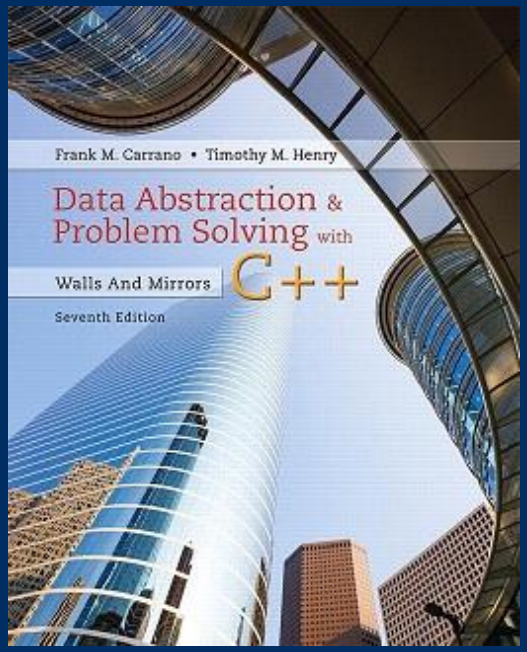
Chapter 12
Sorted Lists and their Implementations

CS 302 - Data Structures

M. Abdullah Canbaz



Modified from authors' slides

Reminders

- Midterm is on **Wednesday March 14th**
  - WRB 2003 at 2:30pm
  - Double sided one page cheat sheet is allowed!

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

```
SortedList
---------------------------------------
---------------------------------------
+isEmpty(): boolean
+getLength(): integer
+insertSorted(newEntry: ItemType): boolean
+removeSorted(newEntry: ItemType): boolean
+remove(position: integer): boolean
+clear(): void
+getEntry(position: integer): ItemType
+getPosition(newEntry: ItemType): integer
```

Interface Template for the ADT Sorted List

```cpp
/** Interface for the ADT sorted list
 @file SortedListInterface.h */

#ifndef SORTED_LIST_INTERFACE_
#define SORTED_LIST_INTERFACE_

template<class ItemType>
class SortedListInterface
{
public:
   /** Inserts an entry into this sorted list in its proper order
       so that the list remains sorted.
    @pre  None.
    @post  newEntry is in the list, and the list is sorted.
    @param newEntry  The entry to insert into the sorted list.
    @return  True if insertion is successful, or false if not. */
   virtual bool insertSorted(const ItemType& newEntry) = 0;

   /** Removes the first or only occurrence of the given entry from this
       sorted list.
```

- A C++ interface for sorted lists

Interface Template for the ADT Sorted List

```
     @pre  None.
     @post  If the removal is successful, the first occurrence of the
         given entry is no longer in the sorted list, and the returned
         value is true. Otherwise, the sorted list is unchanged and the
         returned value is false.
     @param anEntry  The entry to remove.
     @return  True if removal is successful, or false if not. */
    virtual bool removeSorted(const ItemType& anEntry) = 0;

    /** Gets the position of the first or only occurrence of the given
       entry in this sorted list. In case the entry is not in the list,
       determines where it should be if it were added to the list.
     @pre  None.
     @post  The position where the given entry is or belongs is returned.
     The sorted list is unchanged.
     @param anEntry The entry to locate.
     @return  Either the position of the given entry, if it occurs in the
       sorted list, or the position where the entry would occur, but as a
       negative integer. */
    virtual int getPosition(const ItemType& anEntry) const = 0;
```

- A C++ interface for sorted lists

Interface Template for the ADT Sorted List

```
41
42  // The following methods are the same as those given in ListInterface
43  // in Listing 8-1 of Chapter 8 and are completely specified there.
44
45     /** Sees whether this list is empty. */
46     virtual bool isEmpty() const = 0;
47
48     /** Gets the current number of entries in this list. */
49     virtual int getLength() const = 0;
50
51     /** Removes the entry at a given position from this list. */
```

- A C++ interface for sorted lists

Interface Template for the ADT Sorted List

```
    /** Removes the entry at a given position from this list. */
    virtual bool remove(int position) = 0;

    /** Removes all entries from this list. */
    virtual void clear() = 0;

    /** Gets the entry at the given position in this list. */
    virtual ItemType getEntry(int position) const = 0;

    /** Destroys this sorted list and frees its assigned memory.
    virtual ~SortedListInterface() { }

};  // end SortedListInterface
#endif
```

