CS302 - Data Structures

using C++

Topic: Sorted Lists

Kostas Alexis

The ADT Sorted List

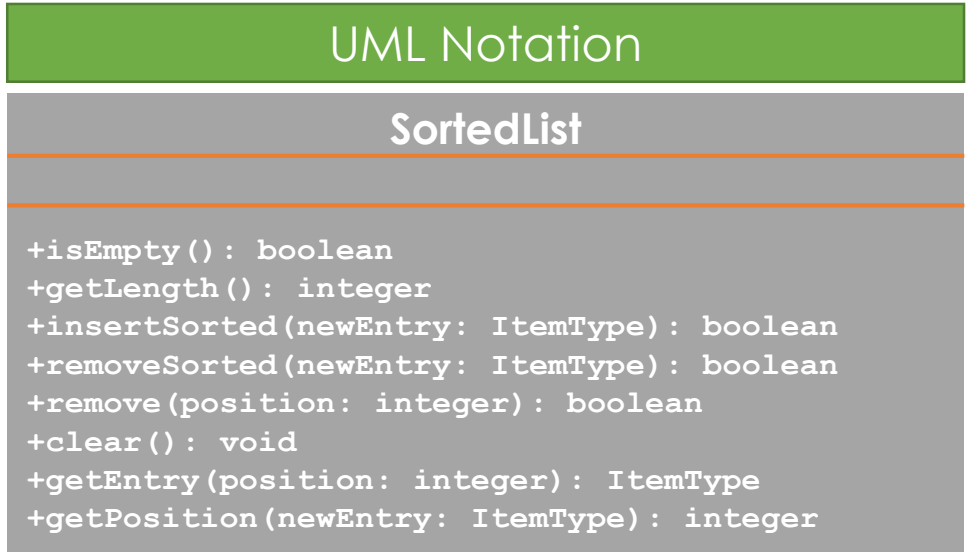
- If your application is using a list in a some phase you want to order its elements in a certain way (e.g., numerically) then you can add a sort operation.
- But when your application only wants sorted data, then a specialized ADT is preferable.
- When you use a sorted list, you only specify the entry you want to add, you never insert it on your own at a cetain position.

Specifying the ADT Sorted List

- ADT Sorted List is a container of items
  - Determines and maintains order of its entries by their values.
- For simplicity, we will allow sorted list to contain duplicate items

Specifying the ADT Sorted List

- UML diagram for the ADT sorted list



Interface Template for the ADT Sorted List

- C++ Interface for sorted lists

```
#ifndef SORTED_LIST_INTERFACE_
#define SORTED_LIST_INTERFACE_

template<class ItemType>
class SortedListInterface
{
public:
      virtual bool insertSorted(const ItemType& newEntry) = 0;
      virtual bool removeSorted(const ItemType& anEntry) = 0;
      virtual int getPosition(const ItemType& anEntry) const = 0;
      virtual bool isEmpty() const = 0;
      virtual int getLength() const = 0;
      virtual bool remove(int position) = 0;
      virtual void clear() = 0;
      virtual ItemType getEntry(int position) const = 0;
      virtual ~SortedListInterface() { }
};
#endif
```

Using the Sorted List Operations

- Abstact Data Type: Sorted List
  - Data: A finite number of objects, not necessarily distinct, having the same data type and ordered by their value.
  - Operations:

Pseudocode	Description
`insertSorted(newEntry)`	Task: Inserts an array into this sorted list in its proper order so that the list remains sorted. Input: newEntry is the new entry. Output: True if insertion is successful, false otherwise.
`removeSorted(anEntry)`	Task: Removes the first or only occurrence of an entry from this sorted list. Input: anEntry is the entry to remove. Output: Returns true if anEntry was located and removed, or false if not. In the latter case, the list remains unchanged.
`getPosition(anEntry)`	Task: Gets the position of the first entry or only occurrence of anEntry in the sorted list. Input: anEntry is the entry to be located. Output: Returns the position of anEntry if it occurs in the sorted list. Otherwise, returns the position where an entry would occur in the list, but as a negative integer.

Using the Sorted List Operations

- ADT sorted list can
  - Add, Remove, Locate an entry
  - Given the entry as an argument
- Operations same as ADT list operations

Using the Sorted List Operations

- ADT sorted list can
  - Add, Remove, Locate an entry
  - Given the entry as an argument
- Operations same as ADT list operations
  - getEntry (by position)
  - Remove (by position)
  - Clear
  - getLength
  - isEmpty

Using the Sorted List Operations

- ADT sorted list can
  - Add, Remove, Locate an entry
  - Given the entry as an argument
- Operations same as ADT list operations
  - getEntry (by position)
  - Remove (by position)
  - Clear
  - getLength
  - isEmpty
- Note: not possible to add or replace entry by position

An example

- We begin by declaring and allocating a sorted list, where we assume that SortedList is an implementation of the operations specified by the interface SortedListInterface.

```
std::unique_ptr<SortedListInterface<std::string>> nameListPtr = std::make_unique<SortedLost<std::string>>();
```

