My Project

4

Generated by Doxygen 1.8.14

Contents

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

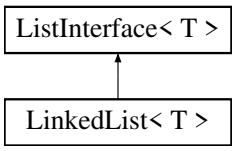
Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

Class Documentation

3.1 LinkedList< T > Class Template Reference

Inheritance diagram for LinkedList< T >:



Public Member Functions

- LinkedList (const LinkedList< T > &rhs)
- LinkedList & operator= (LinkedList rhs)
- bool isEmpty () const
- int getLength () const
- bool insert (int newPosition, const T &newEntry)
- bool remove (int position)
- void clear ()
- T getEntry (int position) const
- T replace (int position, const T &newEntry)
- void swap (LinkedList &lhs, LinkedList &rhs)

3.1.1 Detailed Description

template<class T>
class LinkedList< T >

This is a LinkedList class. It uses Nodes to store data.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 LinkedList()

```
template<typename T>
LinkedList< T >::LinkedList (
            const LinkedList< T > & rhs )
```

This is the copy constructor. It make a copy of the parameter. It is also used by the operator= in the copy-swap paradigm.

Parameters

rhs	- the LinkedList we are copying during construction

3.1.3 Member Function Documentation

3.1.3.1 clear()

```
template<typename T >
void LinkedList< T >::clear ( )  [virtual]
```

Removes all entries from this list.

Postcondition

The list contains no entries and the count of items is 0.

Implements ListInterface< T >.

3.1.3.2 getEntry()

```
template<typename T >
T LinkedList< T >::getEntry (
            int position ) const  [virtual]
```

Exceptions

PrecondViolatedExcept	if position < 1 or position > getLength().

Implements ListInterface< T >.

3.1.3.3 getLength()

```
template<typename T >
int LinkedList< T >::getLength ( ) const  [virtual]
```

Gets the current number of entries in this list.

Returns

The integer number of entries currently in the list.

Implements ListInterface< T >.

3.1.3.4 insert()

```
template<typename T>
bool LinkedList< T >::insert (
            int newPosition,
            const T & newEntry )  [virtual]
```

This will insert the node in the list. Position is 1 based Position must be at least 1 and not more than count+1. Items goes at position.

Implements ListInterface< T >.

3.1.3.5 isEmpty()

```
template<typename T >
bool LinkedList< T >::isEmpty ( ) const  [virtual]
```

Sees whether this list is empty.

Returns

True if the list is empty; otherwise returns false.

Implements ListInterface< T >.

3.1.3.6 operator=()

```
template<typename T >
LinkedList< T > & LinkedList< T >::operator= (
            LinkedList< T > rhs )
```

This is the assignment operator. It uses the copy-swap paradigm to create a copy of the parameter

Parameters

rhs	- the LinkedList we are assigning to this

Returns

a reference to the list that was copied into, a.k.a. ∗this

3.1.3.7 remove()

```
template<typename T >
bool LinkedList< T >::remove (
            int position )  [virtual]
```

Removes the entry at a given position from this list.

Precondition

None.

Postcondition

If 1 <= position <= getLength() and the removal is successful, the entry at the given position in the list is removed, other items are renumbered accordingly, and the returned value is true.

Parameters

position	The list position of the entry to remove.

Returns

True if the removal is successful, or false if not.

Implements ListInterface< T >.

3.1.3.8 replace()

```
template<typename T>
T LinkedList< T >::replace (
            int position,
            const T & newEntry )  [virtual]
```

Exceptions

PrecondViolatedExcept	if position < 1 or position > getLength().

Implements ListInterface< T >.

3.1.3.9 swap()

```
template<typename T >
void LinkedList< T >::swap (
            LinkedList< T > & lhs,
            LinkedList< T > & rhs )
```

This is the swap method. It will swap the internals of the two lists. Notably it is used in the operator= to implement the copy swap paradigm. It is also used by other C++ paradigms.

