next;

}

else {

while ((item.ComparedTo((location -> next)->info)) != EQUAL)

location = location -> next;

// delete node at location -> next

tmp = location -> next;

location -> next = (location -> next) -> next;

}

if(tmp){

delete tmp;

return SUCCESS;

}else{

return NOT\_FOUND;

}

length--;

}

void SortedType::ResetList ( ){

if(listData){ //prevent dereferencing of null pointer

currentPos->next = listData->next;

currentPos->info = listData->info;

}

}

ERROR\_CODE SortedType::GetNextItem ( StudentType& item ){

if(currentPos){ //prevent dereferencing of null pointer

item = currentPos->info;

currentPos = currentPos->next;

return SUCCESS;

}else{

return FAIL;

}

}

int main(){

SortedType a;

ifstream in("student.txt");

if(in.is\_open()){

string i, f, l, g;

while(in >> i >> f >> l >> g){

StudentType s;

s.Initialize(stoi(i),f,l,g);

a.InsertItem(s);

}

}else{

cerr << "Failed to open file" << endl;

exit(1);

}

cout << "Student details: \n ID \t Name \t\t Grade \n \_\_ \t \_\_\_\_\t\t\_\_\_\_\_\n";

StudentType b;

a.ResetList();

while(a.GetNextItem(b) != FAIL){

if(b.getGrade() == "A"){

b.Print(); cout << endl;

}

}

in.close();

return 0;

}