- Next we add names in an arbitrary order, realizing that the ADT will organize them alphabetically.

```
nameListPtr->insertSorted("Jamie");
nameListPtr->insertSorted("Brenda");
nameListPtr->insertSorted("Sarah");
nameListPtr->insertSorted("Tom");
nameListPtr->insertSorted("Carlos");
```

- The sorted list now contains the following entries: Brenda, Carlos, Jamie, Sarah, Tom. Assuming this list, here are some examples of the operations on the sorted list.

```
nameListPtr->getPosition("Jamie"); // returns 3, the position of Jamie in the list.
nameListPtr->getPosition("Jill"); // returns -4, because would belong to position 4 in the list.
nameListPtr->getEntry(2); // returns Carlos, because he is at position 2 I nthe list.
```

- Now remove Tom and the first name in the list.

```
nameList.remove("Tom");
nameList.remove(1);
```

Link-based Implementation

- Option for different ways to implement
  - Array
  - Chain of Linked nodes
  - Instance of a vector
  - Instance of ADT list
- We first consider a chain of linked nodes

Link-based Implementation

- Header file for the class LinkedSortedList

```
#ifndef LINKED_SORTED_LIST_
#define LINKED_SORTED_LIST_
#include <memory>
#include "SortedListInterface.h"
#include "Node.h"
#include "PrecondViolatedExcept.h"

template<class ItemType>
class LinkedSortedList : public SortedListInterface<ItemType>
{
private:
      std::shared_ptr<Node<ItemType>> headPtr;  // Pointer to first node in chain
      int itemCount;                                    // Current count of list items

      // Locates the node that is before the node that should or does contain the given entry
      auto getNodeBefore(const ItemType& anEntry) const;
      // Locate the node at a given position within the chain
      auto getNodeAt(int position) const;
      // Returns a pointer to a copy of the chain to which origChainPtr points
      auto copyChain(const std:: shared_ptr<Node<ItemType>>& origChainPtr);
public:
      LinkedSortedList();
      LinkedSortedList(const LinkedSortedList<ItemType>& aList);
      virtual ~LinkedSortedList();
      bool insertSorted(const ItemType& newEntry);
      bool removeSorted(const ItemType& anEntry);
      int getPosition(const ItemType& const);

      // following methods are the same as given in ListInterface
      bool isEmpty() const;
      int getLength() const;
      bool remove(int position);
      void clear();
      ItemType getEntry(int position) const throw(PrecondViolatedExcept);
};
#include "LinkedSortedList.cpp"
#endif
```

Link-based Implementation

- Header file for the class LinkedSortedList

```
#ifndef LINKED_SORTED_LIST_
#define LINKED_SORTED_LIST_
#include <memory>
#include "SortedListInterface.h"
#include "Node.h"
#include "PrecondViolatedExcept.h"

template<class ItemType>
class LinkedSortedList : public SortedListInterface<ItemType>
{
private:
      std::shared_ptr<Node<ItemType>> headPtr;  // Pointer to first node in chain
      int itemCount;                                    // Current count of list items

      // Locates the node that is before the node that should or does contain the given entry
      auto getNodeBefore(const ItemType& anEntry) const;
      // Locate the node at a given position within the chain
      auto getNodeAt(int position) const;
      // Returns a pointer to a copy of the chain to which origChainPtr points
      auto copyChain(const std:: shared_ptr<Node<ItemType>>& origChainPtr);
public:
      LinkedSortedList();
      LinkedSortedList(const LinkedSortedList<ItemType>& aList);
      virtual ~LinkedSortedList();
      bool insertSorted(const ItemType& newEntry);
      bool removeSorted(const ItemType& anEntry);
      int getPosition(const ItemType& const);

      // following methods are the same as given in ListInterface
      bool isEmpty() const;
      int getLength() const;
      bool remove(int position);
      void clear();
      ItemType getEntry(int position) const throw(PrecondViolatedExcept);
};
#include "LinkedSortedList.cpp"
#endif
```

std::shared_ptr is a smart pointer that retains shared ownership of an object through a pointer. Several shared_ptr objects may own the same object. The object is destroyed and its memory deallocated when either of the following happens:

- the last remaining shared_ptr owning the object is destroyed;
- the last remaining shared_ptr owning the object is assigned another pointer via operator= or reset().

The object is destroyed using delete-expression or a custom deleter that is supplied to shared_ptr during construction.

The Implementation File

- The default constructor and destructor for our class LinkedSortedList have practically the same definitions as they do in the class LinkedList. We emphasize on the remaining methods.

The Implementation File

- Copy Constructor calls private method **copyChain**

```
template<class ItemType>
class LinkedSortedList<ItemType>::
LinkedSortedList(const LinkedSortedList<ItemType>& aList)
{
      headPtr = copyChain(aList.headPtr);
      itemCount = aList.itemCount;
}
```

The Implementation File

- Copy Constructor calls private method **copyChain**

```
template<class ItemType>
class LinkedSortedList<ItemType>::
LinkedSortedList(const LinkedSortedList<ItemType>& aList)
{
      headPtr = copyChain(aList.headPtr);
      itemCount = aList.itemCount;
}
```

copyChain offers a recursive implementation of the constructor.

