

Qatar University

College of Engineering

Department of computer science and engineering

Fall\_2020

cmps 303 – data structure

14877\_B57

Instructor: Dr. Saleh Al-Hazbi

Lab Project

November 20, 2020

Marim Elhanafy 201803468

Hagar Elsayed 201805123

Mai Saed 201803508

Some snapshots of the output of the code:

Graphical user interface, text, application

Description automatically generatedText, table

Description automatically generated

**Codes:**

**DoublyLinkedList Class:**

**package** model;

/\*\*

\* **@author** Mai Saed 201803508

\*

\*/

**public** **class** DoublyLinkedList<E> {

**private** NodeDoublyLinkedList<E> header;

**private** NodeDoublyLinkedList<E> trailer;

**private** **int** size = 0;

/\*\*

\* Constructor to initialize fields when creating a new doubly linked list

\* - it should have the two sentinel nodes (header/trailer) and they points to each other.

\*/

**public** DoublyLinkedList() {

header = **new** NodeDoublyLinkedList<>(**null**, **null**, **null**);

trailer = **new** NodeDoublyLinkedList<>(**null**, header, **null**);

header.setNext(trailer);

}

**public** NodeDoublyLinkedList<E> getHeader() {

**return** header;

}

**public** NodeDoublyLinkedList<E> getTrailer() {

**return** trailer;

}

/\*\*

\* **@return** size of the doubly linked list

\*/

**public** **int** size() {

**return** size;

}

/\*\*

\* **@return** true (if the doubly linked list is empty)

\* false (if not empty)

\*/

**public** **boolean** isEmpty() {

**return** size == 0;

}

/\*\*

\* **@return** first node in the doubly linked list if it exists

\* or null if the doubly linked list is empty

\*/

**public** E first() {

**if** (isEmpty())

**return** **null**;

**return** header.getNext().getData();

}

/\*\*

\* **@return** last node in the doubly linked list if it exists

\* or null if the doubly linked list is empty

\*/

**public** E last() {

**if** (isEmpty())

**return** **null**;

**return** trailer.getPrev().getData();

}

/\*\*

\* **@param** e: data (generic data type)

\* **@param** predecessor: previous node

\* **@param** successor: next node

\*/

**private** **void** addBetween(E e, NodeDoublyLinkedList<E> predecessor, NodeDoublyLinkedList<E> successor) {

NodeDoublyLinkedList<E> newest = **new** NodeDoublyLinkedList<>(e, predecessor, successor);

predecessor.setNext(newest);

successor.setPrev(newest);

size++;

}

/\*\*

\* **@param** node: that we want to remove

\* **@return** the node that removed and its data

\*/

**private** E remove(NodeDoublyLinkedList<E> node) {

NodeDoublyLinkedList<E> predecessor = node.getPrev();

NodeDoublyLinkedList<E> successor = node.getNext();

predecessor.setNext(successor);

successor.setPrev(predecessor);

size--;

**return** node.getData();

}

/\*\*

\* **@param** e: data (generic data type)

\* -to add between header and the first node or between header and trailer)

\*/

**public** **void** addFirst(E e) {

addBetween(e, header, header.getNext());

}

/\*\*

\* **@param** e: data (generic data type)

\* -to add between trailer and the last node or between trailer and header)

\*/

**public** **void** addLast(E e) {

addBetween(e, trailer.getPrev(), trailer);

}

/\*\*

\* **@return** first node if it exists after removing

\* or null if the doubly linked list is empty

\*/

**public** E removeFirst() {

**if** (isEmpty())

**return** **null**;

**return** remove(header.getNext());

}

/\*\*

\* **@return** last node if it exists after removing

\* or null if the doubly linked list is empty

\*/

**public** E removeLast() {

**if** (isEmpty())

**return** **null**;

**return** remove(trailer.getPrev());

}

@Override

**public** String toString() {

String string="";

**for** (NodeDoublyLinkedList<E> tmp = header.getNext(); tmp != trailer; tmp = tmp.getNext())

string+="\n"+tmp.getData().toString();

string+="\n";

**return** string;

}

}

**NodeDoublyLinkedList Class:**

**package** model;

/\*\*

\* **@author** Mai Saed 201803508

\*

\*/

**public** **class** NodeDoublyLinkedList <E>{

**private** E data;

**private** NodeDoublyLinkedList<E> prev;

**private** NodeDoublyLinkedList<E> next;

/\*\*

\* Constructor to initialize fields

\*/

**public** NodeDoublyLinkedList(E d, NodeDoublyLinkedList<E> p,NodeDoublyLinkedList<E> n)

{

data=d;

prev=p;

next=n;

