Hogebomen

Lisette de Schipper (s1396250) en Micky Faas (s1407937)

Abstract

Blabla

1 Inleiding

AVL-bomen, splay-bomen en treaps zijn klassieke datastructuren die ingezet worden om een verzameling gegevens te faciliteren. Het zijn zelfbalancerende binaire zoekbomen die elk een vorm van ruimte en/of tijd-efficiëntie aanbieden. Er worden experimenten verricht om de prestatie van deze zelf-balancerende zoekbomen te vergelijken, aan de hand van ophaaltijd van data, mate van herstructurering en het verwijderen van knopen. Ook wordt de prestatie van deze zoekbomen uitgezet tegen de ongebalanceerde tegenhanger, de binaire zoekboom.

2 Werkwijze

De vier bomen zijn conceptueel eenvoudig en relatief makkelijk te implementeren.

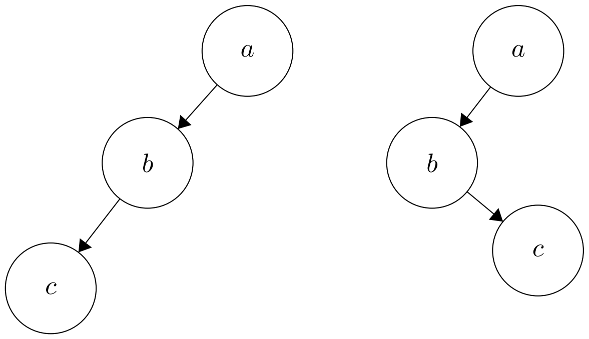
2.1 Implementatie binaire zoekboom

TO DO

2.2 Implementatie AVL-bomen

Knopen van een AVL-boom hebben een *balansfactor*, die altijd -1, 0 of 1 moet zijn. In deze implementatie is de balansfactor de hoogte van de rechtersubboom min de hoogte van de linkersubboom. Dit houdt dus in dat de hoogte van de linkersubboom van de wortel met maar 1 knoop kan verschillen van de hoogte van de rechtersubboom van de wortel. Het moment dat de balansfactor van een knoop minder dan -1 of meer dan 1 wordt, moet de boom geherstructureerd worden, om deze eigenschap te herstellen.

Om de balansfactor voor elke knoop te berekenen, houdt elke knoop zijn eigen hoogte bij. De balansfactor van een knoop wordt hersteld door rotaties. De richting en de hoeveelheid van de rotaties hangt af van de vorm van de betreffende (sub)boom. De volgende twee vormen en hun spiegelbeelden kunnen voorkomen bij het verwijderen of toevoegen van een knoop:



In het eerste geval moet de wortel naar rechts worden geroteerd. In het tweede geval moeten we eerst naar de staat van de eerste subboom komen, door b naar links te roteren. Voor de spiegelbeelden van deze twee vormen geldt hetzelfde alleen in spiegelbeeld.

In deze implementatie van een AVL-boom bedraagt het toevoegen van een knoop in het ergste geval O($logn$) tijd, waarbij n staat voor de hoogte van de boom. Eerst moet er gekeken worden of de data niet al in de boom voorkomt (O($logn$)) en vervolgens moet de boom op basis van de toevoeging geherstructureerd worden. Dit laatste is in het ergste geval O($logn$), omdat dan de gehele boom tot de wortel moeten worden nagelopen.
De complexiteitsgraad van het verwijderen van een knoop is gelijk aan die van het toevoegen van een knoop, omdat dezelfde operaties uitgevoerd moeten worden.

2.3 Implementatie Splay-bomen

TO DO

2.4 Implementatie Treaps

3 Onderzoek

Een praktisch voorbeeld van binair zoeken in een grote boom is de spellingscontrole. Een spellingscontrole moet zeer snel voor een groot aantal strings kunnen bepalen of deze wel of niet tot de taal behoren. Aangezien er honderduizenden woorden in een taal zitten, is lineair zoeken geen optie. Voor onze experimenten hebben wij dit als uitgangspunt genomen en hieronder zullen we kort de experimenten toelichten die wij hebben uitgevoerd. In het volgende hoofdstuk staan vervolgens de resultaten beschreven.

3.1 Hooiberg

“Hooiberg” is de naam van het testprogramma dat we hebben geschreven speciaal ten behoeven van onze experimenten. Het is een klein console programma dat woorden uit een bestand omzet tot een boom in het geheugen. Deze boom kan vervolgens worden doorzocht met de input uit een ander bestand: de “naalden”. De syntax is alsvolgt:

```
hooiberg type hooiberg.txt naalden.txt [treap-random-range]
```

Hierbij is `type` één van `bst, avl, splay, treap`, het eerste bestand bevat de invoer voor de boom, het tweede bestand een verzameling strings als zoekopdracht en de vierde parameters is voorbehouden voor het type `treap`. De bestanden kunnen woorden of zinnen bevatten, gescheiden door regeleinden. De binaire bomen gebruiken lexicografische sortering die wordt geleverd door de operatoren `<` en `>` van de klasse `std::string`. Tijdens het zoeken wordt een exacte match gebruikt (case-sensitive, non-locale-aware).

3.2 Onderzoeks(deel)vragen

Met onze experimenten hebben we gepoogd een aantal eenvoudige vragen te beantwoorden over het gebruik van de verschillende binaire en zelf-organiserende bomen, te weten:

- Hoeveel meer rekenkracht kost het om grote datasets in te voegen in zelf-organiserende bomen tov binaire bomen?
- Levert een zelf-organiserende boom betere zoekprestaties en onder welke opstandigheden?
- Hoeveel extra geheugen kost een SOT?
- Wat is de invloed van de random-factor bij de Treap?

3.3 Meetmethoden

Om de bovenstaande vragen te toetsen, hebben we een aantal meetmethoden bedacht.

- Rekenkracht hebben we gemeten in milliseconden tussen aanvang en termineren van een berekening. We hebben de delta's berekend rond de relevante code blokken dmv de C++11 `chrono` klassen in de Standard Template Library. Alle test zijn volledig sequentieel en single-threaded uitgevoerd. Deze resultaten zijn representatie voor één bepaald systeem, vandaar dat we aantal % 'meer rekenkracht' als eenheid gebruiken.
- Zoekprestatie hebben we zowel met rekenkracht als zoekdiepte gemeten. De zoekdiepte is het aantal stappen dat vanaf de wortel moet worden gemaakt om bij de gewenste knoop te komen. We hebben hierbij naar het totaal aantal stappen gekeken en naar de gemiddelde zoekdiepte.
- Geheugen hebben we gemeten met de `valgrind` memory profiler. Dit programma wordt gebruikt voor het opsporen van geheugen lekken en houdt het aantal allocaties op de heap bij. Dit is representatie voor het aantal gealloceerde nodes. Aangezien hooiberg nauwelijks een eigen geheugenvoetafdruk heeft, zijn deze waarden representatief.

3.4 Input data

Voor ons experiment hebben we een taalbestand gebruikt van OpenTaal.org met meer dan 164.000 woorden. Dit is een relatief klein taalbestand, maar voldoede om verschillen te kunnen zien. We hebben een aantal testcondities gebruikt:

- Voor het inladen een wel of niet alfabetisch gesoorteerd taalbestand gebruiken.
- Als zoekdocument hebben we een gedicht met 62 woorden gebruikt. Er zitten een aantal dubbele woorden in alsook een aantal woorden die niet in de woordenlijst voorkomen (werkwoordsvervoegingen).
- We hebben ook een conditie waarbij we alle woorden gezocht hebben, zowel in dezelfde, als in een andere volgorde dan dat ze zijn ingevoerd.
- We hebben één conditie waarbij we de random-range van de Treap hebben gevariëerd.

3.5 Hypothesen

- De binairy search tree zal vermoedelijk het snelst nieuwe data toevoegen. De splay tree heeft veel ingewikkelde rotatie bij een insert, dus deze zal het traagst zijn.
- Bij het gedicht zal de splay boom waarschijnlijk het snelst zijn omdat deze optimaliseert voor herhalingen.
- ...
- De bomen die een aparte node-klasse gebruiken (avl en treap) gebruiken het meeste geheugen.
- Items over Treap

4 Resultaten

5 Conclusies

6 Appendix

6.1 ExpressionAtom.h

```
/**
 * ExpressionAtom:
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    ExpressionAtom.h
 * @date    26-10-2014
 **/