The Implementation File

- Private method **copyChain**

```
template<class ItemType>
class LinkedSortedList<ItemType>::
copyChain(const std::shared_ptr<Node<ItemType>>& origChainPtr)
{
      std::shared_ptr<Node<ItemType>> copiedChainPtr; // Initial value is nullptr
      if (origChainPtr != nullptr)
      {
            // Build new chain from given one
            // Create new node with the current item
            copiedChainPtr = std::make_shared<Node<ItemType>>(origChainPtr->getItem());
            // Make the node point to the rest of the chain
            copiedChainPtr->setNext(copyChain(origChainPtr->getNext()));
      } // end if
      return copiedChainPtr;
} // end copyChain
```

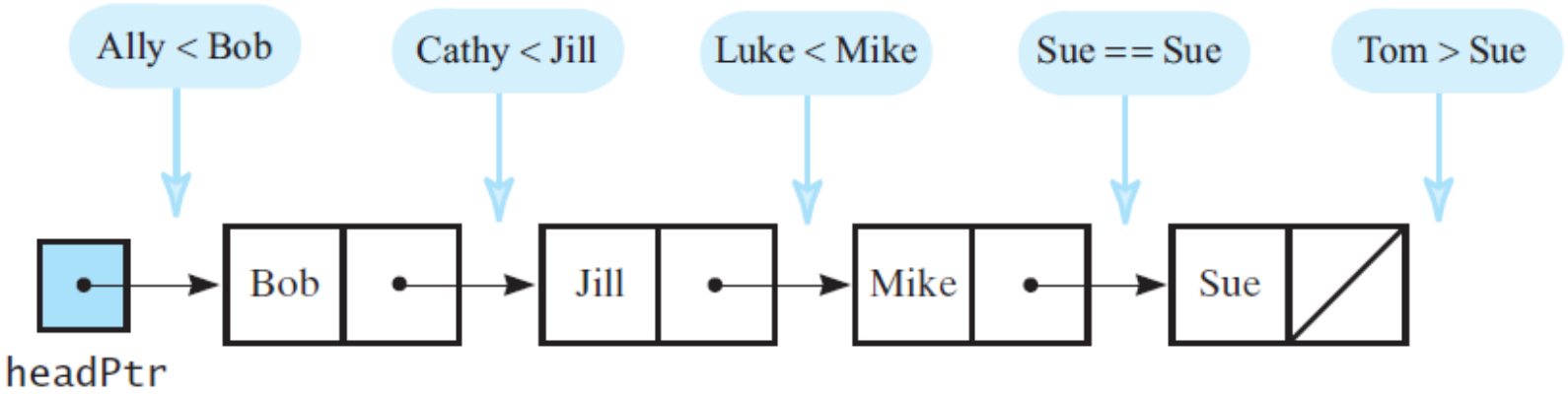
- copyChain begins by testing its pointer argument.
- If it contains nullptr, the original sorted list is empty, so the method returns nullptr.
- Otherwise, the method creates a new node containing the data from the first node of the given chain.
- The method then recursively inserts the new node into the copy of the chain. After each insertion, the method returns a pointer to the new chain.

The Implementation File

- Adding an entry to a sorted list requires that you find where in the list the new entry belongs.
- Since the entries are sorted, you compare the new entry with the entries in the sorted list until you reach an entry that is not smaller than the new enetry.

The Implementation File

- Places to insert strings into a sorted chain of linked nodes



The Implementation File

- What is critical is to have a method that gives the node before the place of the new insertion. Then we know where to insert our new entry.

The Implementation File

- Private method **insertSorted**

```
template<class ItemType>
auto LinkedSortedList<ItemType>::insertSorted(const ItemType& anEntry)
{
      auto newNodePtr(std::make_shared<Node<ItemType>>(newEntry));
      auto prevPtr = getNodeBefore(newEntry);
      if (isEmpty() || (prevPtr == nullptr)) // Add at the beinning
      {
            newNodePtr->setNext(headPtr);
            headPtr = newNodePtr;
      }
      else
      {
            auto aftPtr = prevPtr->getNext();
            newNodePtr->setNext(aftPtr);
            prevPtr->setNext(newNodePtr);
      } // end if
      itemCount++;
      return true;
} // end insertSorted
```

The Implementation File

- Private method **insertSorted**

```
template<class ItemType>
auto LinkedSortedList<ItemType>::insertSorted(const ItemType& anEntry)
{
      auto newNodePtr(std::make_shared<Node<ItemType>>(newEntry));
      auto prevPtr = getNodeBefore(newEntry);
      if (isEmpty() || (prevPtr == nullptr)) // Add at the beinning
      {
            newNodePtr->setNext(headPtr);
            headPtr = newNodePtr;
      }
      else
      {
            auto aftPtr = prevPtr->getNext();
            newNodePtr->setNext(aftPtr);
            prevPtr->setNext(newNodePtr);
      } // end if
      itemCount++;
      return true;
} // end insertSorted
```

make_shared pointer
Allocates and constructs an object of type T passing args to its constructor, and returns an object of type shared_ptr<T> that owns and stores a pointer to it (with a use count of 1).

This function uses ::new to allocate storage for the object. A similar function, allocate_shared, accepts an allocator as argument and uses it to allocate the storage.