Parameters

lhs	- the LinkedList on the left...Left Hand Side (lhs)
rhs	- the LinkedList on the right...Right Hand Side (rhs)

3.2 ListInterface< ItemType > Class Template Reference

Public Member Functions

- virtual bool isEmpty () const =0
- virtual int getLength () const =0
- virtual bool insert (int newPosition, const ItemType &newEntry)=0
- virtual bool remove (int position)=0
- virtual void clear ()=0
- virtual ItemType getEntry (int position) const =0
- virtual ItemType replace (int position, const ItemType &newEntry)=0
- virtual ~ListInterface ()

3.2.1 Detailed Description

template<class ItemType>
class ListInterface< ItemType >

This is the basis for a List. It's a pure virtual abstract class meant to be an interface

3.2.2 Constructor & Destructor Documentation

3.2.2.1 ~ListInterface()

```
template<class ItemType>
virtual ListInterface< ItemType >::~ListInterface ( )  [inline], [virtual]
```

Destroys this list and frees its assigned memory.

3.2.3 Member Function Documentation

3.2.3.1 clear()

```
template<class ItemType>
virtual void ListInterface< ItemType >::clear ( )  [pure virtual]
```

Removes all entries from this list.

Postcondition

The list contains no entries and the count of items is 0.

Implemented in LinkedList< T >, and LinkedList< Term >.

3.2.3.2 getEntry()

```
template<class ItemType>
virtual ItemType ListInterface< ItemType >::getEntry (
            int position ) const  [pure virtual]
```

Gets the entry at the given position in this list.

Precondition

1 <= position <= getLength().

Postcondition

The desired entry has been returned.

Parameters

position	The list position of the desired entry.

Returns

The entry at the given position.

Implemented in LinkedList< T >, and LinkedList< Term >.

3.2.3.3 getLength()

```
template<class ItemType>
virtual int ListInterface< ItemType >::getLength ( ) const  [pure virtual]
```

Gets the current number of entries in this list.

Returns

The integer number of entries currently in the list.

Implemented in LinkedList< T >, and LinkedList< Term >.

3.2.3.4 insert()

```
template<class ItemType>
virtual bool ListInterface< ItemType >::insert (
             int newPosition,
             const ItemType & newEntry )  [pure virtual]
```

Inserts an entry into this list at a given position.

Precondition

None.

Postcondition

If 1 <= position <= getLength() + 1 and the insertion is successful, newEntry is at the given position in the list, other entries are renumbered accordingly, and the returned value is true.

Parameters

newPosition	The list position at which to insert newEntry.
newEntry	The entry to insert into the list.

Returns

True if the insertion is successful, or false if not.

Implemented in LinkedList< T >, and LinkedList< Term >.

3.2.3.5 isEmpty()

```
template<class ItemType>
virtual bool ListInterface< ItemType >::isEmpty ( ) const  [pure virtual]
```

Sees whether this list is empty.

Returns

True if the list is empty; otherwise returns false.

Implemented in LinkedList< T >, and LinkedList< Term >.

3.2.3.6 remove()

```
template<class ItemType>
virtual bool ListInterface< ItemType >::remove (
            int position )  [pure virtual]
```

Removes the entry at a given position from this list.

Precondition

None.

Postcondition

If 1 <= position <= getLength() and the removal is successful, the entry at the given position in the list is removed, other items are renumbered accordingly, and the returned value is true.

Parameters

position	The list position of the entry to remove.

Returns

True if the removal is successful, or false if not.

Implemented in LinkedList< T >, and LinkedList< Term >.

3.2.3.7 replace()

```
template<class ItemType>
virtual ItemType ListInterface< ItemType >::replace (
            int position,
            const ItemType & newEntry )  [pure virtual]
```

Replaces the entry at the given position in this list.

Precondition

1 <= position <= getLength().

Postcondition

The entry at the given position is newEntry.

Parameters

position	The list position of the entry to replace.
newEntry	The replacement entry.

Returns

The replaced entry.

Implemented in LinkedList< T >, and LinkedList< Term >.