- A C++ interface for sorted lists

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

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

The Header File

- The header file for the class LinkedSortedList

```
/** ADT sorted list: Link-based implementation.
 @file LinkedSortedList.h */

#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.
   // @param anEntry  The entry to find.
   // @return  Either a pointer to the node before the node that contains
   //  or should contain the given entry, or nullptr if no prior node exists.
```

The Header File

- The header file for the class LinkedSortedList

```
    auto getNodeBefore(const ItemType& anEntry) const;

    // Locates 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& newEntry) const;

    // The 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);
}; // end LinkedSortedList
#include "LinkedSortedList.cpp"
#endif
```

The Implementation File

- Copy constructor calls private method copyChain

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

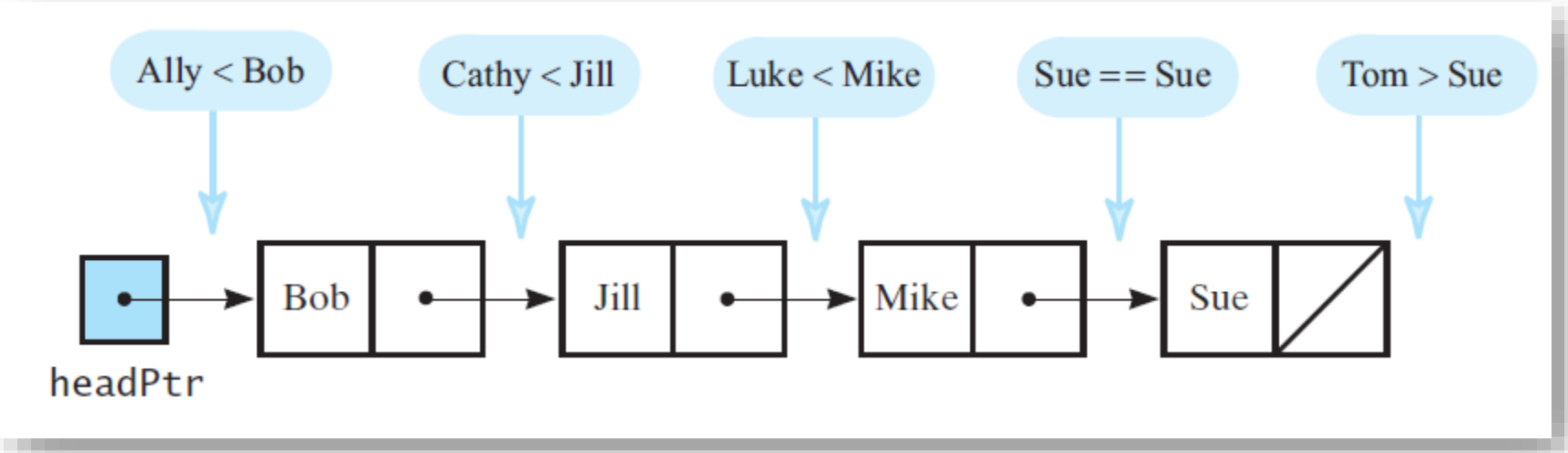
The Implementation File

- Private method copyChain

```
template<class ItemType>
auto 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
```

The Implementation File



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

The Implementation File

```
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
```