The Implementation File

- Private method **insertSorted**

```
template<class ItemType>
auto LinkedSortedList<ItemType>::insertSorted(const ItemType& anEntry)
{
      auto newNodePtr(std::make_shared<Node<ItemType>>(newEntry));
      auto prevPtr = getNodeBefore(newEntry);
      if (isEmpty() || (prevPtr == nullptr)) // Add at the beinning
      {
            newNodePtr->setNext(headPtr);
            headPtr = newNodePtr;
      }
      else
      {
            auto aftPtr = prevPtr->getNext();
            newNodePtr->setNext(aftPtr);
            prevPtr->setNext(newNodePtr);
      } // end if
      itemCount++;
      return true;
} // end insertSorted
```

The Implementation File

- Private method **getNodeBefore**

```
template<class ItemType>
auto LinkedSortedList<ItemType>::
getNodeBefore(const ItemType& anEntry) const
{
      auto curPtr = headPtr;
      std::shared_ptr<Node<ItemType>> prevPtr;
      while ( (curPtr != nullptr) && (anEntry > curPtr->getItem())
      {
            prevPtr = curPtr;
            curPtr = curPtr->getNext();
      } // end while
      return prevPtr;
} // end getNodeBefore
```

Efficiency of Link-based Implementation

- Depends on efficiency of method getNodeBefore
  - Locates insertion point by traversing chain of nodes
- Traversal is O(n)

Implementations that use the ADT List

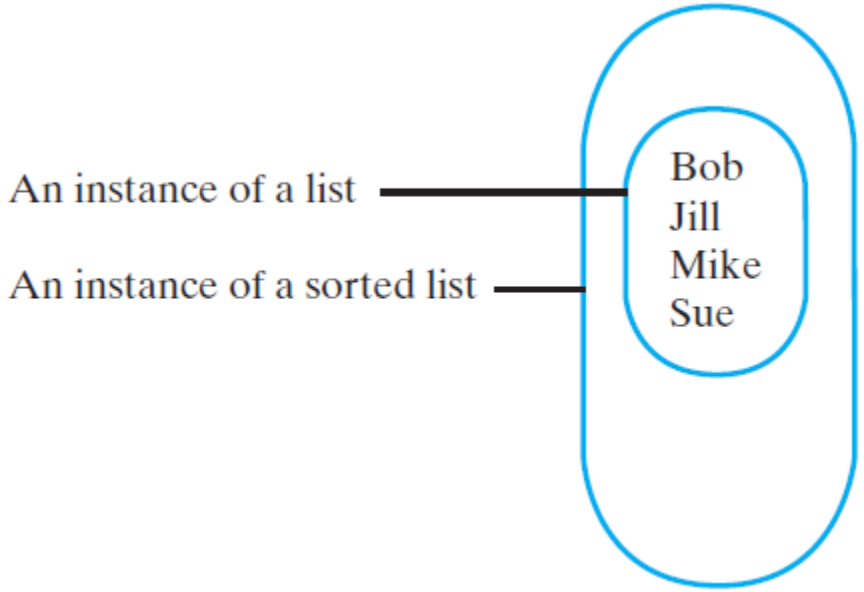
- Avoid duplicates of effort
  - Reuse portions of list's implementation
- Use one of three techniques
  - Containment
  - Public inheritance
  - Private inheritance

Containment

- A sorted list can maintain its entries within a list.
- You use a list as a data field within the class that implements the sorted list.
- This approach uses a type of containment called composition and illustrates the "has-a" relationship between the class of sorted lists and the class of lists.

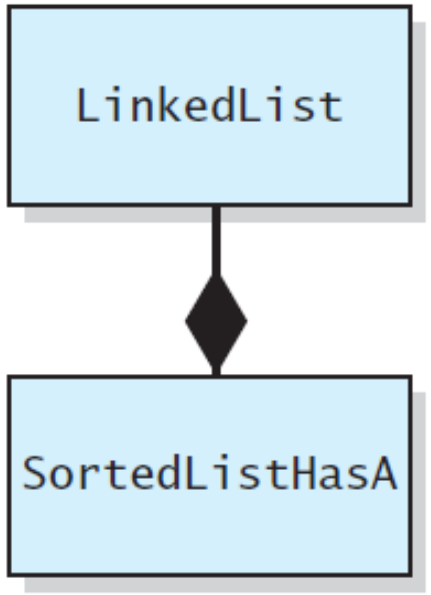
Containment

- An instance of a sorted list that contains a list of its entries



Containment

- SortedListHasA is composed of an instance of the class LinkedList



Containment

- The header file for the class **SortedListHasA**

```
#ifnded SORTED_LIST_HAS_
#define SORTED_LIST_HAS_
#include <memory>
#include “SortedListInterface.h”
#include “ListInterface.h”
#include “Node.h”
#include “PrecondViolatedExcept.h”

template<class ItemType>
class SortedListHasA : public SortedListInterface<ItemType>
{
private:
      std::unique_ptr<ListInterface<ItemType>> listPtr;
public:
      SortedListHasA();
      SortedListHasA(const SortedListHasA<ItemType>& sList);
      virtual ~SortedListHasA();
      bool insertSorted(const ItemType& newEntry);
      bool removeSorted(const ItemType& anEntry);
      int getPosition(const ItemType& newEntry) const;
      // The following methods have the same specifications as given in the ListInterface
      bool isEmpty() const;
      int getLength() const;
      bool remove(int position)
      void clear();
      ItemType getEntry(int position) const throw(PrecondViolatedExcept);
}; // end SortedListHasA
#include “SortedListHasA.cpp”
#endif
```

