**Modified List**

**ADT List Operations**

+isEmpty( ):boolean {query}

determines whether list is empty

+getLength( ):integer {query}

returns number of items in the list

+insert(in newPosition:integer, in newEntry:ListItemType)

{exceptions ListPositionOutOfRangeException, ListException}

inserts newEntry at (list) newPosition if 1 <= newPosition <= getLength( ) + 1

Throws ListPositionOutOfRangeException if newPosition out of range

Throws ListException if list is full

+remove(in position:integer) {exception ListPositionOutOfRangeException}

Deletes item at position if 1 <= position <= getLength( )

Throws ListPositionOutOfRangeException if position out of range

+clear()

Removes all items from list

+getEntry(in position:integer):ListItemType {query}

{exception ListPositionOutOfRangeException}

Returns a copy of the item at position if 1 <= position <= getLength( )

Throws ListPositionOutOfRangeException if position out of range

+setEntry(in position:integer, in newEntry:ListItemType)

{exception ListPositionOutOfRangeException}

Replaces the item at the given position with newEntry

if 1 <= position <= getLength( )

Throws ListPositionOutOfRangeException if position out of range

**ListInterface.h**

// Modified from the following:

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* Interface for the ADT list

Listing 8-1

@file ListInterface.h \*/

#ifndef \_LIST\_INTERFACE

#define \_LIST\_INTERFACE

template<typename ItemType>

class ListInterface

{

public:

/\*\* Sees whether this list is empty.

@return True if the list is empty; otherwise returns false. \*/

virtual bool isEmpty() const = 0;

/\*\* Gets the current number of entries in this list.

@return The integer number of entries currently in the list. \*/

virtual int getLength() const = 0;

/\*\* Inserts an entry into this list at a given position.

@pre 1 <= position <= getLength() + 1

@post newEntry is at the given position in the list,

other entries are renumbered accordingly.

@param newPosition The list position at which to insert newEntry.

@param newEntry The entry to insert into the list.

@throw ListPositionOutOfRangeException if newPosition out of range

@throw ListException if list is full. \*/

virtual void insert(int newPosition, const ItemType& newEntry) = 0;

/\*\* Removes the entry at a given position from this list.

@pre 1 <= position <= getLength()

@post the entry at the given position in the list is removed, other

items are renumbered accordingly.

@param position The list position of the entry to remove.

@throw ListPositionOutOfRangeException if position out of range. \*/

virtual void remove(int position) = 0;

/\*\* Removes all entries from this list.

@post List contains no entries and the count of items is 0. \*/

virtual void clear() = 0;

/\*\* Gets the entry at the given position in this list.

@pre 1 <= position <= getLength().

@post The desired entry has been returned.

@param position The list position of the desired entry.

@return The entry at the given position.

@throw ListPositionOutOfRangeException if position out of range. \*/

virtual ItemType getEntry(int position) const = 0;

/\*\* Replaces the entry at the given position in this list.

@pre 1 <= position <= getLength().

@post The entry at the given position is newEntry.

@param position The list position of the entry to replace.

@param newEntry The replacement entry.

@throw ListPositionOutOfRangeException if position out of range. \*/

virtual void setEntry(int position, const ItemType& newEntry) = 0;

};

#endif

**ListPositionOutOfRangeException.h**

// Modified from Carrano, Data Abstraction & Problem Solving with C++, Fifth Edition,

// Pearson Education, 2007, page 162.

#ifndef \_LIST\_POSITION\_OUT\_OF\_RANGE\_EXCEPTION

#define \_LIST\_POSITION\_OUT\_OF\_RANGE\_EXCEPTION

#include <stdexcept>

#include <string>

using namespace std;

class ListPositionOutOfRangeException : public logic\_error

{

public:

ListPositionOutOfRangeException(const string & message = "")

: logic\_error(message.c\_str())

{ }

};

#endif

**ListException.h**

// Modified from Carrano, Data Abstraction & Problem Solving with C++, Fifth Edition,

// Pearson Education, 2007, pages 162-163.

#ifndef \_LIST\_EXCEPTION

#define \_LIST\_EXCEPTION

#include <stdexcept>

#include <string>

using namespace std;

class ListException : public logic\_error

{

public:

ListException(const string & message = "")

: logic\_error(message.c\_str())

{ }

};

#endif

**ArrayList.h**

// Modified from the following:

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* ADT list: Array-based implementation.

Listing 9-1.

@file ArrayList.h \*/

#ifndef \_ARRAY\_LIST

#define \_ARRAY\_LIST

#include "ListInterface.h"

template<typename ItemType>

class ArrayList : public ListInterface<ItemType>

{

private:

static const int MAX\_LIST = 100;

ItemType items[MAX\_LIST]; // Array of list items

int itemCount; // Current count of list items

public:

ArrayList();

// Copy constructor and destructor are supplied by compiler

bool isEmpty() const;

int getLength() const;

/\*\* @throw ListPositionOutOfRangeException if newPosition out of range

@throw ListException if list is full. \*/

void insert(int newPosition, const ItemType& newEntry);

/\*\* @throw ListPositionOutOfRangeException if newPosition out of range. \*/

void remove(int position);

void clear();

/\*\* @throw ListPositionOutOfRangeException if newPosition out of range. \*/

ItemType getEntry(int position) const;

/\*\* @throw ListPositionOutOfRangeException if newPosition out of range. \*/

void setEntry(int position, const ItemType& newEntry) ;

};

#include "ArrayList.cpp"

#endif

**ArrayList.cpp**

// Modified from the following:

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* Implementation file for the class ArrayList.