3.3 Node< T > Class Template Reference

Public Member Functions

- Node ()
- Node (T data)
- Node (T data, Node *next)
- void setNext (Node *next)
- void setData (T data)
- Node * getNext () const
- T getData () const

3.3.1 Detailed Description

template<typename T>
class Node< T >

This is a basic Node class that can used by any singly linked data structure. It has a data item and a pointer to the next Node.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 Node() [1/3]

```
template<typename T>
Node< T >::Node ( )  [inline]
```

This constructor sets the data to the default value for T and next to nullptr

3.3.2.2 Node() [2/3]

```
template<typename T>
Node< T >::Node (
            T data )  [inline]
```

This uses the parameter to initialize the data and sets next to the nullptr

Parameters

data	the data value to store in the data field

3.3.2.3 Node() [3/3]

```
template<typename T>
Node< T >::Node (
            T data,
            Node< T > * next )  [inline]
```

This uses the parameters to initialize the data and next pointer

Parameters

data	the data value to store in the data field
next	the value to use for the next pointer

3.3.3 Member Function Documentation

3.3.3.1 getData()

```
template<typename T>
T Node< T >::getData ( ) const  [inline]
```

This will return the data value from the Node

Returns

the value of the data in the Node

3.3.3.2 getNext()

```
template<typename T>
Node* Node< T >::getNext ( ) const  [inline]
```

This will return the next pointer from the Node

Returns

the value of the next pointer in the Node

3.3.3.3 setData()

```
template<typename T>
void Node< T >::setData (
            T data )  [inline]
```

This will set the data value to the value given in the parameter

Parameters

data	the value to use for the data field

3.3.3.4 setNext()

```
template<typename T>
void Node< T >::setNext (
            Node< T > * next )  [inline]
```

This will set the next pointer to the value given in the parameter

Parameters

next	the value to use for the next pointer

3.4 Polynomial Class Reference

Public Member Functions

- Polynomial ()
- void insert_term (Term term_to_insert)
- Polynomial operator+ (const Polynomial &rhs)
- Polynomial operator* (const Polynomial &rhs)
- Polynomial operator- (const Polynomial &rhs)
- Polynomial operator/ (const Polynomial &rhs)
- Polynomial operator% (const Polynomial &rhs)

Friends

- std::ostream & operator<< (std::ostream &out, const Polynomial &p)

3.4.1 Detailed Description

This is a polynomial class. It represents a polynomial in algebra. It will contain a LinkedList of Terms.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 Polynomial()

```
Polynomial::Polynomial ( )
```

Constructor

3.4.3 Member Function Documentation

3.4.3.1 insert_term()

```
void Polynomial::insert_term (
            Term term_to_insert )
```

This will insert the Term into the LinkedList of Terms. It should insert the term in descending order of power, so the highest order Term is in position 1.

3.4.3.2 operator%()

```
Polynomial Polynomial::operator% (
            const Polynomial & rhs )
```

This will find the remainder of two Polynomials. The result will be the remainder of the two Polynomials.
this is the pointer that points to the Polynomial on the Left Hand Side of the % sign and rhs is the Polynomial on the Right Hand Side of the % sign I would recommend this url for help with the remainder algorithm https://rosettacode.org/wiki/Polynomial_long_division

Parameters

rhs	- this is the Polynomial on the Right Hand Side of the % sign

Returns

the result of the remainder of the two polynomials

Exceptions

if	the degree of the divisor is < 0

3.4.3.3 operator∗()

```
Polynomial Polynomial::operator* (
            const Polynomial & rhs )
```

This will multiply two Polynomials together. The result will be the multiplication of the two Polynomials.
this is the pointer that points to the Polynomial on the Left Hand Side of the $*$ sign and rhs is the Polynomial on the Right Hand Side of the $*$ sign

Parameters

rhs	- this is the Polynomial on the Right Hand Side of the $*$ sign

Returns

the result of the multiplication of the two polynomials

3.4.3.4 operator+()

```
Polynomial Polynomial::operator+ (
            const Polynomial & rhs )
```

This will add two Polynomials together. The result will be the addition of the two Polynomials.
this is the pointer that points to the Polynomial on the Left Hand Side of the + sign and rhs is the Polynomial on the Right Hand Side of the + sign