```

```
#ifndef EXPRESSIONATOM_H
#define EXPRESSIONATOM_H

#include <ostream>
#include <string>
#include <cmath>

typedef struct {
    int numerator;
    int denominator;
} Fraction;

/**
* @function  operator==( )
* @abstract  Test equality for two Fractions
* @param     lhs and rhs are two sides of the comparison
* @return    true upon equality
* @post      Two Fraction are equal if
*            lhs.numerator/lhs.denominator == rhs.numerator/rhs.denominator
**/
bool operator==( const Fraction& lhs, const Fraction& rhs );

/**
* @function  Arithmetic operators +, -, *, /
* @abstract  Arithmetic result of two Fractions
* @param     lhs and rhs are two sides of the expression
**/
Fraction operator+( const Fraction& lhs, const Fraction& rhs );
Fraction operator-( const Fraction& lhs, const Fraction& rhs );
Fraction operator*( const Fraction& lhs, const Fraction& rhs );
Fraction operator/( const Fraction& lhs, const Fraction& rhs );

using namespace std;

class ExpressionAtom {
    public:
        enum AtomType {
            UNDEFINED =0x0,
            INTEGER_OPERAND,
            FLOAT_OPERAND,
            FRACTION_OPERAND,
            NAMED_OPERAND, // Variable
            OPERATOR,
            FUNCTION
        };

        enum OperatorType {
            SUM,
            DIFFERENCE,
            PRODUCT,
            DIVISION,
            EXPONENT
        };

        enum Function {
            SIN,
            COS,
            TAN,
            LOG,
            LN,
            SQRT,
            ABS,
            E,
            PI,
            UNARY_MINUS
        };

       /**
        * @function  ExpressionAtom( )
        * @abstract  Constructor, defines an ExpressionAtom for various types
        * @param     Either one of AtomType, OperatorType, Function,
        *            float, long int, Fraction or string
        * @post      ExpressionAtom is always valid, containing the
        *            supplied value. No argument yields UNDEFINED.
        **/
        ExpressionAtom( AtomType t =UNDEFINED, long int atom =0l );
        ExpressionAtom( float atom );
        ExpressionAtom( long int atom );
        ExpressionAtom( string var );
        ExpressionAtom( OperatorType op );
        ExpressionAtom( Function func );
        ExpressionAtom( Fraction frac );

       /**
        * @function  operator==( )
        * @abstract  Test equality for two ExpressionAtom
        * @param     ExpressionAtom or either one of AtomType, OperatorType,
        *            Function, float, long int, Fraction  or string
        * @return    true upon equality
        * @post      Two ExpressionAtoms are equal if
        *              - their types are equal
        *              - their value is equal
        *              - they are not UNDEFINED
        **/
        bool operator ==( const ExpressionAtom& rhs ) const;

       /**
        * @function  Inquality operators <, >, <= and >=
        * @abstract  Test equality for two ExpressionAtoms
        * @param     ExpressionAtom or either one of AtomType, OperatorType,
        *            Function, float, long int, Fraction  or string
        * @return    true upon resp. lt, gt, lte or gte
        * @pre       Both operands should be of the numeric operand type
        *            Types do not have to be equal
        * @post      always false if !isNumericOperand( ) or UNDEFINED
        **/
        bool operator <( const ExpressionAtom& rhs ) const;
        bool operator >( const ExpressionAtom& rhs ) const;
```

```
     bool operator <=( const ExpressionAtom& rhs ) const;
     bool operator >=( const ExpressionAtom& rhs ) const;

    /**
     * @function  Arithmetic operators +, -, *, /
     * @abstract  Arithmetic result of two ExpressionAtoms
     * @param     ExpressionAtom or either one of AtomType, OperatorType,
     *            Function, float, long int, Fraction  or string
     * @return    ExpressionAtom (xvalue) containing the result
     *            The type of this ExpressionAtom doesn't need to be
     *            equal to one of the operand's types
     * @pre       Both operands should be of the numeric operand type
     *            Types do not have to be equal
     * @post      undefined if !isNumericOperand( ) or UNDEFINED
     **/
     ExpressionAtom operator+( const ExpressionAtom& rhs ) const;
     ExpressionAtom operator-( const ExpressionAtom& rhs ) const;
     ExpressionAtom operator*( const ExpressionAtom& rhs ) const;
     ExpressionAtom operator/( const ExpressionAtom& rhs ) const;

    /**
     * @function  pow( )
     * @abstract  Raise to power
     * @param     ExpressionAtom or Either one of AtomType, OperatorType,
     *            Function, float, long int, Fraction  or string
     * @return    ExpressionAtom (xvalue) containing the result
     *            The type of this ExpressionAtom doesn't need to be
     *            equal to one of the operand's types
     * @pre       Both operands should be of the numeric operand type
     *            Types do not have to be equal
     * @post      undefined if !isNumericOperand( ) or UNDEFINED
     **/
     ExpressionAtom pow( const ExpressionAtom& power ) const;
    /**
     * @function  sqrt( )
     * @abstract  Square root
     * @pre       Instance should be of the numeric operand type
     *            Types do not have to be equal
     * @return    ExpressionAtom (xvalue) containing the result
     *            The type of this ExpressionAtom doesn't need to be
     *            equal to the operand's types
     * @post      undefined if !isNumericOperand( ) or UNDEFINED
     **/
     ExpressionAtom sqrt( ) const;

    /**
     * @function  setters
     * @abstract  sets ExpressionAtom to a given value
     * @param     Either one of AtomType, OperatorType,
     *            Function, float, long int, Fraction  or string
     * @post      The type is changed to match the new value
     **/
     void setFloat( float d )
         { m_type =FLOAT_OPERAND; m_atom.float_atom =std::move( d ); }
```

```
        void setInteger( long int i )
            { m_type =INTEGER_OPERAND; m_atom.integer_atom =std::move( i ); }
        void setFraction( const Fraction& frac )
            { m_type =FRACTION_OPERAND; m_atom.fraction_atom =std::move( frac ); }
        void setFunction( Function f )
            { m_type =FUNCTION; m_atom.integer_atom =std::move( f ); }
        void setOperator( OperatorType op )
            { m_type =OPERATOR; m_atom.integer_atom =std::move( op ); }
        void setNamed( string str )
            { m_type =NAMED_OPERAND; m_named_atom =std::move( str ); }

       /**
        * @function  getters
        * @abstract  Return the value as a certain type
        * @return    Returns the value as the requested type
        * @pre       Type should match the requested datatype
        * @post      undefined if type doesn't match or UNDEFINED
        **/
        float getFloat( ) const { return m_atom.float_atom; }
        long int getInteger( ) const { return m_atom.integer_atom; }
        Fraction getFraction( ) const { return m_atom.fraction_atom; }
        int getFunction( ) const { return (int)m_atom.integer_atom; }
        int getOperator( ) const { return (int)m_atom.integer_atom; }
        string getNamed( ) const { return m_named_atom; }

       /**
        * @function  isNumericOperand( )
        * @abstract  Returns whether this instance holds a numeric type
        * @return    bool with the result
        **/
        bool isNumericOperand( ) const {
            return m_type == FLOAT_OPERAND
                || m_type == INTEGER_OPERAND
                || m_type == FRACTION_OPERAND; }

       /**
        * @function  numeric casting functions
        * @abstract  Casts the value to a certain type
        * @return    Returns the value as the requested type
        * @pre       Type should be a numeric operand
        *            toFloat( ) and toInteger( ) are defined for
        *            FLOAT_OPERAND, INTEGER_OPERAND and FRACTION_OPERAND
        *            toFraction( ) is defined for INTEGER_OPERAND and FRACTION
        * @post      undefined if !isNumericOperand( )
        **/
        float toFloat( ) const;
        long int toInteger( ) const;
        Fraction toFraction( ) const;

       /**
        * @function  type( )
        * @abstract  Gives the specified type
        * @return    One of AtomType
        **/
        AtomType type( ) const { return m_type; }

       /**
        * @function  arity( )
        * @abstract  Returns the arity of the specified type
        * @return    Arity ranging from 0 to 2
        **/
        short arity( ) const;

    private:
        union {
            long int integer_atom;
            float float_atom;
            Fraction fraction_atom;
        } m_atom;
        string m_named_atom;
        AtomType m_type;
};

/**
* @function  operator <<( ostream& out, const ExpressionAtom& atom )
* @abstract  Overloads operator<< to support ExpressionAtom
* @return    an ostream with the contents of atom inserted
**/
ostream& operator <<( ostream& out, const ExpressionAtom& atom );

#endif
```

6.2 ExpressionAtom.cc

```
/**
 * ExpressionAtom:
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    ExpressionAtom.cc
 * @date    26-10-2014
 **/

#include "ExpressionAtom.h"
#include "ExpressionTree.h"

/* Fraction overloads */

bool operator ==( const Fraction& lhs, const Fraction& rhs ) {
    // This function should be in general namespace
    return ExpressionTree::compare( (float)lhs.numerator/(float)lhs.denominator,
                                    (float)rhs.numerator/(float)rhs.denominator );
}

Fraction operator+( const Fraction& lhs, const Fraction& rhs ) {
    Fraction f;
    if( lhs.denominator == rhs.denominator ) {
        f.denominator =lhs.denominator;
        f.numerator =lhs.numerator + rhs.numerator;
    } else {
        f.denominator =lhs.denominator * rhs.denominator;
        f.numerator =lhs.numerator * rhs.denominator
                    + rhs.numerator * lhs.denominator;
    }
    return f;
}

Fraction operator-( const Fraction& lhs, const Fraction& rhs ) {
    Fraction f;
    if( lhs.denominator == rhs.denominator ) {
        f.denominator =lhs.denominator;
        f.numerator =lhs.numerator - rhs.numerator;
    } else {
        f.denominator =lhs.denominator * rhs.denominator;
        f.numerator =lhs.numerator * rhs.denominator
                    - rhs.numerator * lhs.denominator;
    }
    return f;
}

Fraction operator*( const Fraction& lhs, const Fraction& rhs ) {
    Fraction f;
    f.denominator =lhs.denominator * rhs.denominator;
    f.numerator =lhs.numerator * rhs.numerator;
    return f;
}

Fraction operator/( const Fraction& lhs, const Fraction& rhs ) {
    Fraction f;
    f.denominator =lhs.denominator * rhs.numerator;
    f.numerator =lhs.numerator * rhs.denominator;
    return f;
}

/* ExpressionAtom implementation */

ExpressionAtom::ExpressionAtom( AtomType t, long int atom ) : m_type( t ) {
    m_atom.integer_atom =std::move( atom );
}

ExpressionAtom::ExpressionAtom( float atom ) : m_type( FLOAT_OPERAND ) {
    m_atom.float_atom =std::move( atom );
}

ExpressionAtom::ExpressionAtom( long int atom ) : m_type( INTEGER_OPERAND ) {
    m_atom.integer_atom =std::move( atom );
}

ExpressionAtom::ExpressionAtom( string var ) : m_type( NAMED_OPERAND ) {
    m_named_atom =std::move( var );
}

```

```
ExpressionAtom::ExpressionAtom( OperatorType op ) : m_type( OPERATOR ) {
    m_atom.integer_atom =std::move( op );
}

ExpressionAtom::ExpressionAtom( Function func ) : m_type( FUNCTION ) {
    m_atom.integer_atom =std::move( func );
}

ExpressionAtom::ExpressionAtom( Fraction frac ) : m_type( FRACTION_OPERAND ) {
    m_atom.fraction_atom =std::move( frac );
}

bool ExpressionAtom::operator ==( const ExpressionAtom& rhs ) const {
    if( rhs.m_type != m_type )
        return false;
    switch( m_type ) {
        case UNDEFINED:
            return false;
        case INTEGER_OPERAND:
        case OPERATOR:
        case FUNCTION:
            return m_atom.integer_atom == rhs.m_atom.integer_atom;

        case FLOAT_OPERAND:
            return m_atom.float_atom == rhs.m_atom.float_atom;

        case FRACTION_OPERAND:
            return m_atom.fraction_atom == rhs.m_atom.fraction_atom;

        case NAMED_OPERAND:
            return m_named_atom == rhs.m_named_atom;

    }
    return false;
}

bool ExpressionAtom::operator <( const ExpressionAtom& rhs ) const {
    if( !rhs.isNumericOperand( ) || !isNumericOperand( ) )
        return false;
    return toFloat( ) < rhs.toFloat( );
}

bool ExpressionAtom::operator >( const ExpressionAtom& rhs ) const {
    if( !rhs.isNumericOperand( ) || !isNumericOperand( ) )
        return false;
    return toFloat( ) > rhs.toFloat( );
}

bool ExpressionAtom::operator <=( const ExpressionAtom& rhs ) const {
    if( !rhs.isNumericOperand( ) || !isNumericOperand( ) )
        return false;
    return toFloat( ) <= rhs.toFloat( );
}