Containment

- The header file for the class **SortedListH**

```
#ifnded SORTED_LIST_HAS_
#define SORTED_LIST_HAS_
#include <memory>
#include "SortedListInterface.h"
#include "ListInterface.h"
#include "Node.h"
#include "PrecondViolatedExcept.h"

template<class ItemType>
class SortedListHasA : public SortedListInterface<ItemType>
{
private:
      std::unique_ptr<ListInterface<ItemType>> listPtr;
public:
      SortedListHasA();
      SortedListHasA(const SortedListHasA<ItemType>& sList);
      virtual ~SortedListHasA();
      bool insertSorted(const ItemType& newEntry);
      bool removeSorted(const ItemType& anEntry);
      int getPosition(const ItemType& newEntry) const;
      // The following methods have the same specifications as given in the ListInterface
      bool isEmpty() const;
      int getLength() const;
      bool remove(int position)
      void clear();
      ItemType getEntry(int position) const throw(PrecondViolatedExcept);
}; // end SortedListHasA
#include "SortedListHasA.cpp"
#endif
```

unique_ptr pointer
Manages the storage of a pointer, providing a limited garbage-collection facility, with little to no overhead over built-in pointers (depending on the deleter used).

These objects have the ability of taking ownership of a pointer: once they take ownership they manage the pointed object by becoming responsible for its deletion at some point.

unique_ptr objects automatically delete the object they manage (using a deleter) as soon as they themselves are destroyed, or as soon as their value changes either by an assignment operation or by an explicit call to unique_ptr::reset.

unique_ptr objects own their pointer uniquely: no other facility shall take care of deleting the object, and thus no other managed pointer should point to its managed object, since as soon as they have to, unique_ptr objects delete their managed object without taking into account whether other pointers still point to the same object or not, and thus leaving any other pointers that point there as pointing to an invalid location.

A unique_ptr object has two components:
a stored pointer: the pointer to the object it manages. This is set on construction, can be altered by an assignment operation or by calling member reset, and can be individually accessed for reading using members get or release.
a stored deleter: a callable object that takes an argument of the same type as the stored pointer and is called to delete the managed object. It is set on construction, can be altered by an assignment operation, and can be individually accessed using member get_deleter.

unique_ptr objects replicate a limited pointer functionality by providing access to its managed object through operators * and -> (for individual objects), or operator [] (for array objects). For safety reasons, they do not support pointer arithmetics, and only support move assignment (disabling copy assignments).

Containment

- The header file for the class **SortedListHasA**

```
#ifnded SORTED_LIST_HAS_
#define SORTED_LIST_HAS_
#include <memory>
#include "SortedListInterface.h"
#include "ListInterface.h"
#include "Node.h"
#include "PrecondViolatedExcept.h"

template<class ItemType>
class SortedListHasA : public SortedListInterface<ItemType>
{
private:
      std::unique_ptr<ListInterface<ItemType>> listPtr;
public:
      SortedListHasA();
      SortedListHasA(const SortedListHasA<ItemType>& sList);
      virtual ~SortedListHasA();
      bool insertSorted(const ItemType& newEntry);
      bool removeSorted(const ItemType& anEntry);
      int getPosition(const ItemType& newEntry) const;
      // The following methods have the same specifications as given in the ListInterface
      bool isEmpty() const;
      int getLength() const;
      bool remove(int position)
      void clear();
      ItemType getEntry(int position) const throw(PrecondViolatedExcept);
}; // end SortedListHasA
#include "SortedListHasA.cpp"
#endif
```

- listPtr is a unique pointer. This is because we will not define aliases to the list.
- In this way, the sortd list controls ownership of the list.

Containment

- Methods

```
template<class ItemType>
class SortedListHasA<ItemType>::SortedListHasA()
      : listPtr(std::make_unique<LinkedList<ItemType>>())
{
} // end default constructor

template<class ItemType>
class SortedListHasA<ItemType>::
      SortedListHasA(const SortedListHasA<ItemType>& sList)
      : listPtr(std::make_unique<LinkedList<ItemType>>())
{
      for (int position = 1; position <= sList.getLength(); position++)
            listPtr->insert(position, sList.getEntry(position));
} // end copy constructor
```

Containment

- Methods

```
template<class ItemType>
class SortedListHasA<ItemType>::SortedListHasA()
      : listPtr(std::make_unique<LinkedList<ItemType>>())
{
} // end default constructor

template<class ItemType>
class SortedListHasA<ItemType>::
      SortedListHasA(const SortedListHasA<ItemType>& sList)
      : listPtr(std::make_unique<LinkedList<ItemType>>())
{
      for (int position = 1; position <= sList.getLength(); position++)
            listPtr->insert(position, sList.getEntry(position));
} // end copy constructor
```

Default constructor creates an instance of LinkedList and assigns a reference to it to the pointer variable listPtr.