@file ArrayList.cpp \*/

#include "ListException.h"

#include "ListPositionOutOfRangeException.h"

template<typename ItemType>

ArrayList<ItemType>::ArrayList() : itemCount(0)

{

}

template<typename ItemType>

bool ArrayList<ItemType>::isEmpty() const

{

return itemCount == 0;

}

template<typename ItemType>

int ArrayList<ItemType>::getLength() const

{

return itemCount;

}

/\*\* @throw ListPositionOutOfRangeException if newPosition out of range

@throw ListException if list is full. \*/

template<typename ItemType>

void ArrayList<ItemType>::insert(int newPosition, const ItemType& newEntry)

{

if (newPosition < 1 || newPosition > itemCount + 1)

throw ListPositionOutOfRangeException(

"ListPositionOutOfRangeException: insert position out of range");

else if (itemCount >= MAX\_LIST)

throw ListException("ListException: list full on insert");

else

{

// Make room for new entry by shifting all entries at

// positions >= newPosition toward the end of the array

// (no shift if newPosition == itemCount + 1)

for (int pos = itemCount; pos >= newPosition; pos--)

items[pos] = items[pos - 1];

// Insert new entry

items[newPosition - 1] = newEntry;

itemCount++;

}

}

/\*\* @throw ListPositionOutOfRangeException if position out of range. \*/

template<typename ItemType>

void ArrayList<ItemType>::remove(int position)

{

if (position < 1 || position > itemCount)

throw ListPositionOutOfRangeException(

"ListPositionOutOfRangeException: remove position out of range");

else

{

// Remove entry by shifting all entries after the one at

// position toward the beginning of the array

// (no shift if position == itemCount)

for (int fromIndex = position, toIndex = fromIndex - 1; fromIndex < itemCount;

fromIndex++, toIndex++)

items[toIndex] = items[fromIndex];

itemCount--;

}

}

template<typename ItemType>

void ArrayList<ItemType>::clear()

{

itemCount = 0;

}

/\*\* @throw ListPositionOutOfRangeException if position out of range. \*/

template<typename ItemType>

ItemType ArrayList<ItemType>::getEntry(int position) const

{

if (position < 1 || position > itemCount)

throw ListPositionOutOfRangeException(

"ListPositionOutOfRangeException: getEntry position out of range");

else

return items[position - 1];

}

/\*\* @throw ListPositionOutOfRangeException if position out of range. \*/

template<typename ItemType>

void ArrayList<ItemType>::setEntry(int position, const ItemType& newEntry)

{

if (position < 1 || position > itemCount)

throw ListPositionOutOfRangeException(

"ListPositionOutOfRangeException: getEntry position out of range");

else

items[position - 1] = newEntry;

}

**Node.h** and **Node.cpp**

See pages 136-137 of the text.

**LinkedList.h**

// Modified from the following:

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* ADT list: Link-based implementation.

Listing 9-2.

@file LinkedList.h \*/

#ifndef \_LINKED\_LIST

#define \_LINKED\_LIST

#include "ListInterface.h"

#include "Node.h"

#include "ListPositionOutOfRangeException.h"

#include "ListException.h"

template<typename ItemType>

class LinkedList : public ListInterface<ItemType>

{

private:

Node<ItemType>\* headPtr; // Pointer to first node in the chain;

// (contains the first entry in the list)

int itemCount; // Current count of list items

// Locates a specified node in this linked list.

// @pre position is the number of the desired node;

// position >= 1 and position <= itemCount.

// @post The node is found and a pointer to it is returned.

// @param position The number of the node to locate.

// @return A pointer to the node at the given position.

Node<ItemType>\* getNodeAt(int position) const;

public:

LinkedList();

LinkedList(const LinkedList<ItemType>& aList);

virtual ~LinkedList();

bool isEmpty() const;

int getLength() const;

/\*\* @throw ListPositionOutOfRangeException if newPosition out of range

@throw ListException if list is full. \*/

void insert(int newPosition, const ItemType& newEntry);

/\*\* @throw ListPositionOutOfRangeException if newPosition out of range. \*/

void remove(int position);

void clear();

/\*\* @throw ListPositionOutOfRangeException if newPosition out of range. \*/

ItemType getEntry(int position) const;

/\*\* @throw ListPositionOutOfRangeException if newPosition out of range. \*/

void setEntry(int position, const ItemType& newEntry) ;

};

#include "LinkedList.cpp"

#endif

**LinkedList.cpp**

// Modified from the following:

// Created by Frank M. Carrano and Tim Henry.

// Copyright (c) 2013 \_\_Pearson Education\_\_. All rights reserved.

/\*\* Implementation file for the class LinkedList.