```

```
bool ExpressionAtom::operator >=( const ExpressionAtom& rhs ) const {
    if( !rhs.isNumericOperand( ) || !isNumericOperand( ) )
        return false;
    return toFloat( ) >= rhs.toFloat( );
}

ExpressionAtom ExpressionAtom::operator+( const ExpressionAtom& rhs ) const {
    ExpressionAtom a;
    if( isNumericOperand( ) && rhs.isNumericOperand( ) ) {
        if( m_type == FLOAT_OPERAND || rhs.m_type == FLOAT_OPERAND )
            a.setFloat( toFloat( ) + rhs.toFloat( ) );
        else if( m_type == FRACTION_OPERAND || rhs.m_type == FRACTION_OPERAND )
            a.setFraction( toFraction( ) + rhs.toFraction( ) );
        else
            a.setInteger( getInteger( ) + rhs.getInteger( ) );
    }
    return a;
}

ExpressionAtom ExpressionAtom::operator-( const ExpressionAtom& rhs ) const {
    ExpressionAtom a;
    if( isNumericOperand( ) && rhs.isNumericOperand( ) ) {
        if( m_type == FLOAT_OPERAND || rhs.m_type == FLOAT_OPERAND )
            a.setFloat( toFloat( ) - rhs.toFloat( ) );
        else if( m_type == FRACTION_OPERAND || rhs.m_type == FRACTION_OPERAND )
            a.setFraction( toFraction( ) - rhs.toFraction( ) );
        else
            a.setInteger( getInteger( ) - rhs.getInteger( ) );
    }
    return a;
}

ExpressionAtom ExpressionAtom::operator*( const ExpressionAtom& rhs ) const {
    ExpressionAtom a;
    if( isNumericOperand( ) && rhs.isNumericOperand( ) ) {
        if( m_type == FLOAT_OPERAND || rhs.m_type == FLOAT_OPERAND )
            a.setFloat( toFloat( ) * rhs.toFloat( ) );
        else if( m_type == FRACTION_OPERAND || rhs.m_type == FRACTION_OPERAND )
            a.setFraction( toFraction( ) * rhs.toFraction( ) );
        else
            a.setInteger( getInteger( ) * rhs.getInteger( ) );
    }
    return a;
}

ExpressionAtom ExpressionAtom::operator/( const ExpressionAtom& rhs ) const {
    ExpressionAtom a;
    if( isNumericOperand( ) && rhs.isNumericOperand( ) ) {
        if( m_type == FLOAT_OPERAND || rhs.m_type == FLOAT_OPERAND )
            a.setFloat( toFloat( ) / rhs.toFloat( ) );
        else if( m_type == FRACTION_OPERAND || rhs.m_type == FRACTION_OPERAND )
            a.setFraction( toFraction( ) / rhs.toFraction( ) );
        else
            a.setInteger( getInteger( ) / rhs.getInteger( ) );
    }
    return a;
}

ExpressionAtom ExpressionAtom::pow( const ExpressionAtom& power ) const {
    ExpressionAtom a;
    if( isNumericOperand( ) && power.isNumericOperand( ) ) {

        if( power.m_type == FRACTION_OPERAND
            && power.m_atom.fraction_atom == Fraction( { 1, 2 } ) ) {
            return sqrt( );
        }
        else if( m_type == FLOAT_OPERAND
            || power.m_type == FLOAT_OPERAND
            || power.m_type == FRACTION_OPERAND )
            a.setFloat( ::powf( toFloat( ), power.toFloat( ) ) );
        else if( m_type == FRACTION_OPERAND ) {
            Fraction f;
            f.numerator =m_atom.fraction_atom.numerator;
            f.denominator =::pow( m_atom.fraction_atom.denominator,
                                power.getInteger( ) );
            a.setFraction( f );
        }
        else {
            if( power.getInteger( ) > 0 )
                a.setInteger( ::powl( getInteger( ), power.getInteger( ) ) );
            else if( power.getInteger( ) == 0 )
                a.setInteger( 1 );
            else {
                Fraction f;
                f.numerator =1;
                f.denominator =::pow( m_atom.integer_atom,
                                    abs( power.m_atom.integer_atom ) );
                a.setFraction( f );
            }
        }
    }
    return a;
}

ExpressionAtom ExpressionAtom::sqrt( ) const {
    ExpressionAtom a;
    if( isNumericOperand( ) ) {
        if( m_type == FLOAT_OPERAND  ) {
            a.setFloat( ::sqrtf( toFloat( ) ) );
        }
        else if( m_type == FRACTION_OPERAND ) {
            float f =::sqrtf( (float)m_atom.fraction_atom.denominator );
            if( ceil( f ) == floor( f ) )
                a.setFraction(
                    Fraction( { m_atom.fraction_atom.numerator, (int)f } ) );
            else
                a.setFloat( f );
        }
        else {
            float f =::sqrtf( (float)m_atom.integer_atom );
            if( ceil( f ) == floor( f ) )
                a.setInteger( (int)f );
            else
                a.setFloat( f );
        }
    }
    return a;
}

float ExpressionAtom::toFloat( ) const {
    if( m_type == INTEGER_OPERAND )
        return (float)m_atom.integer_atom;
    else if( m_type == FLOAT_OPERAND )
        return m_atom.float_atom;
    else if( m_type == FRACTION_OPERAND )
        return (float)m_atom.fraction_atom.numerator /
               (float)m_atom.fraction_atom.denominator;

    return float( );
}
long int ExpressionAtom::toInteger( ) const {
    if( m_type == INTEGER_OPERAND )
        return m_atom.integer_atom;
    else if( m_type == FLOAT_OPERAND )
        return (long int)m_atom.float_atom;
    else if( m_type == FRACTION_OPERAND )
        return m_atom.fraction_atom.numerator /
               m_atom.fraction_atom.denominator;

    return int( );
}
Fraction ExpressionAtom::toFraction( ) const {
    Fraction frac;
    if( m_type == FRACTION_OPERAND )
        return m_atom.fraction_atom;
    else if( m_type == INTEGER_OPERAND ) {
        frac.numerator =m_atom.integer_atom;
        frac.denominator =1;
    }
    return frac;
}

short ExpressionAtom::arity( ) const {
    switch( type( ) ) {
        case ExpressionAtom::INTEGER_OPERAND:
        case ExpressionAtom::FLOAT_OPERAND:
        case ExpressionAtom::FRACTION_OPERAND:
        case ExpressionAtom::NAMED_OPERAND:
            return 0;

        case ExpressionAtom::OPERATOR:
            return 2;
        case ExpressionAtom::FUNCTION:
            switch( getFunction( ) ) {
                case ExpressionAtom::SIN:
                case ExpressionAtom::COS:
                case ExpressionAtom::TAN:
                case ExpressionAtom::LN:
                case ExpressionAtom::SQRT:
                case ExpressionAtom::ABS:
                case ExpressionAtom::UNARY_MINUS:
                    return 1;
                case ExpressionAtom::E:
                case ExpressionAtom::PI:
                    return 0;
                case ExpressionAtom::LOG:
                    return 2;
            }
            break;
        case ExpressionAtom::UNDEFINED:
        default:
            return 0;
    }
    return 0;
}

/* General namespace */

ostream& operator <<( ostream& out, const ExpressionAtom& atom ) {
    switch( atom.type( ) ) {
        case ExpressionAtom::INTEGER_OPERAND:
            out << atom.getInteger( );
            break;
        case ExpressionAtom::FLOAT_OPERAND:
            out << atom.getFloat( );
            break;
        case ExpressionAtom::FRACTION_OPERAND:
            out << atom.getFraction( ).numerator << ”/”
                 << atom.getFraction( ).denominator;
            break;
        case ExpressionAtom::NAMED_OPERAND:
            out << atom.getNamed( );
            break;
        case ExpressionAtom::OPERATOR:
            switch( atom.getOperator( ) ) {
                case ExpressionAtom::SUM:
                    out << ”+”;
                    break;
                case ExpressionAtom::DIFFERENCE:
                    out << ”−”;
                    break;
                case ExpressionAtom::PRODUCT:
                    out << ”∗”;
                    break;
                case ExpressionAtom::DIVISION:
                    out << ”/”;
                    break;
                case ExpressionAtom::EXPONENT:
                    out << "^";
                    break;
            }
            break;
        case ExpressionAtom::FUNCTION:
            switch( atom.getFunction( ) ) {
                case ExpressionAtom::SIN:
                    out << "sin";
                    break;
                case ExpressionAtom::COS:
                    out << "cos";
                    break;
                case ExpressionAtom::TAN:
                    out << "tan";
                    break;
                case ExpressionAtom::LOG:
                    out << "log";
                    break;
                case ExpressionAtom::LN:
                    out << "ln";
                    break;
                case ExpressionAtom::SQRT:
                    out << "sqrt";
                    break;
                case ExpressionAtom::ABS:
                    out << "abs";
                    break;
                case ExpressionAtom::E:
                    out << "e";
                    break;
                case ExpressionAtom::PI:
                    out << "pi";
                    break;
                case ExpressionAtom::UNARY_MINUS:
                    out << "-";
                    break;
            }
            break;
        case ExpressionAtom::UNDEFINED:
        default:
            break;
    }
    return out;
}
```

6.3 ExpressionTree.h

```
/**
 * ExpressionTree:
 *
 * @author Micky Faas (s1407937)
 * @author Lisette de Schipper (s1396250)
 * @file ExpressionTree.h
 * @date 10-10-2014
 **/

#ifndef EXPRESSIONTREE_H
#define EXPRESSIONTREE_H

#include "Tree.h"
#include "ExpressionAtom.h"
#include <fstream>
#include <string>
#include <exception>
#include <stdexcept>
#include <sstream>
#include <cmath>
#include <map>

using namespace std;

class ParserException : public exception
{
    public:
       ParserException( const string &str ) : s( str ) {}
       ~ParserException() throw () {}
       const char* what() const throw() { return s.c_str(); }

    private:
       string s;
};

class ExpressionTree : public Tree<ExpressionAtom>
{
    public:
       /**
        * @function  ExpressionTree( )
        * @abstract  Constructor, creates an object of the tree.
        * @post      The tree has been declared.
        **/
        ExpressionTree( ) : Tree<ExpressionAtom>() { }

       /**
        * @function  ExpressionTree( )
        * @abstract  fromString is called to make a tree from the string.
        * @param     str, a string that will be parsed to create the three.
        * @post      The tree has been declared and initialized.
        **/
        ExpressionTree( const string& str ) : Tree<ExpressionAtom>() {
            fromString( str );
        }

       /**
        * @function  tokenize( )
        * @abstract  Breaks the string provided by fromString up into tokens
        * @param     str, a string expression
```

```
         * @return    tokenlist, a list of ExpressionAtom's
         * @pre       str needs to be a correct space-separated string
         * @post      We have tokens of the string
         **/
         static list<ExpressionAtom> tokenize( const string& str );

        /**
         * @function  fromString( )
         * @abstract  calls tokenize to generate tokens from an expression and
         *            fills the ExpressionTree with them.
         * @param     expression, a string expression
         * @post      The provided expression will be converted to an
         *            ExpressionTree if it has the right syntax.
         **/
         void fromString( const string& expression );

        /**
         * @function  differentiate( )
         * @abstract  calls the other differentiate function and returns the
         *            derivative in the form of a tree
         * @param     string varName, the variable
         * @return    the derivative of the original function in the form of a
         *            tree
         * @pre       There needs to be a tree
         * @post      Derivatree has been changed by the private differentiate
         *            function.
         **/
         ExpressionTree differentiate( string varName );

        /**
         * @function  simplify( )
         * @abstract  Performs mathematical simplification on the expression
         * @post      Upon simplification, nodes may be deleted.
         *            references and iterators may become invalid
         **/
         void simplify( );

        /**
         * @function  evaluate( )
         * @abstract  Evaluates the tree as far as possible given a variable and
         *            its mapping
         * @return    A new ExpressionTree containing the evaluation (may be a
         *            single node)
         * @param     varName, variable name to match (e.g, 'x')
         * @param     expr, expression to put in place of varName
         **/
         ExpressionTree evaluate( string varName, ExpressionAtom expr ) const;

        /**
         * @function  evaluate( )
         * @abstract  Evaluates the tree as far as possible using a given mapping
         * @return    A new ExpressionTree containing the evaluation (may be a
         *            single node)
         * @param     varmap, list of varName/expr pairs
         **/
         ExpressionTree evaluate( const map<string,ExpressionAtom>& varmap ) const;

        /**
         * @function  mapVariable( )
         * @abstract  Replaces a variable by an expression
         * @param     varName, variable name to match (e.g, 'x')
         * @param     expr, expression to put in place of varName
         * @post      Expression may change, references and iterators
         *            remain valid after this function.
         **/
         void mapVariable( string varName, ExpressionAtom expr );

        /**
         * @function  mapVariables( )
         * @abstract  Same as mapVariable( ) for a set of variables/expressions
         * @param     varmap, list of varName/expr pairs
         * @post      Expression may change, references and iterators
         *            remain valid after this function.
         **/
         void mapVariables( const map<string,ExpressionAtom>& varmap );

        /**
         * @function  generateInOrder( )
         * @abstract  generates the infix notation of the tree.
         * @param     out, the way in which we want to see the output
         * @post      The infix notation of the tree has been generated
         **/
         void generateInOrder( ostream& out ) const {
             generateInOrderRecursive( m_root, out );
         }

     private:
        /**
         * @function  differentiate( ), differentiateExponent( ),
         *            differentiateDivision( ), differentiateProduct( ),
         *            differentiateFunction( ), differentiateAddition( )
         * @abstract  differentiates ExpressionTree and places the derivative in
         *            the tree assigned to the last variable
         * @param     n, the node we need to start differentiating from
         * @param     varName, variable name to match (e.g, 'x')
         * @param     derivative, the node we want to differentiate from
         * @param     derivatree, the tree we want to differentiate to
         * @return    the derivative of the original function in the form of a
         *            tree
         * @pre       There needs to be a tree
         * @post      The derivatree has been changed, now it shows the
         *            derivative of ExpressionTree.
         **/
         void differentiate( node_t * n, string varName,
                             node_t * derivative,
                             ExpressionTree &derivatree );
         void differentiateExponent( node_t * n, string varName,
                                     node_t * derivative,
                                     ExpressionTree &derivatree );
         void differentiateDivision( node_t * n, string varName,
                                     node_t * derivative,
                                     ExpressionTree &derivatree );
         void differentiateProduct( node_t * n, string varName,
                                    node_t * derivative,
                                    ExpressionTree &derivatree );
         void differentiateFunction( node_t * n, string varName,
                                     node_t * derivative,
                                     ExpressionTree &derivatree );
         void differentiateAddition( node_t * n, string varName,
                                     node_t * derivative,
                                     ExpressionTree &derivatree );

        /**
         * @function  simplify( )
         * @abstract  Performs mathematical simplification on the expression
         * @param     root, root of the subtree to simplify
         * @return    New node in place of the passed value/node for root
         * @post      Upon simplification, nodes may be deleted.
         *            references and iterators may become invalid
         **/
         node_t *simplifyRecursive( node_t* root );

        /**
         * @function  generateInOrderRecursive( )
         * @abstract  Recursively goes through the tree to get the infix notation
         *            of the tree
         * @param     root, the node we're looking at
         * @param     buffer, the output
         * @post      Eventually the infix notation of the tree with parenthesis
         *            has been generated.
         **/
         void generateInOrderRecursive( node_t *root, ostream& buffer ) const;

     public:
        /**
         * @function  compare( )
         * @abstract  Throws a parser expression.
         * @param     f1, the first value we want to compare
         * @param     f2, the second value we want to compare
         * @param     error, the marge in which the difference is accepted.
         * @return    if the difference between f1 and f2 is smaller or equal to
         *            error
         * @post      A ParserException is thrown.
         **/
         static bool compare( const float &f1, const float &f2, float &&error =0.00001
             return ( fabs( f1-f2 ) <= error );
         }

 };
 #endif
```