}

**public** E getData() {

**return** data;

}

**public** NodeDoublyLinkedList<E> getNext(){

**return** next;

}

**public** NodeDoublyLinkedList<E> getPrev(){

**return** prev;

}

**public** **void** setNext(NodeDoublyLinkedList<E> n) {

next=n;

}

**public** **void** setPrev(NodeDoublyLinkedList<E> p) {

prev=p;

}

}

**NodeTree Class:**

**package** model;

/\*\*

\* **@author** Hagar Elsayed 201805123

\*

\*/

**public** **class** NodeTree<E> {

**int** key;

E data;

NodeTree<E> leftChild;

NodeTree<E> rightChild;

/\*\*

\* Constructor to initialize fields

\*/

**public** NodeTree(**int** k, E e) {

key = k;

data = e;

leftChild = **null**;

rightChild = **null**;

}

/\*\*

\* display the nodes key and data

\*/

**public** **void** display() {

System.***out***.print(key + ":");

System.***out***.println(data);

}

}

**Student Class:**

**package** model;

**import** java.util.ArrayList;

/\*\*

\* **@author** Marim Elhanafy 201803468

\*

\*/

**public** **class** Student {

**private** **long** id;

**private** String name;

ArrayList<Subject> subjects = **new** ArrayList<Subject>();

/\*\*

\* Constructor to initialize fields

\*/

**public** Student(**long** id, String name) {

**this**.id = id;

**this**.name = name;

}

**public** **long** getId() {

**return** id;

}

**public** **void** setId(**long** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** ArrayList<Subject> getSubjects() {

**return** subjects;

}

**public** **void** setSubjects(ArrayList<Subject> subjects) {

**this**.subjects = subjects;

}

@Override

**public** String toString() {

**return** id + " " + name;

}

}

**Subject Class:**

**package** model;

/\*\*

\* **@author** Marim Elhanafy 201803468

\*

\*/

**public** **class** Subject {

**private** **int** id;

**private** String title;

**private** **int** creditHours;

**private** **double** grade;

/\*\*

\* Constructor to initialize fields

\*/

**public** Subject(**int** id, String title, **int** creditHours, **double** grade) {

**super**();

**this**.id = id;

**this**.title = title;

**this**.creditHours = creditHours;

**this**.grade = grade;

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getTitle() {

**return** title;

}

**public** **void** setTitle(String title) {

**this**.title = title;

}

**public** **int** getCreditHours() {

**return** creditHours;

}

**public** **void** setCreditHours(**int** creditHours) {

**this**.creditHours = creditHours;

}

**public** **double** getGrade() {

**return** grade;

}

**public** **void** setGrade(**double** grade) {

**this**.grade = grade;

}

@Override

**public** String toString() {

**return** "Subject [id=" + id + ", title=" + title + ", creditHours=" + creditHours + ", grade=" + grade + "]";

}

}

**Tree Class:**

**package** model;

/\*\*

\* **@author** Hagar Elsayed 201805123

\*

\*/

**public** **class** Tree<E> {

**private** NodeTree<E> root;

/\*\*

\* **@param** k: key of a node

\* **@return** data of the given node

\*/

**public** E search(**int** k) {

**if** (root == **null**)

**return** **null**;

NodeTree<E> current = root;

**while** (current.key != k) {

**if** (k < current.key)

current = current.leftChild;

**else**

current = current.rightChild;

**if** (current == **null**)

**return** **null**;

}

**return** current.data;

}

/\*\*

\* **@param** k: key of a node

\* **@param** e: data of a node

\*/

**public** **void** insert(**int** k, E e) {

NodeTree<E> newNode = **new** NodeTree(k, e);

**if** (root == **null**)

root = newNode;

**else** {

NodeTree current = root;

NodeTree parent;

**while** (**true**) {

parent = current;

**if** (k < current.key) {

current = current.leftChild;

**if** (current == **null**) {

parent.leftChild = newNode;

**return**;

}

} **else** {

current = current.rightChild;

**if** (current == **null**) {

parent.rightChild = newNode;

**return**;

}

}

}

}

}

/\*\*

\* **@param** k: key of a node

\* **@return** true if the node deleted successfully

\* false if not found

\*/

**public** **boolean** delete(**int** k) {

NodeTree current = root;

NodeTree parent = root;

