LaSalle College
Montréal

# Final Evaluation: 40%

## Course Identification

| | |
|---|---|
| Name of program – Code: | COMPUTER SCIENCE TECHNOLOGY – PROGRAMMING – 420.BP |
| Course title: | **DATA STRUCTURES** |
| Course number: | 420-SD4-AS |
| Group: | 07252 \| 07258 \| 07268 |
| Teacher's name: | MICHELLE M. KHALIFE |
| Duration: | TAKE HOME |
| Semester: | WINTER 2024 |

## Student Identification

Name: _____ Student number: _____

Date: _____ Result: _____

☐ I declare that this is an original work, and that I credited all content sources of which I am not the author (online and printed, images, graphics, films, etc.), in the required quotation and citation style for this work.

## Standard of the Evaluated Competency

**Competency:**
Develop native applications with a database – 00SS
**Elements of the competency**
5. Program the application logic
6. Control the quality of the application

**Competency:**
Develop gaming or simulation applications – 00SW

8. Produce the documentation

## Instructions

- Take home project. Students can work alone, in pairs. Teams of 3 need permission.
- Demo/presentations dates follow the exam schedule, [unless otherwise communicated in class.
- Plagiarism, attempts at plagiarism, or complicity in plagiarism during a summative evaluation results in a mark of zero (0). In the case of recidivism, in the same course or in another course, the student will be given a grade of '0' for the course in question. (IPEL – Article 5.16).

**If we feel that your answers may not be yours, the department reserves the right to complete your evaluation with a virtual meeting to verify that you have reached the required competency.**

## Mark Breakdown

This evaluation is on 100 points

| | |
|---|---|
| Contact Tree Methods (13 methods in total) | |
| - 8 methods (5pts ea.) | 40pts |
| - 5 methods (10pts ea.) | 50pts |
| Self-documenting & well-commented code | 10pts |

**TOTAL: 100 POINTS**

## TREES – Implementation of a Lightweight Contact Tracing Data Structure

A TreeNode structure consists of the following members:

```
String medicareId; // unique identifier cannot be duplicated
int directContacts = 0; // no. of children a node has, default 0
int totalCases = 1; // no. of nodes rooted at this tree node including self
TreeNode* parentPtr; // pointer to the parent node
List<TreeNode*> directContactsPtrList; // list of pointers to children nodes
```

And the following constructors:

```
TreeNode(const String& medId) { medicareId = medId; parent = nullptr; };
TreeNode(TreeNode* parentPtr, const String& medId) {medicareId = medId; parent = parentPtr;};
```

The `String` type is defined as such: `typedef std::string String;`

Use the TreeNode definition to implement a ContactTree class with the methods in the table below.

| Methods | Description |
|---|---|
| `ContactTree()` | Constructor |
| `~ContactTree()` | Destructor |
| `bool IsEmpty()` | Returns true if the tree is empty |
| `int GetSize()` | Returns the number of nodes in the tree including root |
| `void AddPatient0(const String&)` | Patient0 becomes the root of the tree |
| `void AddContact(const String&, const String&)` | Second parameter is the child of the first. |
| `TreeNode* LookUpContact(const String&)` | Returns a pointer to the node if found, nullptr otherwise |
| `void DeleteContact(const String&)` | Deletes the subtree rooted at that node |
| `void DisplayContact(const String&)` | Displays the node's content given an Id |
| `void DisplayContact(TreeNode*)` | Displays the node's content given a pointer to the node |
| `void TraceSource(const String&)` | Displays all the nodes from this node till the root |
| `void PrintContactCases(const String&)` | Print direct children contacts, given a contact |
| `void PrintContactTree()` | An iterative Breadth First Traversal |
| `void PrintHierarchicalTree()` | Hierarchical printout based on recursion |

**NOTES:**

1. The **const String&** parameter refers to a health card Id number.
2. The following pages detail general directions to implement the class's methods. If you need to add a method, combine two methods, and/or want to edit a method – be it in its signature or implementation – you are welcome to do so, provided you are maintaining efficiency, sound logic and simplicity.
3. **You will need to decide on the best suited linear data structure for traversing the tree for LookUp & other purposes, and subsequently, implement it. See Project description on page 7 – worth 30% of your final grade. If your template implementation does not work, you are then welcome to the STL's.**
4. The recursive PrintHierarchicalTree() method would lay out the tree nodes in a pre-order fashion – refer to the Fibonacci output in the Stack lecture (slide 11).
5. Keep in mind that reducing complexity is one of the goals. Do not forget to check whether the tree is empty or not, whether the root contains what you're looking for or not, etc…

ContactTree()

Constructor – sets the root to nullptr
Alternatively, you are welcome to consider constructing the tree by passing a patient0 instead of adding it.

| **Element of competency:** Programming the application logic (00SS.5) | |
|---|---|
| **Performance criteria** | **weight** |
| 5.1 Proper programming or integration of authentication and authorization mechanisms | /5 |

---

~ContactTree()

Destructor –call to DeleteContact() at the root

| **Element of competency:** Programming the application logic (00SS.5) | |
|---|---|
| **Performance criteria** | **weight** |
| 5.1 Proper programming or integration of authentication and authorization mechanisms | /5 |

---

bool IsEmpty()

returns true if the tree has no nodes or if the size of the tree is zero, false otherwise

| **Element of competency:** Programming the application logic (00SS.5) | |
|---|---|
| **Performance criteria** | **weight** |
| 5.1 Proper programming or integration of authentication and authorization mechanisms | /2.5 |

---

int GetSize()

returns the total number of nodes in the tree including the root

| **Element of competency:** Programming the application logic (00SS.5) | |
|---|---|
| **Performance criteria** | **weight** |
| 5.1 Proper programming or integration of authentication and authorization mechanisms | /2.5 |