6.4 ExpressionTree.cc

```
/**
 * ExpressionTree:
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    ExpressionTree.cc
 * @date    26-10-2014
 **/

#include ”ExpressionTree.h”

list<ExpressionAtom> ExpressionTree::tokenize( const string& str ) {

    list<ExpressionAtom> tokenlist;
    stringstream ss( str );
    while( ss.good( ) ) {
        string token;
        ss >> token;
        ExpressionAtom atom;
        bool unary_minus =false;

        if( token.size( ) > 1 && token[0] == '−' ) {
            token =token.substr( 1 );
            unary_minus =true;
        }

        if( token.find( ”.” ) != string::npos ) { // Float
            try {
                atom.setFloat( (unary_minus ? −1.0f : 1.0f)
                                * std::stof( token ) );
                unary_minus =false;
            } catch( std::invalid_argument& e ) {
                throw ParserException( string (”Invalid float ‘”)
                                        + token
                                        + string(”‘”) );
            }
        }
        else if( token == ”*” )
            atom.setOperator( ExpressionAtom::PRODUCT );
        else if( token == ”/” )
            atom.setOperator( ExpressionAtom::DIVISION );
        else if( token == ”+” )
            atom.setOperator( ExpressionAtom::SUM );
        else if( token == ”−” )
            atom.setOperator( ExpressionAtom::DIFFERENCE );
        else if( token == ”ˆ” )
            atom.setOperator( ExpressionAtom::EXPONENT );
        else if( token == ”sin” )
            atom.setFunction( ExpressionAtom::SIN );
        else if( token == ”cos” )
            atom.setFunction( ExpressionAtom::COS );
        else if( token == ”tan” )
            atom.setFunction( ExpressionAtom::TAN );
        else if( token == ”ln” )
```

```
            atom.setFunction( ExpressionAtom::LN );
        else if( token == ”log” )
            atom.setFunction( ExpressionAtom::LOG );
        else if( token == ”sqrt” )
            atom.setFunction( ExpressionAtom::SQRT );
        else if( token == ”abs” )
            atom.setFunction( ExpressionAtom::ABS );
        else if( token == ”e” )
            atom.setFunction( ExpressionAtom::E );
        else if( token == ”pi” )
            atom.setFunction( ExpressionAtom::PI );
        else if( token.find( ”/” ) != string::npos ) { // Fraction
            size_t pos =token.find( ”/” );
            Fraction f;
            try {
                f.numerator =(unary_minus ? -1 : 1)
                             * std::stoi( token.substr( 0, pos ) );
                f.denominator =std::stoi( token.substr( pos + 1 ) );
                atom.setFraction( f );
                unary_minus =false;
            }
            catch( std::invalid_argument& e ){
                throw ParserException( string (”Invalid fraction ‘”)
                                       + token
                                       + string(”‘”) );
            }
        }
        else {
            try { // Try integer
                atom.setInteger( (unary_minus ? -1 : 1) * std::stol( token ) );
                unary_minus =false;

            } // Try variable
            catch( invalid_argument& e ){
                for( unsigned int i =0; i < token.size( ); ++i )
                    if( !isalpha( token[i] ) )
                        throw ParserException( string (”Invalid token ‘”)
                                               + token
                                               + string(”‘”) );
                atom.setNamed( token );
            }
        }

        if( unary_minus )
            tokenlist.push_back( ExpressionAtom::UNARY_MINUS );
        tokenlist.push_back( atom );
    }
    return tokenlist;
}

void ExpressionTree::fromString( const string& expression ) {
    list<ExpressionAtom> tokenlist;

    try{
        tokenlist =ExpressionTree::tokenize( expression );
    } catch( ParserException & e ) {
        throw e;
    }

    Tree<ExpressionAtom>::node_t *n =0;

    for( auto atom : tokenlist ) {
        if( !n ) {
            n =pushBack( atom );
            continue;
        }
        while ( !n->info( ).arity( )
        || ( n->info( ).arity( ) == 1 && n->hasChildren( ) )
        || ( n->info( ).arity( ) == 2 && n->isFull( ) ) ) {
            n =n->parent ( );
            if( !n )
                throw ParserException( "Argument count to arity mismatch" );
        }

        n =insert( atom, n );
    }
}

ExpressionTree ExpressionTree::differentiate( string varName ) {
    ExpressionTree derivatree;
    differentiate( root( ), varName, derivatree.root( ), derivatree );
    derivatree.simplify( );
    return derivatree;
}

void ExpressionTree::simplify( ) {
    m_root =simplifyRecursive( root( ) );
}

ExpressionTree
ExpressionTree::evaluate( string varName, ExpressionAtom expr ) const {
    ExpressionTree t( *this );
    t.mapVariable( varName, expr );
    t.simplify( );
    return std::move( t );
}

ExpressionTree
ExpressionTree::evaluate( const map<string,ExpressionAtom>& varmap ) const {
    ExpressionTree t( *this );
    t.mapVariables( varmap );
    t.simplify( );
    return std::move( t );
}

void ExpressionTree::mapVariable( string varName, ExpressionAtom expr ) {
    map<string,ExpressionAtom> varmap;
    varmap[varName] =expr;
    mapVariables( varmap );
}

void ExpressionTree::mapVariables( const map<string,ExpressionAtom>& varmap ) {
    for( auto &node : *this ) {
        if( node.info( ).type( ) == ExpressionAtom::NAMED_OPERAND ) {
            auto it =varmap.find( node.info( ).getNamed( ) );
            if( it != varmap.cend( ) )
                node =it->second;
        }
    }
}

void ExpressionTree::differentiate( node_t * n, string varName,
                                    node_t * derivative,
                                    ExpressionTree &derivatree ) {
    ExpressionAtom atom =(*n);
    switch( atom.type( ) ) {
        case ExpressionAtom::OPERATOR:
        switch( atom.getOperator( ) ) {
            case ExpressionAtom::SUM:
            case ExpressionAtom::DIFFERENCE:
                differentiateAddition( &(*n), varName, derivative, derivatree );
                break;
            case ExpressionAtom::PRODUCT:
                differentiateProduct( &(*n), varName, derivative, derivatree );
                break;
            case ExpressionAtom::EXPONENT:
                differentiateExponent( &(*n), varName, derivative, derivatree );
                break;
            case ExpressionAtom::DIVISION:
                differentiateDivision( &(*n), varName, derivative, derivatree );
                break;
        }
        break;
        case ExpressionAtom::FUNCTION:
            differentiateFunction( &(*n), varName, derivative, derivatree );
            break;
        case ExpressionAtom::NAMED_OPERAND:
            atom.getNamed( ) == string( varName ) ?
            derivatree.insert( 1L, derivative ) :
            derivatree.insert( 0L, derivative );
            break;
        default:
            derivatree.insert( 0L, derivative );
    }
}

void ExpressionTree::differentiateFunction( node_t * n, string varName,
                                            node_t * derivative,
                                            ExpressionTree &derivatree ) {
    Tree<ExpressionAtom> tempTree;
    Tree<ExpressionAtom >::node_t *temp;
    ExpressionAtom atom =(*n);
    switch( atom.getFunction( ) ){
        case ExpressionAtom::SIN:
            temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative );
            differentiate( (*n).leftChild( ), varName, temp, derivatree );
            temp =derivatree.insert( ExpressionAtom::COS, temp );
            copyFromNode( (*n).leftChild( ), temp, true );
            break;
        case ExpressionAtom::TAN:;
            temp =tempTree.insert( ExpressionAtom::DIVISION, tempTree.root( ) );
            temp =tempTree.insert( ExpressionAtom::SIN, temp );
            copyFromNode( (*n).leftChild( ), temp, true );
            temp =temp->parent( );
            temp =tempTree.insert( ExpressionAtom::COS, temp );
            copyFromNode( (*n).leftChild( ), temp, true );
            differentiate( tempTree.root( ), varName, derivative, derivatree );
            tempTree.clear( );
            break;
        case ExpressionAtom::COS:
            temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative );
            temp =derivatree.insert( ExpressionAtom::UNARY_MINUS, temp );
            differentiate( (*n).leftChild( ), varName, temp, derivatree );
            temp =temp->parent( );
            temp =derivatree.insert( ExpressionAtom::SIN, temp );
            copyFromNode( (*n).leftChild( ), temp, true );
            break;
        case ExpressionAtom::LN:
            if( contains( (*n).leftChild( ), string( varName ) ) ) {
                temp =derivatree.insert( ExpressionAtom::DIVISION, derivative);
                differentiate( (*n).leftChild( ), varName, temp, derivatree );
                copyFromNode( (*n).leftChild( ), temp, false );
            }
            else
                derivatree.insert( 0L, derivative);
            break;
        case ExpressionAtom::SQRT:
            temp =derivatree.insert( ExpressionAtom::DIVISION, derivative );
            differentiate( (*n).leftChild( ), varName, temp, derivatree );
            temp =derivatree.insert( ExpressionAtom::PRODUCT, temp );
            derivatree.insert( 2L, temp );
            copyFromNode( &(*n), temp, false );
            break;
        case ExpressionAtom::LOG:
            temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative );
            temp =derivatree.insert( ExpressionAtom::DIVISION, temp );
            derivatree.insert( 1L, temp);
            temp =derivatree.insert( ExpressionAtom::LN, temp );
            copyFromNode( (*n).leftChild( ), temp, true );
            temp =temp->parent( )->parent( );
            temp =derivatree.insert( ExpressionAtom::DIVISION, temp );
            differentiate( (*n).rightChild( ), varName, temp, derivatree );
            copyFromNode( (*n).rightChild( ), temp, false );
            break;
        case ExpressionAtom::ABS:
            if( (*n).leftChild( )->info( ).type( ) ==
                ExpressionAtom::NAMED_OPERAND &&
                (*n).leftChild( )->info( ).getNamed( ) == string( varName ) ) {
                temp =derivatree.insert( ExpressionAtom::DIVISION, derivative );
                copyFromNode( (*n).leftChild( ), temp, true );
                copyFromNode( &(*n), temp, false );
            }
            else {
                temp =derivatree.insert( ExpressionAtom::DIVISION, derivative );
                temp =derivatree.insert( ExpressionAtom::PRODUCT, temp );
                copyFromNode( (*n).leftChild( ), temp, true );
                differentiate( (*n).leftChild( ), varName, temp, derivatree );
                temp =temp->parent( );
                copyFromNode( &(*n), temp, false );
            }
            break;
    }
}

void ExpressionTree::differentiateAddition( node_t * n, string varName,
                                            node_t * derivative,
                                            ExpressionTree &derivatree ) {
    Tree<ExpressionAtom>::node_t *temp;
    ExpressionAtom atom =(*n);
    if( atom.getOperator( ) == ExpressionAtom::SUM )
        temp =derivatree.insert( ExpressionAtom::SUM, derivative );
    else
        temp =derivatree.insert( ExpressionAtom::DIFFERENCE, derivative );
    differentiate( (*n).leftChild( ), varName, temp, derivatree );
    if( (*n).rightChild( ) )
        differentiate( (*n).rightChild( ), varName, temp, derivatree );
}

void ExpressionTree::differentiateDivision( node_t * n, string varName,
                                            node_t * derivative,
                                            ExpressionTree &derivatree ) {
    Tree<ExpressionAtom>::node_t *temp;
    temp =derivatree.insert( ExpressionAtom::DIVISION, derivative );
    temp =derivatree.insert( ExpressionAtom::DIFFERENCE, temp );
    temp =derivatree.insert( ExpressionAtom::PRODUCT, temp);
    copyFromNode( (*n).rightChild( ), temp, true );
    differentiate( (*n).leftChild( ), varName, temp, derivatree );
    temp =temp->parent( );
    temp =derivatree.insert( ExpressionAtom::PRODUCT, temp);
    copyFromNode( (*n).leftChild( ), temp, true );
    differentiate( (*n).rightChild( ), varName, temp, derivatree );
    temp =temp->parent( )->parent( );
    temp =derivatree.insert( ExpressionAtom::EXPONENT, temp );
    copyFromNode( (*n).rightChild( ), temp, true );
    derivatree.insert( 2L, temp );
}

void ExpressionTree::differentiateProduct( node_t * n, string varName,
                                           node_t * derivative,
                                           ExpressionTree &derivatree ) {
```

```
    Tree<ExpressionAtom>::node_t *temp;
    if( (*n).leftChild( )->info( ).isNumericOperand( ) ) {
        // n * x
        if( (*n).rightChild( )->info( ).type( ) ==
            ExpressionAtom::NAMED_OPERAND &&
            (*n).rightChild( )->info( ).getNamed( ) == string( varName ) )
            derivatree.insert( (*n).leftChild( )->info( ), derivative );
        // n * f(x)
        else {
            temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative );
             derivatree.insert( (*n).leftChild( )->info( ), temp );
             differentiate( (*n).rightChild( ), varName, temp, derivatree );
        }
    }
    else if( (*n).rightChild( )->info( ).isNumericOperand( ) ) {
        // x * n
        if( (*n).leftChild( )->info( ).type( ) ==
            ExpressionAtom::NAMED_OPERAND &&
            (*n).leftChild( )->info( ).getNamed( ) == string( varName ) )
            derivatree.insert( (*n).rightChild( )->info( ), derivative );
        // f(x) * n
        else {
            temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative );
             derivatree.insert( (*n).rightChild( )->info( ), temp );
             differentiate( (*n).leftChild( ), varName, temp, derivatree );
        }
    }
    // f(x) * g(x)
    else {
        temp =derivatree.insert( ExpressionAtom::SUM, derivative );
        temp =derivatree.insert( ExpressionAtom::PRODUCT, temp );
        copyFromNode( (*n).rightChild( ), temp, true );
        differentiate( (*n).leftChild( ), varName, temp, derivatree );
        temp =temp->parent( );
        temp =derivatree.insert( ExpressionAtom::PRODUCT, temp );
        copyFromNode( (*n).leftChild( ), temp, true );
        differentiate( (*n).rightChild( ), varName, temp, derivatree );
    }
}

void ExpressionTree::differentiateExponent( node_t * n, string varName,
                                            node_t * derivative,
                                            ExpressionTree &derivatree ) {
    Tree<ExpressionAtom>::node_t *temp;
    Tree<ExpressionAtom> tempTree;
    if( contains( (*n).leftChild( ), string( varName ) ) ) {
        // f(x) ^ g(x)
        if( contains( (*n).rightChild( ), string( varName ) ) ) {
            // f(x)^g(x) =e^(ln(f(x))g(x))
            temp =tempTree.insert( ExpressionAtom::EXPONENT, tempTree.root( ) );
            tempTree.insert( ExpressionAtom::E, temp );
            temp =tempTree.insert( ExpressionAtom::PRODUCT, temp );
            temp =tempTree.insert( ExpressionAtom::LN, temp );
            copyFromNode( (*n).leftChild( ), temp, true );
            temp =temp->parent( );
            copyFromNode( (*n).rightChild( ), temp, false );
            differentiate( tempTree.root( ), varName, derivative, derivatree );
            tempTree.clear( );
        }
        // f(x) ^ n
        else {
            if( (*n).leftChild( )->info( ).type( ) ==
                ExpressionAtom::NAMED_OPERAND &&
                (*n).leftChild( )->info( ).getNamed( ) ==
                string( varName )  ) {
                // x ^ 0
                if( (*n).rightChild( )->info( )  == 0L )
                    derivatree.insert( 1L, derivative );
                // x ^ 1
                else if( (*n).rightChild( )->info( ) == 1L )
                    derivatree.insert( string( "x" ), derivative );
                // x ^ n ( n > 1 )
                else if( (*n).rightChild( )->info( ) > 1L ) {
                    temp =derivatree.insert( ExpressionAtom::PRODUCT,
                                             derivative );
                    derivatree.insert( (*n).rightChild( )->info( ), temp );
                    temp =derivatree.insert( ExpressionAtom::EXPONENT , temp );
                    derivatree.insert( string( varName ) , temp );
                    derivatree.insert( (*n).rightChild( )->info( ) - 1L, temp );
                }
                // x ^ n ( n < 0 )
                else if( (*n).rightChild( )->info( ) < 0L ) {
                    temp =derivatree.insert( ExpressionAtom::DIVISION,
                                             derivative);
                    derivatree.insert( (*n).rightChild( )->info( ), temp );
                    temp =derivatree.insert( ExpressionAtom::EXPONENT, temp);
                    derivatree.insert( string( varName ), temp);
                    derivatree.insert( (*n).rightChild( )->info( ) -
                                       (*n).rightChild( )->info( ) -
                                       (*n).rightChild( )->info( ) + 1L, temp);
                }
            }
            else {
                temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative );
                temp =derivatree.insert( ExpressionAtom::PRODUCT, temp );
                copyFromNode( (*n).rightChild( ), temp, true );
                temp =derivatree.insert( ExpressionAtom::EXPONENT, temp );
                copyFromNode( (*n).leftChild( ), temp, true );
                derivatree.insert( (*n).rightChild( )->info( ) -
                                   (*n).rightChild( )->info( ) - 1L, temp );
                temp =temp->parent( )->parent( );
                differentiate( (*n).leftChild( ), varName, temp, derivatree );
            }
        }
    }
    //e ^ f(x)
    else if( (*n).leftChild( )->info( ).type( ) == ExpressionAtom::FUNCTION &&
             (*n).leftChild( )->info( ).getFunction( ) == ExpressionAtom::E ) {
```

```
        temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative) ;
        differentiate( (*n).rightChild( ), varName, temp, derivatree );
        copyFromNode( &(*n), temp, false );
    }
    // n ^ f(x)
    else if( contains( (*n).rightChild( ), string( varName ) ) ) {
        temp =derivatree.insert( ExpressionAtom::PRODUCT, derivative );
        temp =derivatree.insert( ExpressionAtom::PRODUCT, temp );
        differentiate( (*n).rightChild( ), varName, temp, derivatree );
        temp =derivatree.insert( ExpressionAtom::LN, temp );
        copyFromNode( (*n).leftChild( ), temp, true );
        temp =temp->parent( )->parent( );
        temp =derivatree.insert( ExpressionAtom::EXPONENT, temp );
        copyFromNode( (*n).leftChild( ), temp, true );
        copyFromNode( (*n).rightChild( ), temp, false );
    }
}


ExpressionTree::node_t *
ExpressionTree::simplifyRecursive( node_t* root ) {
    if( !root )
        return 0;

    node_t *n =root->leftChild( );
    node_t *m =root->rightChild( );

    /* cascade( ): removes root and child n, replaces root with child m */
    auto cascade =[&]( ) -> node_t* {
        remove( n );
        if( root->parent( ) ) {
            if( root ==root->parent( )->leftChild( ) )
                root->parent( )->setLeftChild( m );
            else
                root->parent( )->setRightChild( m );
            m->setParent( root->parent( ) );
        }
        else
            m->setParent( 0 );
        delete root;
        return m;
    };

    /* merge( ):
       replaces the root by the result of its operation on the children */
    auto merge =[&]( ) -> node_t* {

        ExpressionAtom &lhs =root->leftChild( )->info( );
        ExpressionAtom &rhs =root->rightChild( )->info( );
        ExpressionAtom &op =root->info( );

        assert( lhs.isNumericOperand( ) && rhs.isNumericOperand( ) );

        switch( op.getOperator( ) ) {
            case ExpressionAtom::SUM:
                op =std::move( lhs + rhs );
                break;
            case ExpressionAtom::DIFFERENCE:
                op =std::move( lhs - rhs );
                break;
            case ExpressionAtom::PRODUCT:
                op =std::move( lhs * rhs );
                break;
            case ExpressionAtom::DIVISION:
                op =std::move( lhs / rhs );
                break;
            case ExpressionAtom::EXPONENT:
                op =std::move( lhs.pow( rhs ) );
                break;
        }

        remove( m );
        remove( n );
        return root;
    };

    /* mergeInto( ): replaces the root by expr and removes the children */
    auto mergeInto =[&]( ExpressionAtom&& expr ) -> node_t* {
        remove( m );
        remove( n );
        root->info( ) =std::move( expr );
        return root;
    };

    bool stop =false;
    do {

        if( n ) {
            n =simplifyRecursive( n );
            if( n && !n->hasChildren( ) ) {
                // Simplify the one-fraction
                if( n->info( ).type( ) == ExpressionAtom::FRACTION_OPERAND
                        && n->info( ).getFraction( ).numerator == 1 )
                    n->info( ).setInteger( 1 );

                // two operands-case
                if( n->info( ).isNumericOperand( )
                    && m && m->info( ).isNumericOperand( ) ) {
                    root =merge( );
                    return root;
                }

                // 1 case
                if( n->info( ).isNumericOperand( )
                    && compare( 1.0f, n->info( ).toFloat( ) ) ) {
                    if( root->info( ) == ExpressionAtom::PRODUCT ) {
                        root =cascade( );
                    }
                    else if( root->info( ) == ExpressionAtom::EXPONENT ) {
                        if( n == root->leftChild( ) )
                            root =mergeInto( 1l );
                        else
                            root =cascade( );
                    }
                    else if( root->info( ) == ExpressionAtom::DIVISION ) {
                        if( n == root->rightChild( ) )
                            root =cascade( );
                    }
                }
                // 0 case
                else if( n->info( ).isNumericOperand( )
                         && compare( 0.0f, n->info( ).toFloat( ) ) ) {
                    if( root->info( ) == ExpressionAtom::SUM )
                        root =cascade( );
                    else if( root->info( ) == ExpressionAtom::PRODUCT ) {
                        root =mergeInto( 0l );
                    }
                    else if( root->info( ) == ExpressionAtom::DIVISION ) {
                        if( n == root->leftChild( ) )
                            root =mergeInto( 0l );
                    }
                    else if( root->info( ) == ExpressionAtom::DIFFERENCE ) {
                        if( n == root->rightChild( ) )
                            root =cascade( );
                        else if( m && m->info( ).isNumericOperand( ) ) {
                            root =mergeInto( ExpressionAtom( -1l )
                                             * m->info( ) );
                        }
                    }
                    else if( root->info( ) == ExpressionAtom::EXPONENT ) {
                        if( n == root->leftChild( ) ) {
                            if( m && m->info( ).isNumericOperand( )
                                  && compare( 1.0f, m->info( ).toFloat( ) ) )
                                root =mergeInto( 1l );
                            else {
                                root =mergeInto( 0l );
                            }
                        }
                        else {
                            root =mergeInto( 1l );
                        }
                    }
                }
                // trivial functions
                else if( root->info( ).type( ) == ExpressionAtom::FUNCTION ) {
                    switch( root->info( ).getFunction( ) ) {
                        case ExpressionAtom::UNARY_MINUS:
                            if( n->info( ).isNumericOperand( ) )
                                root =mergeInto( ExpressionAtom( -1l )
                                                 * n->info( ) );
                            break;
                        case ExpressionAtom::LN: // ln(e)
```

```
                            if( n->info( ) == ExpressionAtom::E )
                                root =mergeInto( l1 );
                            break;
                    }
                }
            }
        }

        if( stop )
            break;

        n =root->rightChild( );
        m =root->leftChild( );
        stop =true;
    } while( n );

return root;
}

void
ExpressionTree::generateInOrderRecursive( node_t *root, ostream& buffer ) const{
    if( !root )
        return;

    if( root->info( ).type( ) == ExpressionAtom::FUNCTION ) {
// Function type
        bool enclose =root->isFull( ) // Only enclose in ( )'s if neccessary
            || ( root->leftChild( ) && !root->leftChild( )->hasChildren( ) )
            || ( root->rightChild( ) && !root->rightChild( )->hasChildren( ) );

        if( root->info( ).getFunction( ) == ExpressionAtom::UNARY_MINUS ) {
            buffer << '(';
            enclose =false;
        }

        buffer << root->info( );

        if( enclose )
            buffer << '(';

        generateInOrderRecursive( root->leftChild( ), buffer );

        if( root->isFull( ) ) // Function with two params, otherwise no comma
            buffer << ',';

        generateInOrderRecursive( root->rightChild( ), buffer );

        if( enclose )
            buffer << ')';

        if( root->info( ).getFunction( ) == ExpressionAtom::UNARY_MINUS )
            buffer << ')';
    } else {    // Operator+operands type
        if( root->hasChildren( ) && root != m_root )
```

```
            buffer << '(';

        generateInOrderRecursive( root->leftChild( ), buffer );

        if( !(root->info( ) == ExpressionAtom::PRODUCT  // implicit multipl.
            && root->leftChild( )
            && root->leftChild( )->info( ).isNumericOperand( ) ) )
            buffer << root->info( );
        generateInOrderRecursive( root->rightChild( ), buffer );

        if( root->hasChildren( ) && root != m_root )
            buffer << ')';
    }
}
```