Containment

- Methods

```
template<class ItemType>
class SortedListHasA<ItemType>::SortedListHasA()
      : listPtr(std::make_unique<LinkedList<ItemType>>())
{
} // end default constructor

template<class ItemType>
class SortedListHasA<ItemType>::
      SortedListHasA(const SortedListHasA<ItemType>& sList)
      : listPtr(std::make_unique<LinkedList<ItemType>>())
{
      for (int position = 1; position <= sList.getLength(); position++)
            listPtr->insert(position, sList.getEntry(position));
} // end copy constructor
```

The copy constructor creates a new list and then copies the entries in the given sorted list to the new list.

Containment

- Methods

```
template<class ItemType>
class SortedListHasA<ItemType>::~SortedListHasA()
{
      clear();
} // end destructor

template<class ItemType>
bool SortedListHasA<ItemType>::insertSorted(const ItemType& newEntry)
{
      int newPosition = std::abs(getPosition(newEntry));
      return listPtr->insert(newPosition, newEntry);
} // end insertSorted
```

Containment

- Methods

The destructor must call clear.

```
template<class ItemType>
class SortedListHasA<ItemType>::~SortedListHasA()
{
      clear();
} // end destructor

template<class ItemType>
bool SortedListHasA<ItemType>::insertSorted(const ItemType& newEntry)
{
      int newPosition = std::abs(getPosition(newEntry));
      return listPtr->insert(newPosition, newEntry);
} // end insertSorted
```

Containment

- Methods

```
template<class ItemType>
class SortedListHasA<ItemType>::~SortedListHasA()
{
      clear();
} // end destructor

template<class ItemType>
bool SortedListHasA<ItemType>::insertSorted(const ItemType& newEntry)
{
      int newPosition = std::abs(getPosition(newEntry));
      return listPtr->insert(newPosition, newEntry);
} // end insertSorted
```

To add a new entry we first get its position and then insert it using the methods of the LinkedList.

Containment

- Methods

```
template<class ItemType>
bool SortedListHasA<ItemType>::remove(int position)
{
      return listPtr->remove(position)
} // end remove
```

Containment

- Methods

```
template<class ItemType>
bool SortedListHasA<ItemType>::remove(int position)
{
      return listPtr->remove(position)
} // end remove
```

The remove method exploits the previous relevant implementation on the linked list.

Containment

- Method removeSorted calls getPosition
  - Method returns false if not found
- Other methods
  - isEmpty
  - getLength
  - Remove
  - Clear
  - getEntry

Invoke corresponding list method

Containment

- Worst-case efficiencies of the ADT sorted list operations when implemented using an instance of the ADT List

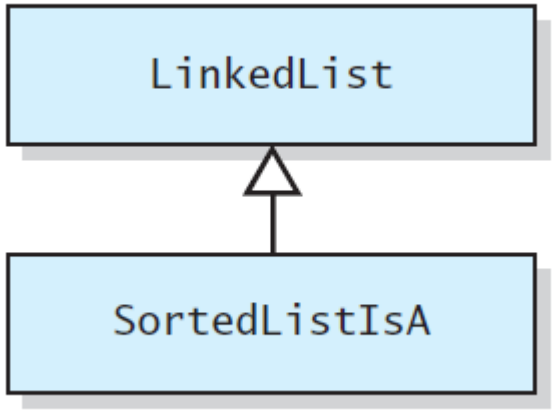
ADT Sorted List Operation	Array-based	Link-based
`insertSorted(newEntry)`	O(n)	$O(n^2)$
`removeSorted(anEntry)`	O(n)	$O(n^2)$
`getPosition(anEntry)`	O(n)	$O(n^2)$
`getEntry(position)`	O(1)	O(n)
`remove(givenPosition)`	O(n)	O(n)
`clear()`	O(1)	O(n)
`getLength()`	O(1)	O(1)
`isEmpty()`	O(1)	O(1)

Public Inheritance

- A list is a container of items you reference by position nuber,
- The sorted list **is-a** list.
- Therefore the sortedList inherits a set of operations available at the LinkedList class.
- Most operations for ADT list are almost the same as
  - Corresponding operations for ADT sorted list
- We use an is-a relationship

Public Inheritance

- **SortedListIsA** as a descendant of **LinkedList**



Public Inheritance

- The class SortedListIsA now has useful operations of the ADT list, such as getEntry, remove, clear, isEmpty and getLength – which it inherits from the class LinkedList – in addition to the methods insertSorted, removeSorted, and getPosition.
- However, it also inherits insert and replace from LinkedList. By using either of these two position-oriented methods, a client could destroy the order of the sorted list.
- To prevent this from happening, SortedListIsA must override them.

