Algorithms

ICT283 – Data Structures and Abstractions – Assignment 2

32067232 – Jake Kroon

I have not written an algorithm for any set and get methods as that is a waste of time. I have instead written algorithms for anytime a calculation is performed or there is some form of processing being done. Each algorithm will be placed under a heading dictating the class it belongs to, this will be in a **bold** font. I've also avoided anything that does not involve a pure calculation.

MyVector

```
returns datatype reference MyVector<datatype>(constant int j)
        if j < 0
                THEN throw exception

        unsigned i = cast_to_unsigned(j)

        if (i + 1) >= arraySize
                THEN GrowVector(i)

        if i >= counter
                THEN counter = i + 1

        return array[i]

/*****************************************************************************/

GrowVector(constant unsigned i)
        arraySize = i * 2
        datatype tmp = new datatype[arraySize]

        for j = 0; j < counter; j++
                tmp[j] = array[j]

        delete from memory [] vec

        vec = temp
```

```
/*****************************************************************************/

Copy(constant MyVector &newVec)
        if this != &newVec
                THEN
                        counter = newVec.Counter
                        counter = newVec.arraySize

                        delete from memory [] vec
                        vec = new dataType[arraySize]

                        for unsigned i = 0; i < newVec.GetSize(); i++
                                vec[i] = newVec[i]

/*****************************************************************************/
```

BinaryTree

```
returns void Clear(node parent)
        if parent is NULL
                THEN Clear( parent.left )
                THEN Clear ( parent.right )
                THEN delete parent
                THEN parent = NULL

returns node Copy(constant node parent)
        if parent is NULL
                THEN return NULL

        node * newNode = newNode(parent.data)

        newNode.left = NULL
        newNode.right = NULL

        newNode.right = Copy(parent.right)
        newNode.left = Copy(parent.left)
```

```
 returns void Insert(datatype newData)
        root = Insert(newdata, root)

returns node pointer Insert(T newData, node * parent)
        if parent IS NULL
                THEN return new node(newData)

        if newData < parent.data
                THEN parent.left = Insert(newData, parent.left)
        else if (parent.data < newData)
                THEN parent.right = Insert(newData, parent.right)

        return Balance(parent)

returns node pointer Balance(node * parent)
        FixHeight(parent)

        if BalanceDifference(parent) is 2
                THEN if Balance difference(parent.right) is 0
                        THEN parent.right = RotateRight(parent.left)
                THEN RotateLeft(parent);

        if BalanceDifference(parent) is -2
                THEN if BalanceDifference(parent.left) greater than 0
                        THEN parent.left = RotateLeft(parent.left)
                THEN return RotateRight(parent)

        return parent
```

```
returns node pointer RotateRight(node * parent)
        node y = parent.left
        parent.left = y.right
        y.right = parent

        FixHeight(parent)
        FixHeight(y)
        return y

returns node pointer RotateLeft(node * parent)
        node x = parent.right
        parent.right = x.left
        x.left = parent

        FixHeight(parent)
        FixHeight(x)
        return x

returns unsigned integer Height(node * parent)
        If parent is NULL
                THEN return 0
        ELSE
                return height

returns integer BalanceDifference(node * parent)
        Height(parent.left) – Height(parent.right)
```

```
returns void FixHeight
        unsigned hl = Height(parent.left)
        unsigned hr = Height(parent.right)

        if(h1 greater than hr)
                THEN parent.height = hl + 1
        else
                THEN parent.height = hr + 1

InOrderHelper(node *& parent, function func)
        IF parent is not NULL
                THEN InOrderHelper(parent.left, func)
                THEN func(parent.data)
                THEN InOrderHelper(parent right, func)

PreOrderHelper(node *& parent, function func)
        IF parent is not NULL
                THEN func (parent.data)
                THEN PreOrderHelper(parent.left, func)
                THEN PreOrderHelper(parent.right, func)

PostOrderHelper(node *& parent, function func)
        IF parent is not NULL
                THEN PreOrderHelper(parent.left, func)
                THEN PreOrderHelper(parent.right, func)
                THEN func (parent data)
```

```
returns boolean Search(T key, function func)
        boolean found
        found = SearchHelper(key, root, func)

        return found

returns boolean SearchHelper(T key, node *&parent, function func)
        IF parent is not NULL
                THEN
                        If key is parent.data
                                THEN func(parent.data)
                                THEN return true
                        if key is less than parent.data
                                THEN return SearchHelper(key, parent.left, func)
                        else
                                THEN return SearchHelper(key, parent.right, func)

        return false
```