6.5 main.cc

```
/**
 * main.cc:
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    main.cc
 * @date    26-10-2014
 **/

#include <iostream>
#include "BinarySearchTree.h"
#include "Tree.h"
#include "AVLTree.h"
#include "SplayTree.h"
#include "Treap.h"
#include <string>

using namespace std;

// Makkelijk voor debuggen, moet nog beter
template<class T> void printTree( Tree<T> tree, int rows ) {
    typename Tree<T>::nodelist list =tree.row( 0 );
    int row =0;
    while( !list.empty( ) && row < rows ) {
        string offset;
        for( int i =0; i < ( 1 << (rows - row) ) - 1 ; ++i )
            offset += ' ';


        for( auto it =list.begin( ); it != list.end( ); ++it ) {
            if( *it )
                cout << offset << (*it)->info() << " " << offset;
            else
                cout << offset << ". " << offset;
        }
        cout << endl;
        row++;
        list =tree.row( row );
    }
}

int main ( int argc , char **argv ) {

    /* BST hieronder */

    cout << "BST:" << endl;
    BinarySearchTree<int> bst;

   /* auto root =bst.pushBack( 10 );
    bst.pushBack( 5 );
    bst.pushBack( 15 );

    bst.pushBack( 25 );
    bst.pushBack( 1 );
    bst.pushBack( -1 );
    bst.pushBack( 11 );
    bst.pushBack( 12 );*/

    Tree<int>* bstP =&bst; // Dit werkt gewoon :-)

    auto root =bstP->pushBack( 10 );
    bstP->pushBack( 5 );
    bstP->pushBack( 15 );

    bstP->pushBack( 25 );
    bstP->pushBack( 1 );
    bstP->pushBack( -1 );
    bstP->pushBack( 11 );
    bstP->pushBack( 12 );

    //printTree<int>( bst, 5 );


    //bst.remove( bst.find( 0, 15 ) );
    //bst.replace( -2, bst.find( 0, 5 ) );


    printTree<int>( bst, 5 );

    bst.remove( root );


    printTree<int>( bst, 5 );

    /* Splay Trees hieronder */

    cout << "Splay Boom:" << endl;
    SplayTree<int> splay;

    splay.pushBack( 10 );
    auto a =splay.pushBack( 5 );
    splay.pushBack( 15 );

    splay.pushBack( 25 );
    auto b =splay.pushBack( 1 );
    splay.pushBack( -1 );
    auto c =splay.pushBack( 11 );
    splay.pushBack( 12 );

    //printTree<int>( splay, 5 );

    //a->swapWith( b );
    //splay.remove( splay.find( 0, 15 ) );
    //splay.replace( -2, splay.find( 0, 5 ) );


    printTree<int>( splay, 5 );

    //splay.remove( root );

    splay.splay( c );

    printTree<int>( splay, 5 );

    // Test AVLTree //

    AVLTree<char> test;
    test.insert( 'a');
    auto d =test.insert( 'b');
    test.insert( 'c');
    test.insert( 'd');
    test.insert( 'e');
    test.insert( 'f');
    test.insert( 'g');
    cout << "AVL Boompje:" << endl;
    printTree<char>( test, 5 );
    cout << d->info( ) << " verwijderen: " << endl;
    test.remove( d );
    printTree<char>( test, 5 );

    // Test Treap //

    cout << "Treap" << endl;

    Treap<int> testTreap(5);
    testTreap.insert(2);
    testTreap.insert(3);
    auto e =testTreap.insert(4);
    testTreap.insert(5);
    printTree<int>( testTreap, 5 );
    testTreap.remove(e);
    printTree<int>( testTreap, 5 );

    return 0;
}
```

