Sorted List Implementation

CS110C

Max Luttrell, CCSF

sorted list ADT implementation

- We already have implemented a list, which is quite similar to a sorted list. There are several ways we can implement a sorted list.

  - start from scratch

  - start with our list code base, rip out methods we no longer want (insert at position, etc.). add new methods we want (insertSorted, etc.)

  - containment

  - public inheritance

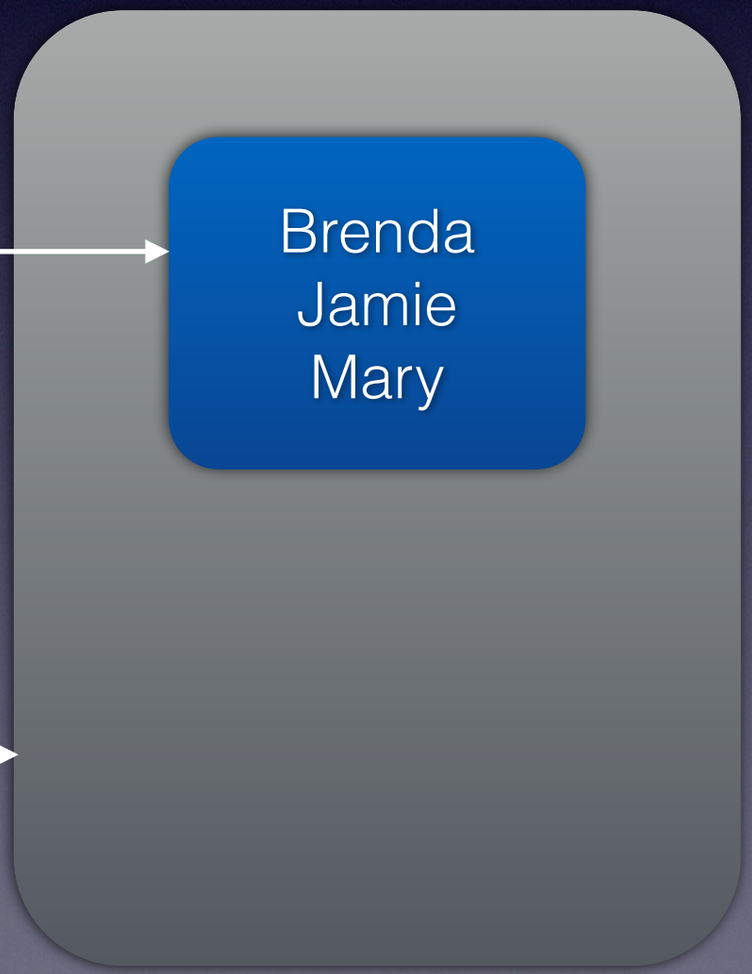
  - private inheritance

containment - "has a"

- strategy: our sorted list will have a list inside.

An instance of a List

An instance of
a Sorted List



containment

```
template<class ItemType>
class SortedListHasA : public SortedListInterface<ItemType>
{
private:
   ListInterface<ItemType>* listPtr;

public:
   SortedListHasA();
   SortedListHasA(const SortedListHasA<ItemType>& sList);
   virtual ~SortedListHasA();

   // The following methods are new for sortedList:
   void insertSorted(const ItemType& newEntry);
   bool removeSorted(const ItemType& anEntry);
   int getPosition(const ItemType& anEntry) const;

   // The following methods have the same specifications
   // as given in ListInterface:
   bool isEmpty() const;
   int getLength() const;
   bool remove(int position);
   void clear();
   ItemType getEntry(int position) const throw(PrecondViolatedExcep);
}; // end SortedListHasA
```

containment - insertSorted

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

containment - efficiency

- The efficiency of the new methods depend on `getPosition`. This will depend on the underlying List implementation (link based or array based).

```
// determine the position of anEntry in SortedList list
// assume that anEntry is in the list
getPosition(anEntry: ItemType, list: SortedList) : integer
for (pos in 1..list.getLength())
  if (list.getEntry(pos)==anEntry)
    return pos
```

efficiency

- getPosition() calls getEntry() many times (worst case: n times, assuming we use a linear search).
- if our underlying list is link-based, getEntry() can require traversing the entire list, and is thus itself an O(n) operation.
- for a link-based implementation, getPosition() is thus a $O(n^2)$ operation!

```
// determine the position of anEntry in SortedList list
// assume that anEntry is in the list
getPosition(anEntry: ItemType, list: SortedList) : Integer
for (pos in 1..list.getLength())
  if (list.getEntry(pos)==anEntry)
    return pos
```

inheritance

- Public inheritance: if we derive SortedList from LinkedList using **public** inheritance, we get all the functions that LinkedList supports, including the ones we don't want (insert at position, etc.)

- Private inheritance: if we derive SortedList from LinkedList using **private** inheritance, all of the public methods in LinkedList are hidden from the user. We can use them to implement SortedList.

smart pointers

- a **smart pointer** is like a regular pointer, but automatically deallocates memory when the pointer goes out of scope

- we will use regular pointers in our examples and homework but the 7th edition textbook uses smart pointers. more info in 7th edition C++ Interlude 4

```
template<class ItemType>
class SortedListHasA : public SortedListInterface<ItemType>
{
private:
   ListInterface<ItemType>* listPtr;
…
}
```

```
template<class ItemType>
class SortedListHasA : public SortedListInterface<ItemType>
{
private:
    std::unique_ptr<ListInterface<ItemType>> listPtr;
…
}
```