Date

```
returns string GetDate()
        string date
        if day < 10
                THEN date += 0 + “/”
        else
                THEN date += day + “/”

        if month < 10
                THEN date += 0 + month + “/”
        else
                THEN date += month + “/”

        if year < 10
                THEN date += 000 + year
        else if year < 100
                THEN date += 00 + year
        else if year < 1000
                THEN date += 0 + year
        else
                THEN date += year
        return date
```

Student

```
returns double CalculateGPA(BinaryTree & units)
        double totalCreditPoints = 0
        double preDivisionTotal = 0
        double GPA = 0

        for unsigned I = 0; I < GetSize(); i++
                UnitGPACalc(units, totalCreditPoints, PreDivisionTotal, I)

        GPA = preDivisionTotal / totalCreditPoints

        return GPA

returns double CalculateGPA(BinaryTree & units, unsigned year)
        double totalCreditPoints = 0
        double preDivisionTotal = 0
        double GPA = 0

        for unsigned I = 0; I < GetSize(); i++
                if(year == substring(3,1) of GetUnitId(i)
                        THEN UnitGPACalc(units, totalCreditPoints, PreDivisionTotal, I)

        if preDivisionTotal is 0 and totalCreditPoints is 0
                return 0
        else
                GPA = preDivisionTotal/ totalCreditPoints
        return GPA
```

```
returns void UnitGPACalc(binaryTree & units, double & tcp, double & pdt, unsigned I)
        SetUnit(GetUnitId(i), units)
        if GetResult(i) >= 80
                THEN pdt += statUnit.GetCredits() * 4
        else if GetResult(i) >= 70
                THEN pdt += statUnit.GetCredits() * 3
        else if GetResult(i) >= 60
                THEN pdt += statUnit.GetCredits() * 2
        else if GetResult(i) >= 50
                THEN pdt += statUnit.GetCredits() * 1

        tcp + = statUnit.GetCredits()

returns constant unsigned GetHighestMark
        unsigned highest = 0

        for unsigned i = 0; i < GetSize(); i++
                if highest < GetResult(i)
                        THEN highest = GetResult(i)

        return highest


returns constant unsigned GetLowestMark
        unsigned lowest = 101

        for unsigned i = 0; i < GetSize(); i++
                if lowest > GetResult(i)
                        THEN lowest = GetResult(i)

        return lowest
```

```
Copy(const Student &obj)
    if this!= &obj
        THEN
            delete from memory results
            results = new MyVector<Results>()

            SetStudentId(obj.GetStudentId())
            SetFirstName(obj.GetFirstName())
            SetLastName(obj.GetLastName())

            for unsigned i = 0; i < obj.GetSize(); i++
                SetUnitName(i, obj.GetUnitName(i))
                SetUnitId(i, obj,GetUnitId(i))
                SetUnitCredits(i, obj,GetUnitCredits(i))
                SetResult(i, obj.GetResult(i))
                SetResultSemester(i, obj.GetResultSemester(i))
                SetDay(i, obj.GetDay(i))
                SetMonth(i, obj.GetMonth(i))
                SetYear(i, obj.GetYear(i))
```

StudentIO

returns void GetHighestMarkOutput(const long sId, map<long, Student> & students, binaryTree,Unit> & units)

for unsigned I = 0; I < students[sId].GetSize(); i++

if(students[sId].GetResult(i) is students[sId].GetHighestMark())

THEN if Student.SetUnit(students[sId].GetHighestMark())

THEN WRITE TO File

WRITE “Student ID: “ + students[sId].GetStudentId(i)

WRITE “Surname : “ + students[sId].GetLastName(i)

WRITE “Unit code :“ + student[sId].UnitId(i)

WRITE “Unit name :” + student.UnitName(i)

WRITE “Unit mark :” + student[sId].GetResult(i)

WRITE “Date :” + student[sId].GetDay(i) + “/” +

THEN WRITE TO Console

WRITE “Student ID: “ + students[sId].GetStudentId(i)