**boolean** isLeftChild = **true**;

**while** (current.key != k) {

parent = current;

**if** (k < current.key) {

isLeftChild = **true**;

current = current.leftChild;

} **else** {

isLeftChild = **false**;

current = current.rightChild;

}

**if** (current == **null**)

**return** **false**;

}

**if** (current.leftChild == **null** && current.rightChild == **null**) {

**if** (current == root)

root = **null**;

**else** **if** (isLeftChild)

parent.leftChild = **null**;

**else**

parent.rightChild = **null**;

} **else** **if** (current.rightChild == **null**)

**if** (current == root)

root = current.leftChild;

**else** **if** (isLeftChild)

parent.leftChild = current.leftChild;

**else** // right child of parent

parent.rightChild = current.leftChild;

**else** **if** (current.leftChild == **null**)

**if** (current == root)

root = current.rightChild;

**else** **if** (isLeftChild) // left child of parent

parent.leftChild = current.rightChild;

**else** // right child of parent

parent.rightChild = current.rightChild;

**else** {

NodeTree successor = getSuccessor(current);

**if** (current == root)

root = successor;

**else** **if** (isLeftChild)

parent.leftChild = successor;

**else**

parent.rightChild = successor;

successor.leftChild = current.leftChild;

}

**return** **true**;

}

/\*\*

\* **@param** delNode: the node that we want to delete

\* **@return** the successor

\*/

**private** NodeTree<E> getSuccessor(NodeTree<E> delNode) {

NodeTree<E> successorParent = delNode;

NodeTree<E> successor = delNode;

NodeTree<E> current = delNode.rightChild; // go to right child

**while** (current != **null**) // until no more

{ // left children,

successorParent = successor;

successor = current;

current = current.leftChild; // go to left child

}

// if successor not

**if** (successor != delNode.rightChild) // right child,

{ // make connections

successorParent.leftChild = successor.rightChild;

successor.rightChild = delNode.rightChild;

}

**return** successor;

}

/\*\*

\* **@param** traverse the tree in ascending order

\*/

**public** **void** traverse() {

inorder(root);

}

/\*\*

\* **@param** n: root

\* -print (left - root - right)

\*/

**public** **void** inorder(NodeTree<E> n) {

**if** (n == **null**)

**return**;

**else** {

inorder(n.leftChild);

n.display();

inorder(n.rightChild);

}

}

}

**TreeOfLists Class:**

**package** model;

/\*\*

\* **@author** Marim Elhanafy 201803468

\*

\*/

**public** **class** TreeOfLists {

**private** Tree<DoublyLinkedList<Student>> treeOfStudents = **new** Tree<>();

/\*\*

\* **@param** student

\*/

**public** **void** insert(Student student) {

**int** year = Integer.*parseInt*(String.*valueOf*(student.getId()).substring(0, 4));// 2017

DoublyLinkedList<Student> DLOfStudents01 = treeOfStudents.search(year);

**if** (DLOfStudents01 == **null**) {

DoublyLinkedList<Student> DLOfStudents02 = **new** DoublyLinkedList<Student>();

DLOfStudents02.addLast(student);

treeOfStudents.insert(year, DLOfStudents02);

} **else** {

DLOfStudents01.addLast(student);

}

}

/\*\*

\* **@param** id of student

\* **@return** Student

\*/

**public** Student find(**int** id) {

**int** year = Integer.*parseInt*(String.*valueOf*(id).substring(0, 4));

DoublyLinkedList<Student> DLOfStudents01 = treeOfStudents.search(year);

**if** (DLOfStudents01 == **null**) {

**return** **null**;

}

**else** {

Student student = **null**;

**for** (NodeDoublyLinkedList<Student> temporary = DLOfStudents01.getHeader().getNext(); temporary != DLOfStudents01.getTrailer(); temporary = temporary.getNext()) {

**if** (temporary.getData().getId() == id)

student = temporary.getData();

}

**return** student;

}

}

/\*\*

\* **@param** id of student

\* **@return** true if the student deleted successfully

\* false if not found

\*/

**public** **boolean** delete(**int** id) {

**int** year = Integer.*parseInt*(String.*valueOf*(id).substring(0, 4));

DoublyLinkedList<Student> DLOfStudents01 = treeOfStudents.search(year);

**if** (DLOfStudents01 == **null**) {

**return** **false**;

} **else** {

Student student = **null**;

**for** (NodeDoublyLinkedList<Student> temporary = DLOfStudents01.getHeader().getNext(); temporary != DLOfStudents01.getTrailer(); temporary = temporary.getNext())

