CS1020E: Data Structures and Algorithms I

Tutorial 4 – ADTs, Lists

17 February 2017

# 1. NUS Modules

Have you ever thought about writing a program to calculate your NUS Cumulative Average Point (CAP)? Here, we will be implementing an **NUSModule** ADT, and use an array of **NUSModule** objects to calculate a CAP score.

The **NUSModule** ADT needs to be able to store the module code, the number of modular credits that it is worth, and the module letter grade. In addition, it should be able to get the grade point equivalent of the letter grade. These should all be stored as private attributes within the class, due to encapsulation.

All modules are initially given a letter grade from A+ to F, but it can also be declared as an S/U grade at a later time (assume all modules can be declared as S/U). S and U are special grades. If a module is given any of those grades, that module will contribute 0 modular credits to the CAP.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Letter | A+ | A | A- | B+ | B | B- | C+ | C | D+ | D | F |
| Point | 5.0 | 5.0 | 4.5 | 4.0 | 3.5 | 3.0 | 2.5 | 2.0 | 1.5 | 1.0 | 0.0 |
| S/U | S | S | S | S | S | S | S | S | U | U | U |

Source: <http://www.eng.nus.edu.sg/ugrad/SI_caps.html>

A skeleton code of the class is provided below.

class NUSModule {

private:

string \_name;

int \_credits;

string \_grade;

double \_point;

// other private attributes and methods here

public:

NUSModule( string name, int credits, string grade = “F” ) // some default grade

: \_name( name ), \_credits( credits ), \_grade( grade )

{ /\* anything else needed? \*/ }

string toString () {

ostringstream oss;

oss << \_name << “(“ << \_credits << “): ” << \_grade << “ = “ << \_point;

return oss.str();

}

void setGrade( string grade ); // sets the grade, and updates the point

// other public methods here

};

(a) List some methods that you think should be included, and specify whether they should be public or private methods.

**We will need a method to declare a module as S/U. It should be public so that the module can be set from outside of itself.**

**We could also have a method to do the internal conversion from a letter grade to a grade point value, in order to make the code more modular. It should be private as only the class will need to call this method.**

(b) The method **setGrade** has not been implemented. Implement the method.

(c) Implement a **calcCAP** function that calculates the CAP, given an array of **NUSModules.**

double calcCAP( NUSModule moduleList[ ], int numModules);

# 2. Fractions

C++ contains several numeric data types, e.g. int, double, etc. However, it does not support fractions. We will store our fraction as an irreducible fraction, which means having an integer numerator and denominator with no common factors within them. We also need to implement some arithmetic operations on fractions.

One way of implementing a fraction in C++ is to inherit from the STL **pair< int, int >,** which is defined in the header **< utility >**:

class Fraction : pair<int, int> {

private:

//some private methods here

public:

Fraction( int num, int den) {

first=num;

second=den;

//anything else needed?

}

string getFraction () {

ostringstream oss;

oss << first << "/" << second;

return oss.str();

}

Fraction add(Fraction other);

//adds the current fraction with another fraction, and then returns the result as a fraction

//other public methods here

};

(a) List some of methods that you think should be included, and specify whether they should be public or private methods.

**We should have a method to reduce a fraction into its lowest terms, as the result of some mathematical operations would give a reducible fraction. It should be private as we don’t need to call this function from outside of the class.**

**The different mathematical operations between fractions (subtract, multiply, divide) should also be implemented. They should be public as they will be called from outside of the class.**

(b) The method **add** has not been defined. Implement the method.

(c) Implement 3 other methods proposed in (a), not including getter/setter methods.

# 3. List ADT Implementations

In the lectures, we have discussed the array and singly linked list implementations for a list ADT. Let us compare their performances. Given a list containing **N** elements, how many elements will be accessed/modified for each implementation when:

(a) Adding an element to the front of the list.

**Array: N elements will be modified, as all the existing elements will need to be copied one space to the right before the new element can be inserted to the front.**

**Linked list: 1 element will be modified. The new element to be added will point to the old head, and the head will now point at the new element.**  
(b) Adding an element to the back of the list.

**Array: Usually 1 element, if there is spare space at the end of the array. Otherwise N elements will be modified as the entire array will need to be copied over to a larger sized array before we can add the new element.**

**Linked list: N elements will be accessed, as we will need to traverse the list to get to the end first, before we can add the new element.**  
(c) Removing an element from the front of the list.

**Array: N elements will be modified, as the second element onwards will need to be copied one space to the left.**

**Linked list: 1 element will be accessed. The element to be removed is simply deleted, and the head pointer redirected.**  
(d) Removing an element from the back of the list.

**Array: 0 elements will be accessed. We simply need to decrement the size count and the last element is effectively gone from the list.**

**Linked list: N elements will be accessed, as we will need to traverse the list to get to the last element before we can remove it.**  
(e) Retrieve an element by index.

**Array: 1 element will be accessed, as we can easily access any element in the array by index.**

**Linked list: The number of elements access will be equal to the index, as we will need to traverse to the correct index to retrieve it.**

# 4. Singly Linked List

Implement an integer linked list class, with the following methods:

(a) **push** – adds an integer to the front of the list  
(b) **pop** – removes an integer from the front of the list  
(c) **retrieve** – returns the integer stored at position idx  
(d) **remove** – removes the integer stored at position idx   
(e) **sort** – sorts the integers in ascending order  
(f) **constructor** – ensure that there is no improper pointer dereferencing at any point  
(g) **destructor**– ensure that there is no memory leak at the end of the life of a **LinkedList** object.

A skeleton code is provided for you below. You may implement other helper methods.

class LinkedList {

private:

struct Node {

int number;

Node\* next;

};

Node\* \_head;

public:

LinkedList () ;

~LinkedList () ;

void push( int value );

void pop () ;

int retrieve( int idx );

void remove( int idx );

void sort () ;

};