- Private method getNodeBefore

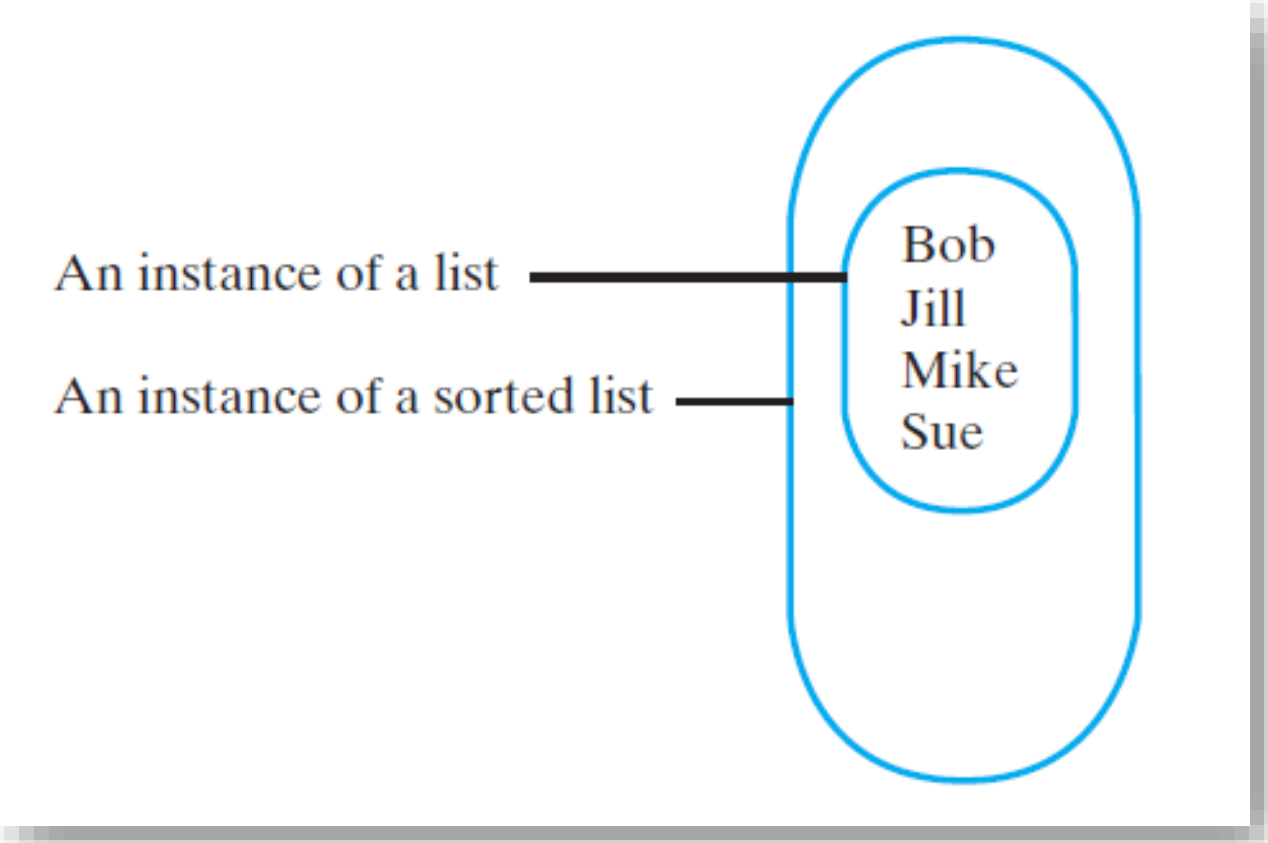
Efficiency of the Link-Based Implementation

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

Implementations That Use the ADT List

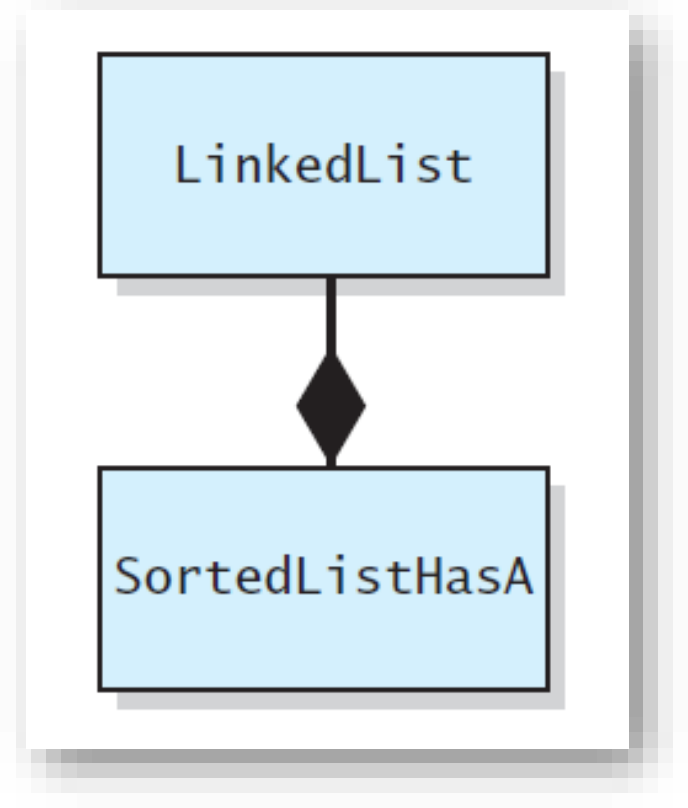
- Avoid duplication of effort
  - Reuse portions of list's implementation
- Use one of three techniques
  - Containment
  - Public inheritance
  - Private inheritance