**if** (temporary.getData().getId() == id)

student = temporary.getData();

**if** (student == **null**)

**return** **false**;

**else** {

**if** (DLOfStudents01.size() == 1)

treeOfStudents.delete(year);

**else** {

**for** (NodeDoublyLinkedList<Student> tmp = DLOfStudents01.getHeader().getNext(); tmp != DLOfStudents01.getTrailer(); tmp = tmp.getNext()) {

**if** (tmp.getData().getId() == id) {

NodeDoublyLinkedList<Student> p = tmp.getPrev();

NodeDoublyLinkedList<Student> n = tmp.getNext();

p.setNext(n);

n.setPrev(n);

}

}

}

**return** **true**;

}

}

}

/\*\*

\* **@param** id of student

\* **@param** subject

\* **@return** true if the student exist and the subject assigned successfully

\* false if student not found

\*/

**public** **boolean** addSubject(**int** id, Subject subject){

**int** year = Integer.*parseInt*(String.*valueOf*(id).substring(0, 4));

DoublyLinkedList<Student> DLOfStudents01 = treeOfStudents.search(year);

**if** (DLOfStudents01 == **null**) {

**return** **false**;

} **else** {

Student student = **null**;

**for** (NodeDoublyLinkedList<Student> temporary = DLOfStudents01.getHeader().getNext(); temporary != DLOfStudents01.getTrailer(); temporary = temporary.getNext()) {

**if** (temporary.getData().getId() == id)

student = temporary.getData();

}

**if**(student==**null**)

**return** **false**;

**else** {

student.getSubjects().add(subject);

**return** **true**;

}

}

}

/\*\*

\* **@param** year

\* **@return** the average grade of all students in that year

\*/

**public** **double** Avg(**int** year) {

DoublyLinkedList<Student> DLOfStudents01 = treeOfStudents.search(year);

**if** (DLOfStudents01 == **null**) {

**return** 0;

} **else** {

**int** NoOfSubForAllStudent = 0;

**double** totalGradesForAllStudents = 0;

**for** (NodeDoublyLinkedList<Student> temporary = DLOfStudents01.getHeader().getNext(); temporary != DLOfStudents01.getTrailer(); temporary = temporary.getNext()) {

**for** (Subject subject : temporary.getData().getSubjects()) {

NoOfSubForAllStudent++;

totalGradesForAllStudents += subject.getGrade();

}

}

**if**(NoOfSubForAllStudent == 0)

**return** 0;

**else**

**return** totalGradesForAllStudents/NoOfSubForAllStudent;

}

}

/\*\*

\* prints all students, in an ascending order

\*/

**public** **void** PrintStudents(){

treeOfStudents.traverse();

}

}

**App Class:**

**package** Test;

**import** model.\*;

/\*\*

\* **@author** Hagar Elsayed - Mai Saed - Marim Elhanafy

\*

\*/

**public** **class** App {

**public** **static** **void** main(String[] args) {

TreeOfLists studentsOfQU = **new** TreeOfLists();

studentsOfQU.insert(**new** Student(201903468, "Hagar"));//

studentsOfQU.insert(**new** Student(201903508, "Mai"));

studentsOfQU.insert(**new** Student(202005123, "Marim"));

studentsOfQU.insert(**new** Student(201812345, "Rawan"));

studentsOfQU.insert(**new** Student(20181234, "Salma"));

System.***out***.println("Students' List:\n");

studentsOfQU.PrintStudents();

System.***out***.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

System.***out***.println("Using method (find) to find student with id (201812345): ");

System.***out***.println(studentsOfQU.find(201812345));

System.***out***.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

studentsOfQU.delete(201903468);

System.***out***.println("Students' List\nafter deleting student with id (201903468):\n");

studentsOfQU.PrintStudents();

System.***out***.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

studentsOfQU.addSubject(201903508, **new** Subject(1, "CMPS 151", 3, 95));

studentsOfQU.addSubject(201903508, **new** Subject(2, "CMPS 310", 4, 86));

studentsOfQU.addSubject(202005123, **new** Subject(1, "ELEC 101", 3, 91));

studentsOfQU.addSubject(202005123, **new** Subject(2, "CMPS 151", 3, 80));

System.***out***.println("Average Grades of 2019's students = "+studentsOfQU.Avg(2019));

System.***out***.println("Average Grades of 2020's students = "+studentsOfQU.Avg(2020));

}

}