Public Inheritance

- A header file for the class **SortedListIsA**

```
#ifndef SORTED_LIST_IS_A_
#define SORTED_LIST_IS_A_
#include <memory>
#include "LinkedList.h"
#include "Node.h"
#include "PrecondViolatedExcept.h"

template<class ItemType>
class SortedListIsA: public LinkedList<ItemType>
{
public:
      SortedListIsA();
      SortedListIsA(const SortedListIsA<ItemType>& sList);
      virtual ~SortedListIsA();
      bool insertSorted(const ItemType& newEntry);
      bool removeSorted(const ItemType& anEntry);
      int getPosition(const ItemType& anEntry) const;

      // The inherited methods remove, clear, getEntry, isEmpty, and getLength have the same specifications as in ListInterface

      // The following methods must be overridden to disable their effect on a sorted list
      bool insert(int newPosition, const ItemType& newEntry) override;
      void replace(int position, const ItemType& newEntry)
            throw(PrecondViolatedExcept) override;
}; // end SortedListIsA
#include "SortedListIsA.cpp"
#endif
```

Public Inheritance

- A header file for the class **SortedListIsA**

```
#ifndef SORTED_LIST_IS_A_
#define SORTED_LIST_IS_A_
#include <memory>
#include "LinkedList.h"
#include "Node.h"
#include "PrecondViolatedExcept.h"

template<class ItemType>
class SortedListIsA: public LinkedList<ItemType>
{
public:
      SortedListIsA();
      SortedListIsA(const SortedListIsA<ItemType>& sList);
      virtual ~SortedListIsA();
      bool insertSorted(const ItemType& newEntry);
      bool removeSorted(const ItemType& anEntry);
      int getPosition(const ItemType& anEntry) const;

      // The inherited methods remove, clear, getEntry, isEmpty, and getLength have the same specifications as in ListInterface

      // The following methods must be overridden to disable their effect on a sorted list
      bool insert(int newPosition, const ItemType& newEntry) override;
      void replace(int position, const ItemType& newEntry)
            throw(PrecondViolatedExcept) override;
}; // end SortedListIsA
#include "SortedListIsA.cpp"
#endif
```

Public Inheritance

- Methods

```
template<class ItemType>
SortedListIsA<ItemType>::SortedListIsA()
{
} // end default constructor

template<class ItemType>
SortedListIsA<ItemType>::SortedListIsA(const SortedListIsA<ItemType>& sList)
      : LinkedList<ItemType>(sList)
{
} // end copy constructor

template<class ItemType>
SortedListIsA<ItemType>::~SortedListIsA()
{
} // end destructor
```

Public Inheritance

- Methods

The default constructor and destructor are straightforward.

```
template<class ItemType>
SortedListIsA<ItemType>::SortedListIsA()
{
} // end default constructor

template<class ItemType>
SortedListIsA<ItemType>::SortedListIsA(const SortedListIsA<ItemType>& sList)
      : LinkedList<ItemType>(sList)
{
} // end copy constructor

template<class ItemType>
SortedListIsA<ItemType>::~SortedListIsA()
{
} // end destructor
```

Public Inheritance

- Methods

The copy constructor invokes LinkdList's copy constructor by using an initializer.

```
template<class ItemType>
SortedListIsA<ItemType>::SortedListIsA()
{
} // end default constructor

template<class ItemType>
SortedListIsA<ItemType>::SortedListIsA(const SortedListIsA<ItemType>& sList)
      : LinkedList<ItemType>(sList)
{
} // end copy constructor

template<class ItemType>
SortedListIsA<ItemType>::~SortedListIsA()
{
} // end destructor
```

Public Inheritance

- Methods

```
template<class ItemType>
bool SortedListIsA<ItemType>::insertSorted(const ItemType& newEntry)
{
      int newPosition = std::abs(getPosition(newEntry));
      // We need to call the LinkedList version of insert, since the SortedListIsA version does nothing but return false
      return LinkedList<ItemType>::insert(newPosition, newEntry);
} // end insertSorted

template<class ItemType>
bool SortedListIsA<ItemType>::removeSorted(const ItemType& anEntry)
{
      int position = getPosition(anEntry);
      bool ableToRemove = position > 0;
      if (ableToRemove)
            ableToRemove = LinkedList<ItemType>::remove(position);
      return ableToRemove;
} // end removeSorted
```

Public Inheritance

- Methods

```
template<class ItemType>
bool SortedListIsA<ItemType>::insertSorted(const ItemType& newEntry)
{
      int newPosition = std::abs(getPosition(newEntry));
      // We need to call the LinkedList version of insert, since the SortedListIsA version does nothing but return false
      return LinkedList<ItemType>::insert(newPosition, newEntry);
} // end insertSorted

template<class ItemType>
bool SortedListIsA<ItemType>::removeSorted(const ItemType& anEntry)
{
      int position = getPosition(anEntry);
      bool ableToRemove = position > 0;
      if (ableToRemove)
            ableToRemove = LinkedList<ItemType>::remove(position);
      return ableToRemove;
} // end removeSorted
```

Public Inheritance

- Methods

```
template<class ItemType>
bool SortedListIsA<ItemType>::insertSorted(const ItemType& newEntry)
{
      int newPosition = std::abs(getPosition(newEntry));
      // We need to call the LinkedList version of insert, since the SortedListIsA version does nothing but return false
      return LinkedList<ItemType>::insert(newPosition, newEntry);
} // end insertSorted

template<class ItemType>
bool SortedListIsA<ItemType>::removeSorted(const ItemType& anEntry)
{
      int position = getPosition(anEntry);
      bool ableToRemove = position > 0;
      if (ableToRemove)
            ableToRemove = LinkedList<ItemType>::remove(position);
      return ableToRemove;
} // end removeSorted
```

insertSorted first calls getPosition to get the intended position of the new entry, ignores the sign of this position, and uses the result and LinkedList's insert method to complete the task.