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

```
/** ADT sorted list using the ADT list.
 @file SortedListHasA.h */
#ifndef SORTED_LIST_HAS_A_
#define SORTED_LIST_HAS_A_
#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;

```

Containment

- The header file for the class SortedListHasA

```
17  public:
18      SortedListHasA();
19      SortedListHasA(const SortedListHasA<ItemType>& sList);
20      virtual ~SortedListHasA();
21
22      bool insertSorted(const ItemType& newEntry);
23      bool removeSorted(const ItemType& anEntry);
24      int getPosition(const ItemType& newEntry) const;
25
26      // The following methods have the same specifications
27      // as given in ListInterface in Chapter 8:
28      bool isEmpty() const;
29      int getLength() const;
30      bool remove(int position);
31      void clear();
32      ItemType getEntry(int position) const throw(PrecondViolatedExcept);
33  }; // end SortedListHasA
34  #include "SortedListHasA.cpp"
35  #endif
```

Containment

- Methods

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

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

Containment

- Method removeSorted calls getPosition
  - Method returns false if not found
- Other methods
  - isEmpty
  - getLength
  - remove
  - clear
  - getEntry

Invoke corresponding list method

Containment

ADT Sorted List Operation	List Implementation	
	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(), isEmpty()`	$O(1)$	$O(1)$

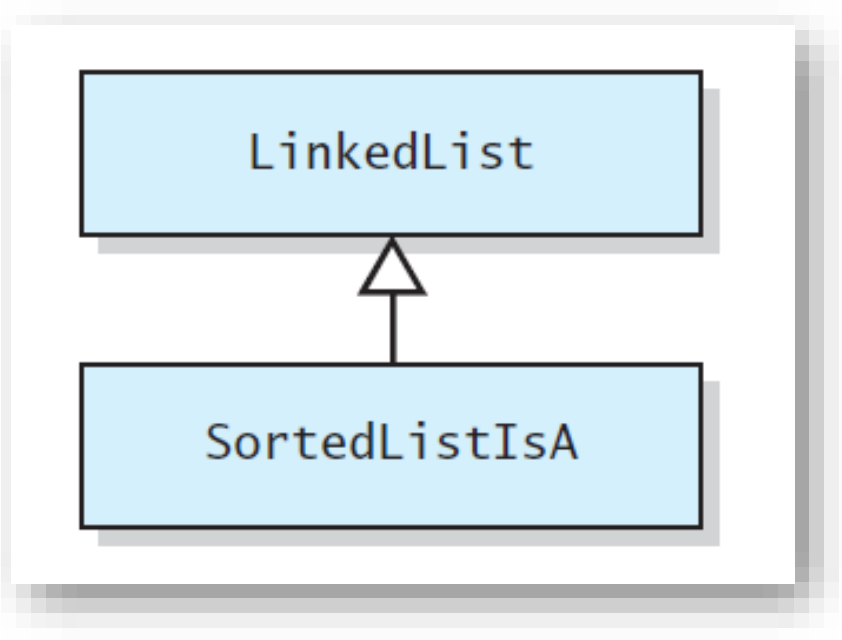
- The worst-case efficiencies of the ADT sorted list operations when implemented using an instance of the ADT list

Public Inheritance

- 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

- A header file for the class SortedListIsA

```
/** ADT sorted list using ADT list.
 @file SortedListIsA.h */
#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();
```

Public Inheritance

- A header file for the class 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 given 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

```
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
```

- Methods

Public Inheritance

- Method insertSorted

```
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
```