WRITE “Surname : “ + students[sId].GetLastName(i)

WRITE “Unit code :“ + student[sId].UnitId(i)

WRITE “Unit name :” + student.UnitName(i)

WRITE “Unit mark :” + student[sId].GetResult(i)

WRITE “Date :” + student[sId].GetDay(i) + “/” +

else

write output failed to console and file

returns void GetLowestMarkOutput(const long sId, map<long, Student> & students, binaryTree,Unit> & units)

for unsigned I = 0; I < students[sId].GetSize(); i++

if(students[sId].GetResult(i) is students[sId].GetLowestMark())

THEN if Student.SetUnit(students[sId].GetLowestMark())

THEN WRITE TO File

WRITE “Student ID: “ + students[sId].GetStudentId(i)

WRITE “Surname : “ + students[sId].GetLastName(i)

WRITE “Unit code :“ + student[sId].UnitId(i)

WRITE “Unit name :” + student.UnitName(i)

WRITE “Unit mark :” + student[sId].GetResult(i)

WRITE “Date :” + student[sId].GetDay(i) + “/” +

THEN WRITE TO Console

WRITE “Student ID: “ + students[sId].GetStudentId(i)

WRITE “Surname : “ + students[sId].GetLastName(i)

WRITE “Unit code :“ + student[sId].UnitId(i)

WRITE “Unit name :” + student.UnitName(i)

WRITE “Unit mark :” + student[sId].GetResult(i)

WRITE “Date :” + student[sId].GetDay(i) + “/” +

else

write output failed to console and file

```
returns void GetHighestLowest(map<long, Student> & students, BinaryTree<Unit> & units)
        double highest = -1
        double lowest = 4.00
        MyVector<string> hnames
        MyVector<long> hsids
        MyVector<string> lnames
        MyVector<long> lsids
        double total = 0

        map iterator = it

        for(it = beginnning of map; it does not equal end of map; it++)
                double gpa = it.second.CalculateGPA(units)
                HighestGPA(hnames, hsids, highest, it.second, gpa)
                LowestGPA(lnames, lsids, lowest, it.second, gpa)

                total += gpa

        total  = total / students.size()

        HighestLowestOutput(console. lnames, lsids, hsids, lowest, highest, total)
        HighestLowestOutput(file,  lnames, lsids, hsids, lowest, highest, total)
```

returns void HighestGPA(MyVector<string & hnames, MyVector<long> & hsids, double & highest, const Student & check, double gpa)

```
if(gpa > highest)
        THEN
                hnames.clear()
                hsids.clear()

                highest  = gpa
                hsids[hsids.GetSize()] = check.GetStudentId()
                hsids[hsids.GetSize()] = check.GetLastName()
else if gpa is highest
        THEN
                hsids[hsids.GetSize()] = check.GetStudentId()
                hnames[hnames.GetSize()] = check.GetLastName()
```

returns void LowesttGPA(MyVector<string & lnames, MyVector<long> & lsids, double & lowest, const Student & check, double gpa)

```
if(gpa < lowest)
        THEN
                lnames.clear()
                lsids.clear()

                lowest  = gpa
                lsids[lsids.GetSize()] = check.GetStudentId()
                lsids[lsids.GetSize()] = check.GetLastName()
else if gpa is lowest
        THEN
                lsids[lsids.GetSize()] = check.GetStudentId()
                lnames[lnames.GetSize()] = check.GetLastName()
```

Main

menu

bool flag

while flag is true

print Please enter your choice of input:

switch(user input)

case 1:

sId = GetInput()

if student exists in map

Then GetHighestMarkOutput(sId, students, unitTree)

else

print student not found

case 2:

sId = GetInput()

if student exists in map

THEN GetLowestMarkOutput(sId, students, unitTree)

else

print student not found

case 3:

sId = GetInput

if student exists in map

GetGPACalcOutput(sId, students, unitTree)

else

print student not found

case 4:

GetHighestLowest(students, unitTree)

```
case 5:
        sId = GetInput()
        if student exists in map
                print please enter a year
                year = user input
                if year does not equal 0
                        then GetYearGPA(sId, year, students, unitTree)
                else
                        print that wasn't a year try again

        else
                print student not found

case 6:
        end program.
```