Public Inheritance

- Methods

removeSorted uses an approach similar to that of insertSorted, but it does not ignore the sign. You cannot remove something it is not there.

```
template<class ItemType>
bool SortedListIsA<ItemType>::insertSorted(const ItemType& newEntry)
{
      int newPosition = std::abs(getPosition(newEntry));
      // We need to call the LinkedList version of insert, since the SortedListIsA version does nothing but return false
      return LinkedList<ItemType>::insert(newPosition, newEntry);
} // end insertSorted

template<class ItemType>
bool SortedListIsA<ItemType>::removeSorted(const ItemType& anEntry)
{
      int position = getPosition(anEntry);
      bool ableToRemove = position > 0;
      if (ableToRemove)
            ableToRemove = LinkedList<ItemType>::remove(position);
      return ableToRemove;
} // end removeSorted
```

Public Inheritance

- Methods

```
template<class ItemType>
bool SortedListIsA<ItemType>::getPosition(const ItemType& anEntry) const
{
      int position = 1;
      int length = LinkedList<ItemType>::getLength();

      while( (position <= length ) && (anEntry > LinkedList<ItemType>::getEntry(position)) )
      {
            position++;
      } // end while

      if ( (position > length) || (anEntry != LinkedList<ItemType>::getEntry(position)) )
      {
            position = -position;
      } // end if

      return position;
} // end getPosition
```

Public Inheritance

- Methods

getPosition uses the list's method getEntry to access each entry in the list sequentially until it either finds a match or reaches the point where the entry being sought would belong if it were in the list.

```
template<class ItemType>
bool SortedListIsA<ItemType>::getPosition(const ItemType& anEntry) const
{
      int position = 1;
      int length = LinkedList<ItemType>::getLength();

      while( (position <= length ) && (anEntry > LinkedList<ItemType>::getEntry(position)) )
      {
            position++;
      } // end while

      if ( (position > length) || (anEntry != LinkedList<ItemType>::getEntry(position)) )
      {
            position = -position;
      } // end if

      return position;
} // end getPosition
```

Public Inheritance

- Method insert overridden to always return false. Prevents insertions into a sorted list by position

```
template<class ItemType>
bool SortedListIsA<ItemType>::insert(int newPosition, const ItemType& newEntry)
{
      return false;
} // end insert
```

Public Inheritance

- Method insert overridden to always return false. Prevents insertions into a sorted list by position

```
template<class ItemType>
bool SortedListIsA<ItemType>::insert(int newPosition, const ItemType& newEntry)
{
      return false;
} // end insert
```

We override insert such that it cannot be utilized. The only way to insert an item must be through insertSored.

Public Inheritance

- Possible that an is-a relationship does not exist
  - In that case do not use public inheritance
- Private inheritance enables use of methods of a base class
  - Without giving client access to them

Public Inheritance

- The header file for the class SortedListAsA

```
#ifndef SORTED_LIST_AS_A_
#define SORTED_LIST_AS_A
#include <memory>
#include "SortedListInterface.h"
#include "ListInterface.h"
#include "Node.h"
#include "PrecondViolatedExcept.h"

template<class ItemType>
class SortedListAsA:  public SortedListInterface<ItemType>,
                      private LinkedList<ItemType>
{
public:
      SortedListAsA();
      SortedListAsA(const SortedListAsA<ItemType>& sList);
      virtual ~SortedListAsA();

      < the rest of the public section is the same as in SortedListHasA >

}; // end SortedListAsA
#include "SortedListAsA.cpp"
#endif
```

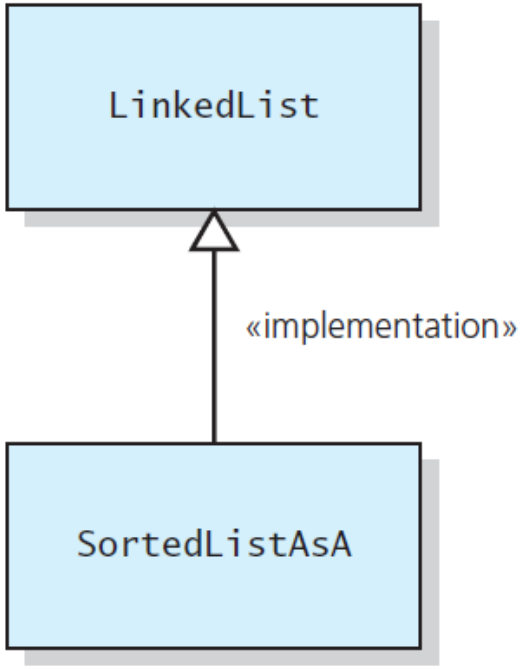
Public Inheritance

- Implementation can use
  - Public methods
  - Protected Methods
- Example

```
template<class ItemType>
ItemType SortedListAsA<ItemType>::getEntry(int position) const throw(PrecondViolatedExcept)
{
      return LinkedList<ItemType>::getEntry(position);
} // end getEntry
```

Public Inheritance

- The SortedListAsA class implemented in terms of the LinkedList class



Thank you