6.6 Tree.h

```
/**
 * Tree:
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    tree.h
 * @date    26-10-2014
 **/

#ifndef TREE_H
#define TREE_H
#include "TreeNodeIterator.h"
#include <assert.h>
#include <list>
#include <map>

using namespace std;

template <class INFO_T> class SplayTree;

template <class INFO_T> class Tree
{
    public:
        enum ReplaceBehavoir {
            DELETE_EXISTING,
            ABORT_ON_EXISTING,
            MOVE_EXISTING
        };

        typedef TreeNode<INFO_T> node_t;
        typedef TreeNodeIterator<INFO_T> iterator;
        typedef TreeNodeIterator_in<INFO_T> iterator_in;
        typedef TreeNodeIterator_pre<INFO_T> iterator_pre;
        typedef TreeNodeIterator_post<INFO_T> iterator_post;
        typedef list<node_t*> nodelist;

       /**
        * @function  Tree( )
        * @abstract  Constructor of an empty tree
        **/
        Tree( )
            : m_root( 0 ) {
        }

       /**
        * @function  Tree( )
        * @abstract  Copy-constructor of a tree. The new tree contains the nodes
        *            from the tree given in the parameter (deep copy)
        * @param     tree, a tree
        **/
        Tree( const Tree<INFO_T>& tree )
            : m_root( 0 ) {
```

```
     *this =tree;
 }

 /**
 * @function  ~Tree( )
 * @abstract  Destructor of a tree. Timber.
 **/
 ~Tree( ) {
   clear( );
 }

/**
 * @function  begin_pre( )
 * @abstract  begin point for pre-order iteration
 * @return    interator_pre containing the beginning of the tree in
 *            pre-order
 **/
 iterator_pre begin_pre( ) {
     // Pre-order traversal starts at the root
     return iterator_pre( m_root );
   }

/**
 * @function  begin( )
 * @abstract  begin point for a pre-order iteration
 * @return    containing the beginning of the pre-Order iteration
 **/
 iterator_pre begin( ) {
     return begin_pre( );
 }

/**
 * @function  end( )
 * @abstract  end point for a pre-order iteration
 * @return    the end of the pre-order iteration
 **/
 iterator_pre end( ) {
     return iterator_pre( (node_t*)0 );
 }

/**
 * @function  end_pre( )
 * @abstract  end point for pre-order iteration
 * @return    interator_pre containing the end of the tree in pre-order
 **/
 iterator_pre end_pre( ) {
     return iterator_pre( (node_t*)0 );
 }

/**
 * @function  begin_in( )
 * @abstract  begin point for in-order iteration
 * @return    interator_in containing the beginning of the tree in
 *            in-order
```

```
     **/
     iterator_in begin_in( ) {
         if( !m_root )
             return end_in( );
         node_t *n =m_root;
         while( n->leftChild( ) )
             n =n->leftChild( );
         return iterator_in( n );
       }

    /**
     * @function  end_in( )
     * @abstract  end point for in-order iteration
     * @return    interator_in containing the end of the tree in in-order
     **/
     iterator_in end_in( ) {
         return iterator_in( (node_t*)0 );
     }

    /**
     * @function  begin_post( )
     * @abstract  begin point for post-order iteration
     * @return    interator_post containing the beginning of the tree in
     *            post-order
     **/
     iterator_post begin_post( ) {
         if( !m_root )
             return end_post( );
         node_t *n =m_root;
         while( n->leftChild( ) )
             n =n->leftChild( );
         return iterator_post( n );
     }

    /**
     * @function  end_post( )
     * @abstract  end point for post-order iteration
     * @return    interator_post containing the end of the tree in post-order
     **/
     iterator_post end_post( ) {
         return iterator_post( (node_t*)0 );
     }

    /**
     * @function  pushBack( )
     * @abstract  a new TreeNode containing ‘info‘ is added to the end
     *            the node is added to the node that :
     *               - is in the row as close to the root as possible
     *               - has no children or only a left-child
     *               - seen from the right hand side of the row
     *            this is the ‘natural‘ left-to-right filling order
     *            compatible with array-based heaps and full b-trees
     * @param     info, the contents of the new node
     * @post      A node has been added.
     **/
     virtual node_t *pushBack( const INFO_T& info ) {
         node_t *n =new node_t( info, 0 );
         if( !m_root ) { // Empty tree, simplest case
             m_root =n;
         }
         else { // Leaf node, there are two different scenarios
             int max =getRowCountRecursive( m_root, 0 );
             node_t *parent;
             for( int i =1; i <= max; ++i ) {

                 parent =getFirstEmptySlot( i );
                 if( parent ) {
                     if( !parent->leftChild( ) )
                         parent->setLeftChild( n );
                     else if( !parent->rightChild( ) )
                         parent->setRightChild( n );
                     n->setParent( parent );
                     break;
                 }
             }
         }
         return n;
     }

    /**
     * @function  insert( )
     * @abstract  inserts node or subtree under a parent or creates an empty
     *            root node
     * @param     info, contents of the new node
     * @param     parent, parent node of the new node. When zero, the root is
     *            assumed
     * @param     alignRight, insert() checks on which side of the parent
     *            node the new node can be inserted. By default, it checks
     *            the left side first.
     *            To change this behavior, set preferRight =true.
     * @param     replaceBehavior, action if parent already has two children.
     *            One of:
     *            ABORT_ON_EXISTING - abort and return zero
     *            MOVE_EXISTING - make the parent's child a child of the new
     *                            node, satisfies preferRight
     *            DELETE_EXISTING - remove one of the children of parent
     *                              completely also satisfies preferRight
     * @return    pointer to the inserted TreeNode, if insertion was
     *            successfull
     * @pre       If the tree is empty, a root node will be created with info
     *            as it contents
     * @pre       The instance pointed to by parent should be part of the
     *            called instance of Tree
     * @post      Return zero if no node was created. Ownership is assumed on
     *            the new node.
     *            When DELETE_EXISTING is specified, the entire subtree on
     *            preferred side may be deleted first.
     **/
```

```
     virtual node_t* insert( const INFO_T& info,
                     node_t* parent =0,
                     bool preferRight =false,
                     int replaceBehavior =ABORT_ON_EXISTING ) {
         if( !parent )
             parent =m_root;

         if( !parent )
             return pushBack( info );

         node_t *node =0;

         if( !parent->leftChild( )
               && ( !preferRight || ( preferRight &&
                    parent->rightChild( ) ) ) ) ) {
             node =new node_t( info, parent );
             parent->setLeftChild( node );
             node->setParent( parent );

         } else if( !parent->rightChild( ) ) {
             node =new node_t( info, parent );
             parent->setRightChild( node );
             node->setParent( parent );

         } else if( replaceBehavior == MOVE_EXISTING ) {
             node =new node_t( info, parent );
             if( preferRight ) {
                 node->setRightChild( parent->rightChild( ) );
                 node->rightChild( )->setParent( node );
                 parent->setRightChild( node );
             } else {
                 node->setLeftChild( parent->leftChild( ) );
                 node->leftChild( )->setParent( node );
                 parent->setLeftChild( node );
             }

         } else if( replaceBehavior == DELETE_EXISTING ) {
             node =new node_t( info, parent );
             if( preferRight ) {
                 deleteRecursive( parent->rightChild( ) );
                 parent->setRightChild( node );
             } else {
                 deleteRecursive( parent->leftChild( ) );
                 parent->setLeftChild( node );
             }

         }
         return node;
     }

    /**
     * @function  replace( )
     * @abstract  replaces an existing node with a new node
     * @param     info, contents of the new node
```

```
* @param     node, node to be replaced. When zero, the root is assumed
* @param     alignRight, only for MOVE_EXISTING. If true, node will be
*            the right child of the new node. Otherwise, it will be the
*            left.
* @param     replaceBehavior, one of:
*            ABORT_ON_EXISTING - undefined for replace()
*            MOVE_EXISTING - make node a child of the new node,
*                            satisfies preferRight
*            DELETE_EXISTING - remove node completely
* @return    pointer to the inserted TreeNode, replace() is always
*            successful
* @pre       If the tree is empty, a root node will be created with info
*            as it contents
* @pre       The instance pointed to by node should be part of the
*            called instance of Tree
* @post      Ownership is assumed on the new node. When DELETE_EXISTING
*            is specified, the entire subtree pointed to by node is
*            deleted first.
**/
virtual node_t* replace( const INFO_T& info,
                 node_t* node =0,
                 bool alignRight =false,
                 int replaceBehavior =DELETE_EXISTING ) {
    assert( replaceBehavior != ABORT_ON_EXISTING );

    node_t *newnode =new node_t( info );
    if( !node )
        node =m_root;
    if( !node )
        return pushBack( info );

    if( node->parent( ) ) {
        newnode->setParent( node->parent( ) );
        if( node->parent( )->leftChild( ) == node )
            node->parent( )->setLeftChild( newnode );
        else
            node->parent( )->setRightChild( newnode );
    } else
        m_root =newnode;

    if( replaceBehavior == DELETE_EXISTING ) {

        deleteRecursive( node );
    }
    else if( replaceBehavior == MOVE_EXISTING ) {
        if( alignRight )
            newnode->setRightChild( node );
        else
            newnode->setLeftChild( node );
        node->setParent( newnode );
    }
    return node;
}