Parameters

rhs	- this is the Polynomial on the Right Hand Side of the + sign

Returns

the result of the addition of the two polynomials

3.4.3.5 operator-()

```
Polynomial Polynomial::operator- (
            const Polynomial & rhs )
```

This will subtract two Polynomials together. The result will be the subtraction of the two Polynomials.
this is the pointer that points to the Polynomial on the Left Hand Side of the - sign and rhs is the Polynomial on the Right Hand Side of the - sign

Parameters

rhs	- this is the Polynomial on the Right Hand Side of the - sign

Returns

the result of the subtraction of the two polynomials

3.4.3.6 operator/()

```
Polynomial Polynomial::operator/ (
            const Polynomial & rhs )
```

This will divide two Polynomials together. The result will be the division of the two Polynomials.
this is the pointer that points to the Polynomial on the Left Hand Side of the / sign and rhs is the Polynomial on the Right Hand Side of the / sign I would recommend this url for help with the division algorithm https://rosettacode.org/wiki/Polynomial_long_division

Parameters

rhs	- this is the Polynomial on the Right Hand Side of the / sign

Returns

the result of the division of the two polynomials

Exceptions

if	the degree of the divisor is < 0

3.4.4 Friends And Related Function Documentation

3.4.4.1 operator<<

```
std::ostream& operator<< (
            std::ostream & out,
            const Polynomial & p )  [friend]
```

This will output the given polynomial on the given ostream. If the polynomial contains the terms 3x^2, -4x^1, and 1x^0, then this would be the output:

```
3x^2-4x+1
```

It should use the Term's operator<< to output each term. No space should be between each term and after the first term if the coefficient is positive there should be a +.

Parameters

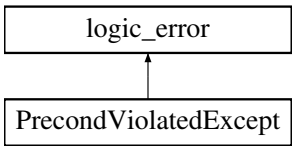
out	-the ostream to output the polynomial on
p	- the Polynomial to output

Returns

the ostream that was passed

3.5 PrecondViolatedExcept Class Reference

Inheritance diagram for PrecondViolatedExcept:



Public Member Functions

- PrecondViolatedExcept (const std::string &message="")

3.5.1 Detailed Description

An exception classed used by the LinkedList to throw when a precondition is not met.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 PrecondViolatedExcept()

```
PrecondViolatedExcept::PrecondViolatedExcept (
            const std::string & message = "" )
```

Constructor for the class

Parameters

message	the message to send to the exception

3.6 Term Class Reference

Public Member Functions

- Term (double coefficient=0, int power=0)
- double get_coefficient () const
- int get_power () const

Friends

- std::ostream & operator<< (std::ostream &out, const Term &t)

3.6.1 Detailed Description

This class represents a term in a polynomial. It stores a coefficient and a power.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 Term()

```
Term::Term (
            double coefficient = 0,
            int power = 0 )
```

Constructor. Set the coefficient and power to the given values. Hint: Only put the default values in the header

Parameters

coefficient	- the double value you want to store in the term as the coefficient
power	- the double value you want to store in the term as the power

3.6.3 Member Function Documentation

3.6.3.1 get_coefficient()

```
double Term::get_coefficient ( ) const
```

This will return the coefficient from the term.

Returns

the coefficient from this term

3.6.3.2 get_power()

```
int Term::get_power ( ) const
```

This will return the power from the term.

Returns

the power from this term

3.6.4 Friends And Related Function Documentation

3.6.4.1 operator<<

```
std::ostream& operator<< (
            std::ostream & out,
            const Term & t )  [friend]
```

This is a friend function. Friends are allowed to access the private data of the object. This will output the object on the given ostream. If the coefficient isn't 0, then it should do the following if the coefficient isn't 1, then the coefficient is shown Show the power if the power is greater than 1 or less then 0 If the power is just one, then just show the x IF the power is 0 then just show the coefficient.

For example

```
1x^1 would just show x
2x^0 would just show 2
1x^2 would just show x^2
1x^0 would just show 1
3x^2 would show 3x^2
```

Parameters

out	- the ostream to display the term on
t	- the Term to display

Returns

return the ostream you were given

Index