@file LinkedList.cpp \*/

#include "LinkedList.h" // Header file

template<typename ItemType>

LinkedList<ItemType>::LinkedList() : headPtr(nullptr), itemCount(0)

{

}

template<class ItemType>

LinkedList<ItemType>::LinkedList(const LinkedList<ItemType>& aList) :

itemCount(aList.itemCount)

{

// Points to nodes in original chain

Node<ItemType>\* origChainPtr = aList.headPtr;

if (origChainPtr == nullptr)

headPtr = nullptr; // Original list is empty

else

{

// Copy first node

headPtr = new Node<ItemType>();

headPtr->setItem(origChainPtr->getItem());

// Copy remaining nodes

Node<ItemType>\* newChainPtr = headPtr;

origChainPtr = origChainPtr->getNext();

while (origChainPtr != nullptr)

{

// Get next item from original chain

ItemType nextItem = origChainPtr->getItem();

// Create a new node containing the next item

Node<ItemType>\* newNodePtr = new Node<ItemType>(nextItem);

// Link new node to end of new chain

newChainPtr->setNext(newNodePtr);

// Advance pointer to new last node

newChainPtr = newChainPtr->getNext();

origChainPtr = origChainPtr->getNext();

}

}

}

template<typename ItemType>

LinkedList<ItemType>::~LinkedList()

{

clear();

}

template<typename ItemType>

bool LinkedList<ItemType>::isEmpty() const

{

return itemCount == 0;

}

template<typename ItemType>

int LinkedList<ItemType>::getLength() const

{

return itemCount;

}

/\*\* @throw ListPositionOutOfRangeException if newPosition out of range

@throw ListException if list is full. \*/

template<typename ItemType>

void LinkedList<ItemType>::insert(int newPosition, const ItemType& newEntry)

{

if (newPosition < 1 || newPosition > itemCount + 1)

throw ListPositionOutOfRangeException(

"ListPositionOutOfRangeException: insert position out of range");

else

{

try // try to allocate a new node

{

Node<ItemType>\* newNodePtr = new Node<ItemType>(newEntry);

// Attach new node to chain

if (newPosition == 1)

{

// Insert new node at beginning of chain

newNodePtr->setNext(headPtr);

headPtr = newNodePtr;

}

else

{

// Find node that will be before new node

Node<ItemType>\* prevPtr = getNodeAt(newPosition - 1);

// Insert new node after node to which prevPtr points

newNodePtr->setNext(prevPtr->getNext());

prevPtr->setNext(newNodePtr);

}

itemCount++;

}

catch (bad\_alloc e)

{

throw ListException("ListException: cannot allocate memory on insert");

}

}

}

/\*\* @throw ListPositionOutOfRangeException if position out of range. \*/

template<typename ItemType>

void LinkedList<ItemType>::remove(int position)

{

if (position < 1 || position > itemCount)

throw ListPositionOutOfRangeException(

"ListPositionOutOfRangeException: remove position out of range");

else

{

Node<ItemType>\* curPtr = nullptr;

if (position == 1)

{

// Remove the first node in the chain

curPtr = headPtr; // Save pointer to node

headPtr = headPtr->getNext();

}

else

{

// Find node that is before the one to delete

Node<ItemType>\* prevPtr = getNodeAt(position - 1);

// Point to node to delete

curPtr = prevPtr->getNext();

// Disconnect indicated node from chain by connecting the

// prior node with the one after

prevPtr->setNext(curPtr->getNext());

}

// Return node to system

curPtr->setNext(nullptr);

delete curPtr;

curPtr = nullptr;

itemCount--;

}

}

template<typename ItemType>

void LinkedList<ItemType>::clear()

{

while (!isEmpty())

remove(1);

}

/\*\* @throw ListPositionOutOfRangeException if position out of range. \*/

template<typename ItemType>

ItemType LinkedList<ItemType>::getEntry(int position) const

{

if (position < 1 || position > itemCount)

throw ListPositionOutOfRangeException(

"ListPositionOutOfRangeException: getEntry position out of range");

else

{

Node<ItemType>\* nodePtr = getNodeAt(position);

return nodePtr->getItem();

}

}

/\* @throw ListPositionOutOfRangeException if position out of range. \*/

template<class ItemType>

void LinkedList<ItemType>::setEntry(int position, const ItemType& newEntry)

{

if (position < 1 || position > itemCount)

throw ListPositionOutOfRangeException(

"ListPositionOutOfRangeException: setEntry position out of range");

else

{

Node<ItemType>\* nodePtr = getNodeAt(position);

nodePtr->setItem(newEntry);

}

}

/\*\* @throw ListPositionOutOfRangeException if position out of range. \*/

template<typename ItemType>

Node<ItemType>\* LinkedList<ItemType>::getNodeAt(int position) const

{

if (position < 1 || position > itemCount)

throw ListPositionOutOfRangeException(

"ListPositionOutOfRangeException: remove position out of range");

else

{

// Count from the beginning of the chain

Node<ItemType>\* curPtr = headPtr;

for (int skip = 1; skip < position; skip++)

curPtr = curPtr->getNext();

return curPtr;

}

}