---

void AddPatient0(const String&)

Create a new tree node given the parameter and let the root of the tree point to this new node. Alternatively, you could also have a parameterized constructor that accepts a contact Id and creates the tree as well as its first node.

| **Element of competency:** Control the quality of the application (00SS.6) | |
|---|---|
| **Performance criteria** | **weight** |
| 6.3 Relevance of the corrective action | /10 |

---

void AddContact(const String&, const String&) – 10pts

Given two contact Ids, the method connects node2 as a child of node1. You need to locate node1 in the tree using LookUpContact(contact Id1). If found, create a new node with contact Id2 and add its location to node1's directContactsPtrList. Increment parent's direct contacts and this parent's –as well as all other ancestors– number of total cases. If not found, you cannot proceed.

| **Element of competency:** Control the quality of the application (00SS.6) | |
|---|---|
| **Performance criteria** | **weight** |
| 6.3 Relevance of the corrective action | /10 |

```
TreeNode* LookUpContact(const String&)
```

Traverse the tree level by level searching for a node with the given contact Id. You will be iterating over pointers, which give you access to the fields of the tree nodes. If found, return the pointer to the node you're seeking. If you're done with your iteration, return false.

Start at the root. Push the root at the back of the data structure. While the data structure is not empty, check the front element, if it points to the node you seek, return it. If not, add its direct contact pointers to the end of your structure, remove this element, and continue. If the data structure is empty && you still haven't returned a pointer, then return nullptr.

| **Element of competency:** Programming the application logic (00SS.5) | |
|---|---|
| **Performance criteria** | **weight** |
| 5.1 Proper programming or integration of authentication and authorization mechanisms | /10 |

---

```
void DeleteContact(const String&)
```

Given a contact Id, traverse the tree and get a pointer to the node by calling LookUpContact(contactId). If found, you must delete the whole subtree rooted at that node – including the node itself. If the contact Id is not found, let the user know.

To delete, you must first collect all the pointers to the subtree's nodes. Proceed in a BFS manner, whereby you add the first pointer into the data structure and subsequently add its children. You keep going until you've added all the elements. Keep track of where you are.

Once all elements have been entered, move backwards by calling delete() on the last pointer and then popping the element to give you access to the next one.

When you arrive at that subtree's root, make sure you delete its pointer from the parents' list of children. Don't forget to decrement the parents' direct contact as well each of the ancestors' total cases.

| **Element of competency:** Control the quality of the application (00SS.6) | |
|---|---|
| **Performance criteria** | **weight** |
| 6.3 Relevance of the corrective action | /10 |

---

```
void DisplayContact(const String&) – 5pts
```

Look up a given contact Id. If found, call DisplayContact(TreeNode*). If not found, let the user know.

| **Element of competency:** Programming the application logic (00SS.5) | |
|---|---|
| **Performance criteria** | **weight** |
| 5.1 Proper programming or integration of authentication and authorization mechanisms | /5 |

---

```
void DisplayContact(TreeNode*) – 5pts
```

Given a pointer to the node, printout the node's data (medicareId, directContacts, totalCases).

| **Element of competency:** Programming the application logic (00SS.5) | |
|---|---|
| **Performance criteria** | **weight** |
| 5.1 Proper programming or integration of authentication and authorization mechanisms | /5 |

---

`void TraceSource(const String&) – 5pts`

Look up a given contact Id. If found, get its parent* and DisplayContact(parent*). Repeat until you reach the root. If not found, let the user know.

| **Element of competency:** Programming the application logic (00SS.5) | |
|---|---|
| **Performance criteria** | **weight** |
| 5.1 Proper programming or integration of authentication and authorization mechanisms | /5 |

---

`void PrintContactCases(const String&) – 5pts`

Look up a given contact Id. If found, iterate over its direct contacts list, calling DisplayContact(TreeNode*). If not found, let the user know.

| **Element of competency:** Programming the application logic (00SS.5) | |
|---|---|
| **Performance criteria** | **weight** |
| 5.1 Proper programming or integration of authentication and authorization mechanisms | /5 |

---

`void PrintContactTree() – 10pts`

An iterative Breadth First Traversal that works the same way as the LookUpContact method. Here, print the node's data. You may find a call to DisplayContact(TreeNode*) useful.

| **Element of competency:** Programming the application logic (00SS.5) | |
|---|---|
| **Performance criteria** | **weight** |
| 5.1 Proper programming or integration of authentication and authorization mechanisms | /5 |

---

`void PrintHierarchicalTree()`

A recursive traversal from the root that prints the tree in a hierarchical directory/file fashion.

| **Element of competency:** Programming the application logic (00SS.5) | |
|---|---|
| **Performance criteria** | **weight** |
| 5.1 Proper programming or integration of authentication and authorization mechanisms | /5 |

---

Code is otherwise self-documenting and well-commented?

| **Element of competency:** Produce the documentation (00SS.8) | |
|---|---|
| **Performance criteria** | |
| 8.1 Proper identification of the information to be written up | /5 |
| 8.2 Clear record of the work carried out | /5 |

## Project Description – /50pts. 30%

The Contact Tracing Data Structure described above requires a linear data structure into which elements are inserted and from which they are retrieved.

Linear options include:

- Linked List
    - Singly
    - Singly & Circular
    - Doubly
    - Doubly & Circular
- Stack
- Queue, and
- Deque – this ADT inherits from both the stack & the queue i.e. push(), pop(), top, and enqueue() and dequeue().

1/ Which of the above is best suited for the job? Justify your choice for the required functions & their complexity. 10pts
2/ Implement the data structure for a type of your choice. 30pts
3/ Update it to handle any given type (use templates). 10pts