Public Inheritance

- Method removeSorted

```
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

- Method getPosition

```
template<class ItemType>
int 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

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

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

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

```
/** ADT sorted list using ADT list.
 @file SortedListAsA.h */
#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 in listing 12-3


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

- The header file for the class SortedListAsA

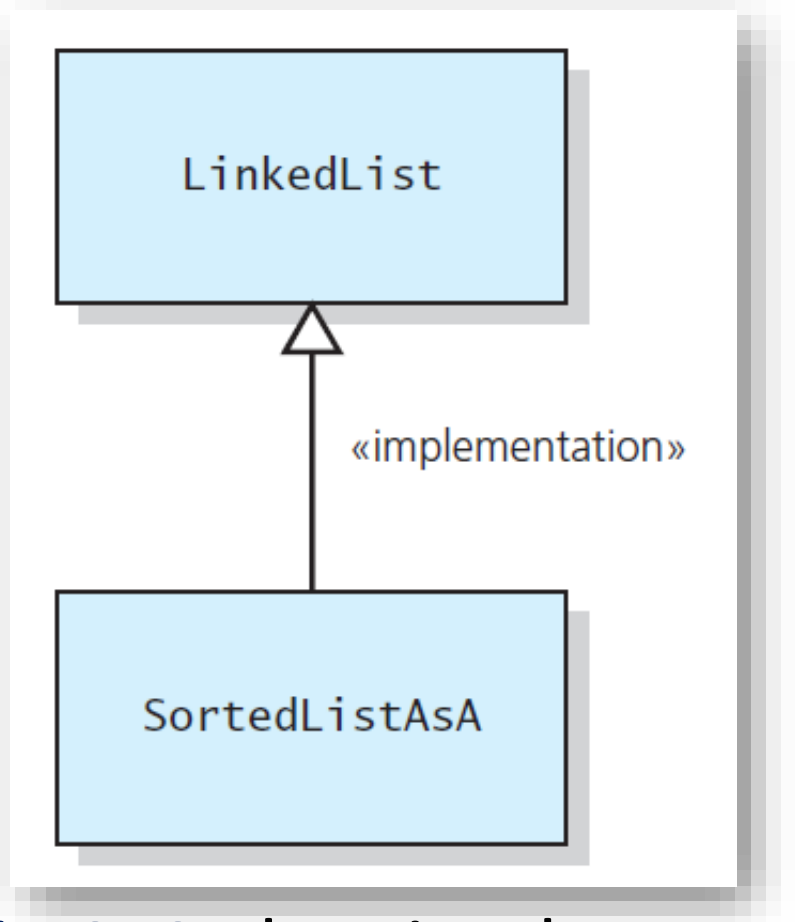
Public Inheritance

- 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
```

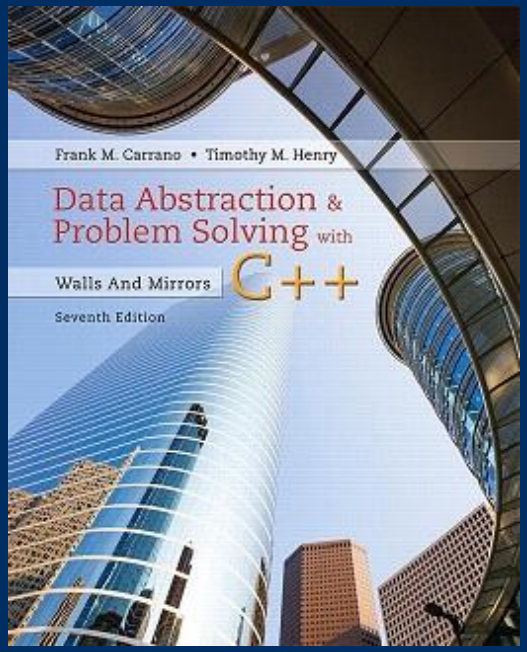
Public Inheritance



- The SortedListAsA class implemented in terms of the LinkedList class

Chapter 12
Sorted Lists and their Implementations

The End



Modified from authors' slides

Midterm Review

- C++ review
- Recursion
- Lists
- Stack
- Algorithm Efficiency
- Sorting
- Bonus