```

```
        /**
         * @function  remove( )
         * @abstract  removes and deletes node or subtree
         * @param     n, node or subtree to be removed and deleted
         * @post      after remove(), n points to an invalid address
         **/
         virtual void remove( node_t *n ) {
             if( !n )
                 return;
             if( n->parent( ) ) {
                 if( n->parent( )->leftChild( ) == n )
                     n->parent( )->setLeftChild( 0 );
                 else if( n->parent( )->rightChild( ) == n )
                     n->parent( )->setRightChild( 0 );
             }
             deleteRecursive( n );
         }

        /**
         * @function  clear( )
         * @abstract  clears entire tree
         * @pre       tree may be empty
         * @post      all nodes and data are deallocated
         **/
         void clear( ) {
             deleteRecursive( m_root );
             m_root =0;
         }

        /**
         * @function  empty( )
         * @abstract  test if tree is empty
         * @return    true when empty
         **/
         bool isEmpty( ) const {
             return !m_root;
         }

       /**
         * @function  root( )
         * @abstract  returns address of the root of the tree
         * @return    the adress of the root of the tree is returned
         * @pre       there needs to be a tree
         **/
         node_t* root( ){
             return m_root;
         }

        /**
         * @function  row( )
         * @abstract  returns an entire row/level in the tree
         * @param     level, the desired row. Zero gives just the root.
         * @return    a list containing all node pointers in that row
         * @pre       level must be positive or zero
```

```
     * @post
     **/
     nodelist row( int level ) {
         nodelist rlist;
         getRowRecursive( m_root, rlist, level );
         return rlist;
     }

    /**
     * @function  find( )
     * @abstract  find the first occurrence of info and returns its node ptr
     * @param     haystack, the root of the (sub)tree we want to look in
     *            null if we want to start at the root of the tree
     * @param     needle, the needle in our haystack
     * @return    a pointer to the first occurrence of needle
     * @post      there may be multiple occurrences of needle, we only return
     *            one. A null-pointer is returned if no needle is found
     **/
     virtual node_t* find( node_t* haystack, const INFO_T& needle ) {
         if( haystack == 0 ) {
                 if( m_root )
                     haystack =m_root;
                 else
                     return 0;
         }
         return findRecursive( haystack, needle );
     }

    /**
     * @function  contains( )
     * @abstract  determines if a certain content (needle) is found
     * @param     haystack, the root of the (sub)tree we want to look in
     *            null if we want to start at the root of the tree
     * @param     needle, the needle in our haystack
     * @return    true if needle is found
     **/
     bool contains( node_t* haystack, const INFO_T& needle ) {
         return find( haystack, needle );
     }

    /**
     * @function  toDot( )
     * @abstract  writes tree in Dot-format to a stream
     * @param     out, ostream to write to
     * @pre       out must be a valid stream
     * @post      out (file or cout) with the tree in dot-notation
     **/
     void toDot( ostream& out, const string & graphName ) {
         if( isEmpty( ) )
             return;
         map<node_t *, int> adresses;
         typename map< node_t *, int >::iterator adrIt;
         int i =1;
         int p;
```

```
        iterator_pre it;
        iterator_pre tempit;
        adresses[m_root] =0;
        out << "digraph " << graphName << '{' << endl << '"' << 0 << '"';
        for( it =begin_pre( ); it != end_pre( ); ++it ) {
            adrIt =adresses.find( &(*it) );
            if( adrIt == adresses.end( ) ) {
                adresses[&(*it)] =i;
                p =i;
                i ++;
            }
            if( (&(*it))->parent( ) != &(*tempit) )
              out << ';' << endl << '"'
                  << adresses.find( (&(*it))->parent( ))->second << '"';
            if( (&(*it)) != m_root )
                out << " -> \"" << p << '"';
            tempit =it;
        }
        out << ';' << endl;
        for ( adrIt =adresses.begin( ); adrIt != adresses.end( ); ++adrIt )
            out << adrIt->second << " [label=\""
                << adrIt->first->info( ) << "\"]";
        out << '}';
    }

   /**
    * @function  copyFromNode( )
    * @abstract  copies the the node source and its children to the node
    *            dest
    * @param     source, the node and its children that need to be copied
    * @param     dest, the node who is going to get the copied children
    * @param     left, this is true if it's a left child.
    * @pre       there needs to be a tree and we can't copy to a root.
    * @post      the subtree that starts at source is now also a child of
    *            dest
    **/
    void copyFromNode( node_t *source, node_t *dest, bool left ) {
        if (!source)
            return;
        node_t *acorn =new node_t( dest );
        if(left) {
            if( dest->leftChild( ))
                return;
            dest->setLeftChild( acorn );
        }
        else {
            if( dest->rightChild( ))
                return;
            dest->setRightChild( acorn );
        }
        cloneRecursive( source, acorn );
    }

    Tree<INFO_T>& operator=( const Tree<INFO_T>& tree ) {
        clear( );
        if( tree.m_root ) {
            m_root =new node_t( (node_t*)0 );
            cloneRecursive( tree.m_root, m_root );
        }
        return *this;
    }

protected:
   /**
    * @function  cloneRecursive( )
    * @abstract  cloning a subtree to a node
    * @param     source, the node we want to start the cloning process from
    * @param     dest, the node we want to clone to
    * @post      the subtree starting at source is cloned to the node dest
    **/
    void cloneRecursive( node_t *source, node_t* dest ) {
        dest->info() =source->info();
        if( source->leftChild( ) ) {
            node_t *left =new node_t( dest );
            dest->setLeftChild( left );
            cloneRecursive( source->leftChild( ), left );
        }
        if( source->rightChild( ) ) {
            node_t *right =new node_t( dest );
            dest->setRightChild( right );
            cloneRecursive( source->rightChild( ), right );
        }
    }

   /**
    * @function  deleteRecursive( )
    * @abstract  delete all nodes of a given tree
    * @param     root, starting point, is deleted last
    * @post      the subtree has been deleted
    **/
    void deleteRecursive( node_t *root ) {
        if( !root )
            return;
        deleteRecursive( root->leftChild( ) );
        deleteRecursive( root->rightChild( ) );
        delete root;
    }

   /**
    * @function  getRowCountRecursive( )
    * @abstract  calculate the maximum depth/row count in a subtree
    * @param     root, starting point
    * @param     level, starting level
    * @return    maximum depth/rows in the subtree
    **/
    int getRowCountRecursive( node_t* root, int level ) {
        if( !root )
            return level;
        return max(
                getRowCountRecursive( root->leftChild( ), level+1 ),
                getRowCountRecursive( root->rightChild( ), level+1 ) );
    }

   /**
    * @function  getRowRecursive( )
    * @abstract  compile a full list of one row in the tree
    * @param     root, starting point
    * @param     rlist, reference to the list so far
    * @param     level, how many level still to go
    * @post      a list of a row in the tree has been made.
    **/
    void getRowRecursive( node_t* root, nodelist &rlist, int level ) {
        // Base-case
        if( !level ) {
            rlist.push_back( root );
        } else if( root ){
            level--;
            if( level && !root->leftChild( ) )
                for( int i =0; i < (level<<1); ++i )
                    rlist.push_back( 0 );
            else
               getRowRecursive( root->leftChild( ), rlist, level );

            if( level && !root->rightChild( ) )
                for( int i =0; i < (level<<1); ++i )
                    rlist.push_back( 0 );
            else
                getRowRecursive( root->rightChild( ), rlist, level );
        }
    }

    /**
    * @function  findRecursive( )
    * @abstract  first the first occurrence of needle and return its node
    *            ptr
    * @param     haystack, root of the search tree
    * @param     needle, copy of the data to find
    * @return    the node that contains the needle
    **/
    node_t *findRecursive( node_t* haystack, const INFO_T &needle ) {
        if( haystack->info( ) == needle )
            return haystack;

        node_t *n =0;
        if( haystack->leftChild( ) )
            n =findRecursive( haystack->leftChild( ), needle );
        if( !n && haystack->rightChild( ) )
            n =findRecursive( haystack->rightChild( ), needle );
        return n;
    }

    friend class TreeNodeIterator_pre<INFO_T>;
    friend class TreeNodeIterator_in<INFO_T>;
    friend class SplayTree<INFO_T>;
    TreeNode<INFO_T> *m_root;

private:
    /**
    * @function  getFirstEmptySlot( )
    * @abstract  when a row has a continuous empty space on the right,
    *            find the left-most parent in the above row that has
    *            at least one empty slot.
    * @param     level, how many level still to go
    * @return    the first empty slot where we can put a new node
    * @pre       level should be > 1
    **/
    node_t *getFirstEmptySlot( int level ) {
        node_t *p =0;
        nodelist rlist =row( level-1 ); // we need the parents of this level
        /** changed auto to int **/
        for( auto it =rlist.rbegin( ); it !=rlist.rend( ); ++it ) {
            if( !(*it)->hasChildren( ) )
                p =(*it);
            else if( !(*it)->rightChild( ) ) {
                p =(*it);
                break;
            } else
                break;
        }
        return p;
    }
};

#endif
```

6.7 TreeNode.h

```
/**
 * Treenode:
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    Treenode.h
 * @date    26-10-2014
 **/

#ifndef TREENODE_H
#define TREENODE_H

using namespace std;

template <class INFO_T> class Tree;
class ExpressionTree;

template <class INFO_T> class TreeNode
{
```

```
public:
   /**
    * @function  TreeNode( )
    * @abstract  Constructor, creates a node
    * @param     info, the contents of a node
    * @param     parent, the parent of the node
    * @post      A node has been created.
    **/
    TreeNode( const INFO_T& info, TreeNode<INFO_T>* parent =0 )
        : m_lchild( 0 ), m_rchild( 0 ) {
        m_info =info;
        m_parent =parent;
    }

   /**
    * @function  TreeNode( )
    * @abstract  Constructor, creates a node
    * @param     parent, the parent of the node
    * @post      A node has been created.
    **/
    TreeNode( TreeNode<INFO_T>* parent =0 )
        : m_lchild( 0 ), m_rchild( 0 ) {
        m_parent =parent;
    }

   /**
    * @function  =
    * @abstract  Sets a nodes content to N
    * @param     n, the contents you want the node to have
    * @post      The node now has those contents.
    **/
    void operator =( INFO_T n ) { m_info =n; }

   /**
    * @function  INFO_T( ), info( )
    * @abstract  Returns the content of a node
    * @return    m_info, the contents of the node
    **/
    operator INFO_T( ) const { return m_info; }
    const INFO_T &info( ) const { return m_info; }
    INFO_T &info( ) { return m_info; }
    /**
    * @function  atRow( )
    * @abstract  returns the level or row-number of this node
    * @return    row, an int of row the node is at
    **/
    int atRow( ) const {
        const TreeNode<INFO_T> *n =this;
        int row =0;
        while( n->parent( ) ) {
            n =n->parent( );
            row++;
        }
        return row;
    }

   /**
    * @function  parent( ), leftChild( ), rightChild( )
    * @abstract  returns the adress of the parent, left child and right
    *            child respectively
    * @return    the adress of the requested family member of the node
    **/
    TreeNode<INFO_T> *parent( ) const { return m_parent; }
    TreeNode<INFO_T> *leftChild( ) const { return m_lchild; }
    TreeNode<INFO_T> *rightChild( ) const { return m_rchild; }

    /**
    * @function  swapWith( )
    * @abstract  Swaps this node with another node in the tree
    * @param     n, the node to swap this one with
    * @pre       both this node and n must be in the same parent tree
    * @post      n will have the parent and children of this node
    *            and vice verse. Both nodes retain their data.
    **/
    void swapWith( TreeNode<INFO_T>* n ) {
        bool this_wasLeftChild =false, n_wasLeftChild =false;
        if( parent( ) && parent( )->leftChild( ) == this )
            this_wasLeftChild =true;
        if( n->parent( ) && n->parent( )->leftChild( ) == n )
            n_wasLeftChild =true;

        // Swap the family info
        TreeNode<INFO_T>* newParent =
            ( n->parent( ) == this ) ? n : n->parent( );
        TreeNode<INFO_T>* newLeft =
            ( n->leftChild( ) == this ) ? n :n->leftChild( );
        TreeNode<INFO_T>* newRight =
             ( n->rightChild( ) == this ) ? n :n->rightChild( );

        n->setParent( parent( ) == n ? this : parent( ) );
        n->setLeftChild( leftChild( ) == n ? this : leftChild( ) );
        n->setRightChild( rightChild( ) == n ? this : rightChild( ) );

        setParent( newParent );
        setLeftChild( newLeft );
        setRightChild( newRight );

        // Restore applicable pointers
        if( n->leftChild( ) )
            n->leftChild( )->setParent( n );
        if( n->rightChild( ) )
            n->rightChild( )->setParent( n );
        if( leftChild( ) )
            leftChild( )->setParent( this );
        if( rightChild( ) )
            rightChild( )->setParent( this );
        if( n->parent( ) ) {
            if( this_wasLeftChild )
                n->parent( )->setLeftChild( n );
            else
                n->parent( )->setRightChild( n );
        }
        if( parent( ) ) {
            if( n_wasLeftChild )
                parent( )->setLeftChild( this );
            else
                parent( )->setRightChild( this );
        }
    }

    /**
    * @function  replace( )
    * @abstract  Replaces the node with another node in the tree
    * @param     n, the node we replace the node with, this one gets deleted
    * @pre       both this node and n must be in the same parent tree
    * @post      The node will be replaced and n will be deleted.
    **/
    void replace( TreeNode<INFO_T>* n ) {
        bool n_wasLeftChild =false;

        if( n->parent( ) && n->parent( )->leftChild( ) == n )
            n_wasLeftChild =true;

        // Swap the family info
        TreeNode<INFO_T>* newParent =
            ( n->parent( ) == this ) ? n : n->parent( );
        TreeNode<INFO_T>* newLeft =
            ( n->leftChild( ) == this ) ? n :n->leftChild( );
        TreeNode<INFO_T>* newRight =
             ( n->rightChild( ) == this ) ? n :n->rightChild( );

        setParent( newParent );
        setLeftChild( newLeft );
        setRightChild( newRight );
        m_info = n->m_info;

        // Restore applicable pointers
        if( leftChild( ) )
            leftChild( )->setParent( this );
        if( rightChild( ) )
            rightChild( )->setParent( this );

        if( parent( ) ) {
            if( n_wasLeftChild )
                parent( )->setLeftChild( this );
            else
                parent( )->setRightChild( this );
        }
        delete n;
    }

    /**
```

```
        * @function  sibling( )
        * @abstract  returns the address of the sibling
        * @return    the address to the sibling or zero if there is no sibling
        **/
        TreeNode<INFO_T>* sibling( ) {
            if( parent( )->leftChild( ) == this )
                return parent( )->rightChild( );
            else if( parent( )->rightChild( ) == this )
                return parent( )->leftChild( );
            else
                return 0;
        }

       /**
        * @function  hasChildren( ), hasParent( ), isFull( )
        * @abstract  Returns whether the node has children, has parents or is
        *            full (has two children) respectively
        * @param
        * @return    true or false, depending on what is requested from the node.
        *            if hasChildren is called and the node has children, it will
        *            return true, otherwise false.
        *            If hasParent is called and the node has a parent, it will
        *            return true, otherwise false.
        *            If isFull is called and the node has two children, it will
        *            return true, otherwise false.
        **/
        bool hasChildren( ) const { return m_lchild || m_rchild; }
        bool hasParent( ) const { return m_parent; }
        bool isFull( ) const { return m_lchild && m_rchild; }

    protected:
        friend class Tree<INFO_T>;
        friend class ExpressionTree;

       /**
        * @function  setParent( ), setLeftChild( ), setRightChild( )
        * @abstract  sets the parent, left child and right child of the
        *            particular node respectively
        * @param     p, the node we want to set a certain family member of
        * @return    void
        * @post      The node now has a parent, a left child or a right child
        *            respectively.
        **/
        void setParent( TreeNode<INFO_T> *p ) { m_parent =p; }
        void setLeftChild( TreeNode<INFO_T> *p ) { m_lchild =p; }
        void setRightChild( TreeNode<INFO_T> *p ) { m_rchild =p; }

    private:
        INFO_T m_info;
        TreeNode<INFO_T> *m_parent;
        TreeNode<INFO_T> *m_lchild;
        TreeNode<INFO_T> *m_rchild;
};

```

```
/**
* @function  <<
* @abstract  the contents of the node are returned
* @param     out, in what format we want to get the contents
* @param     rhs, the node of which we want the contents
* @return    the contents of the node.
**/
template <class INFO_T> ostream &operator <<(ostream& out, const TreeNode<INFO_T> & r
    out << rhs.info( );
    return out;
}

#endif
```

6.8 TreeNodeIterator.h

```
/**
 * TreeNodeIterator: Provides a set of iterators that follow the STL-standard
 *
 * @author  Micky Faas (s1407937)
 * @author  Lisette de Schipper (s1396250)
 * @file    TreeNodeIterator.h
 * @date    26-10-2014
 **/

#include <iterator>
#include "TreeNode.h"

template <class INFO_T> class TreeNodeIterator
                        : public std::iterator<std::forward_iterator_tag,
                                               TreeNode<INFO_T>> {
    public:
        typedef TreeNode<INFO_T> node_t;

       /**
        * @function  TreeNodeIterator( )
        * @abstract  (copy)constructor
        * @pre       TreeNodeIterator is abstract and cannot be constructed
        **/
        TreeNodeIterator( node_t* ptr =0 ) : p( ptr ) { }
        TreeNodeIterator( const TreeNodeIterator& it ) : p( it.p ) { }

       /**
        * @function  (in)equality operator overload
        * @abstract  Test (in)equality for two TreeNodeIterators
        * @param     rhs, right-hand side of the comparison
        * @return    true if both iterators point to the same node (==)
        *            false if both iterators point to the same node (!=)
        **/
        bool operator == (const TreeNodeIterator& rhs) { return p==rhs.p; }
        bool operator != (const TreeNodeIterator& rhs) { return p!=rhs.p; }

       /**
        * @function  operator*( )
        * @abstract  Cast operator to node_t reference
        * @return    The value of the current node
        * @pre       Must point to a valid node
        **/
        node_t& operator*( ) { return *p; }

       /**
        * @function  operator++( )
        * @abstract  pre- and post increment operators
        * @return    TreeNodeIterator that has iterated one step
        **/
        TreeNodeIterator &operator++( ) { next( ); return *this; }
        TreeNodeIterator operator++( int )
            { TreeNodeIterator tmp( *this ); operator++( ); return tmp; }
    protected:

       /**
        * @function  next( ) //(pure virtual)
        * @abstract  Implement this function to implement your own iterator
        */
        virtual bool next( ){ return false; }// =0;
        node_t *p;
};

template <class INFO_T> class TreeNodeIterator_pre
                        : public TreeNodeIterator<INFO_T> {
    public:
        typedef TreeNode<INFO_T> node_t;

        TreeNodeIterator_pre( node_t* ptr =0 )
            : TreeNodeIterator<INFO_T>( ptr ) { }
        TreeNodeIterator_pre( const TreeNodeIterator<INFO_T>& it )
            : TreeNodeIterator<INFO_T>( it ) { }
        TreeNodeIterator_pre( const TreeNodeIterator_pre& it )
            : TreeNodeIterator<INFO_T>( it.p ) { }

        TreeNodeIterator_pre &operator++( ) { next( ); return *this; }
        TreeNodeIterator_pre operator++( int )
            { TreeNodeIterator_pre tmp( *this ); operator++( ); return tmp; }

    protected:
        using TreeNodeIterator<INFO_T >::p;

       /**
        * @function  next( )
        * @abstract  Takes one step in pre-order traversal
        * @return    returns true if such a step exists
        */
        bool next( ) {
            if( !p )
                return false;
            if( p->hasChildren( ) ) { // a possible child that can be the next
                p =p->leftChild( ) ? p->leftChild( ) : p->rightChild( );
                return true;
            }
            else if( p->hasParent( ) // we have a right brother
                    && p->parent( )->rightChild( )
                    && p->parent( )->rightChild( ) != p ) {
                p =p->parent( )->rightChild( );
                return true;
            }
            else if( p->hasParent( ) ) { // just a parent, thus we go up
                TreeNode<INFO_T> *tmp =p->parent( );
                while( tmp->parent( ) ) {
                    if( tmp->parent( )->rightChild( )
                            && tmp->parent( )->rightChild( ) != tmp ) {
                        p =tmp->parent( )->rightChild( );
                        return true;
                    }
                    tmp =tmp->parent( );
                }
            }
            // Nothing left
            p =0;
            return false;
        }

};

template <class INFO_T> class TreeNodeIterator_in
                        : public TreeNodeIterator<INFO_T>{
    public:
        typedef TreeNode<INFO_T> node_t;

        TreeNodeIterator_in( node_t* ptr =0 )
            : TreeNodeIterator<INFO_T>( ptr ) { }
        TreeNodeIterator_in( const TreeNodeIterator<INFO_T>& it )
            : TreeNodeIterator<INFO_T>( it ) { }
        TreeNodeIterator_in( const TreeNodeIterator_in& it )
            : TreeNodeIterator<INFO_T>( it.p ) { }

        TreeNodeIterator_in &operator++( ) { next( ); return *this; }
        TreeNodeIterator_in operator++( int )
            { TreeNodeIterator_in tmp( *this ); operator++( ); return tmp; }

    protected:
        using TreeNodeIterator<INFO_T >::p;
       /**
        * @function  next( )
        * @abstract  Takes one step in in-order traversal
        * @return    returns true if such a step exists
        */
        bool next( ) {
            if( p->rightChild( ) ) {
                p =p->rightChild( );
                while( p->leftChild( ) )
                    p =p->leftChild( );
                return true;
            }
            else if( p->parent( ) && p->parent( )->leftChild( ) == p ) {
                p =p->parent( );
                return true;
            } else if( p->parent( ) && p->parent( )->rightChild( ) == p ) {
                p =p->parent( );
                while( p->parent( ) && p == p->parent( )->rightChild( ) ) {
                    p =p->parent( );
                }
                if( p )
                    p =p->parent( );
                if( p )
                    return true;
                else
                    return false;
            }
            // Er is niks meer
            p =0;
            return false;
        }
};

template <class INFO_T> class TreeNodeIterator_post
                        : public TreeNodeIterator<INFO_T>{
    public:
        typedef TreeNode<INFO_T> node_t;

        TreeNodeIterator_post( node_t* ptr =0 )
            : TreeNodeIterator<INFO_T>( ptr ) { }
        TreeNodeIterator_post( const TreeNodeIterator<INFO_T>& it )
            : TreeNodeIterator<INFO_T>( it ) { }
        TreeNodeIterator_post( const TreeNodeIterator_post& it )
            : TreeNodeIterator<INFO_T>( it.p ) { }

        TreeNodeIterator_post &operator++( ) { next( ); return *this; }
        TreeNodeIterator_post operator++( int )
            { TreeNodeIterator_post tmp( *this ); operator++( ); return tmp; }

    protected:
        using TreeNodeIterator<INFO_T >::p;
       /**
        * @function  next( )
        * @abstract  Takes one step in post-order traversal
        * @return    returns true if such a step exists
        */
        bool next( ) {

            if( p->hasParent( ) // We have a right brother
                    && p->parent( )->rightChild( )
                    && p->parent( )->rightChild( ) != p ) {
                p =p->parent( )->rightChild( );
                while( p->leftChild( ) )
                    p =p->leftChild( );
                return true;
            } else if( p->parent( ) ) {
                p =p->parent( );
                return true;
            }
            // Nothing left
            p =0;
            return false;
        }
};
```