The ADT SortedList

THE ADT SORTED LIST

- **ADT determines where entries belong**

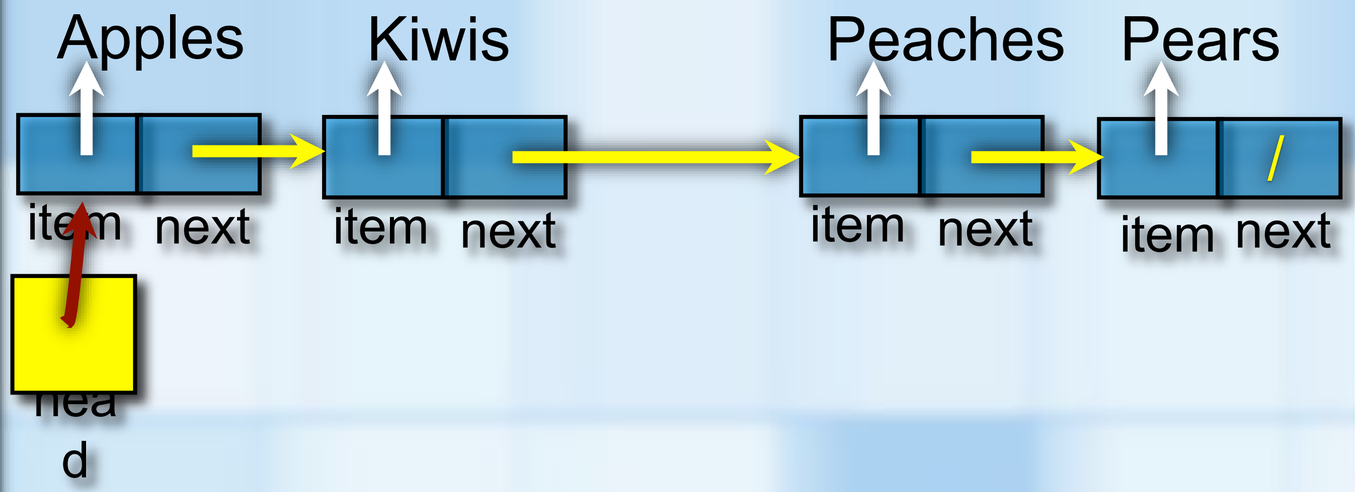

```
template<class ItemType>
class SortedListInterface
{
public:

  virtual void insertSorted(const ItemType& newEntry) = 0;

  virtual bool removeSorted(const ItemType& anEntry) = 0;

  virtual int getPosition(const ItemType& anEntry) = 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;

}; // end SortedListInterface
```

SortedListInterface.h

Pearson

A Linked Sorted List

- Only uses a reference to the first node




```
template<class ItemType>
LinkedSortedList<ItemType>::LinkedSortedList()
                          : headPtr(nullptr), itemCount(0)
{
}  // end default constructor
```

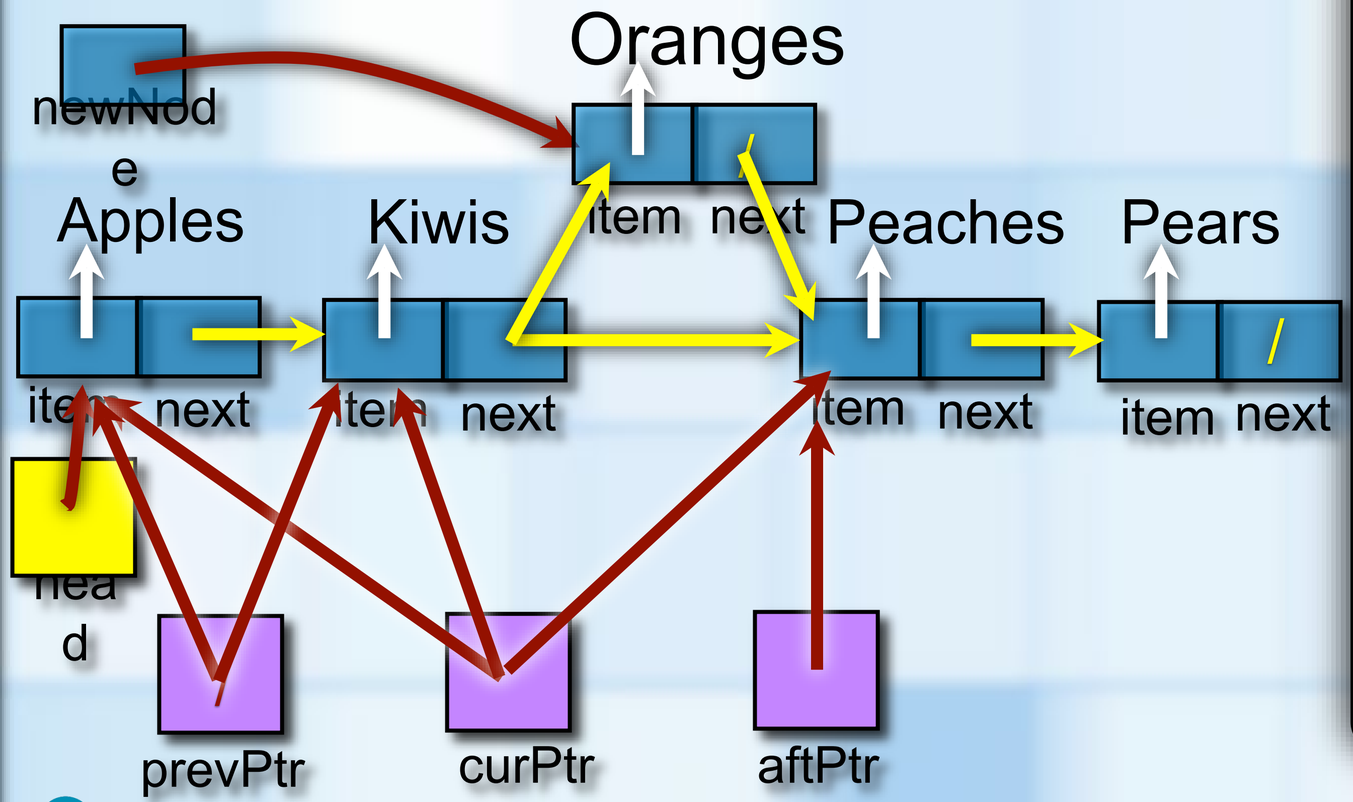
LinkedSortedList.cpp

Pearson

A Linked Sorted List

- **Only uses a reference to the first node**
- **Adding a node to the sorted list**
  - Finding the node before an entry

```
grocerySortedList.insertSorted("Oranges");
```




```
template<class ItemType>
Node<ItemType>* LinkedSortedList<ItemType>::
    getNodeBefore(const ItemType& anEntry) const
{
  Node<ItemType>* curPtr = headPtr;
  Node<ItemType>* prevPtr = nullptr;

  while ( (curPtr != nullptr)
          && (anEntry > curPtr->getItem()) )
  {
    prevPtr = curPtr;
    curPtr = curPtr->getNext();
  } // end while
  return prevPtr;
} // end getNodeBefore
```

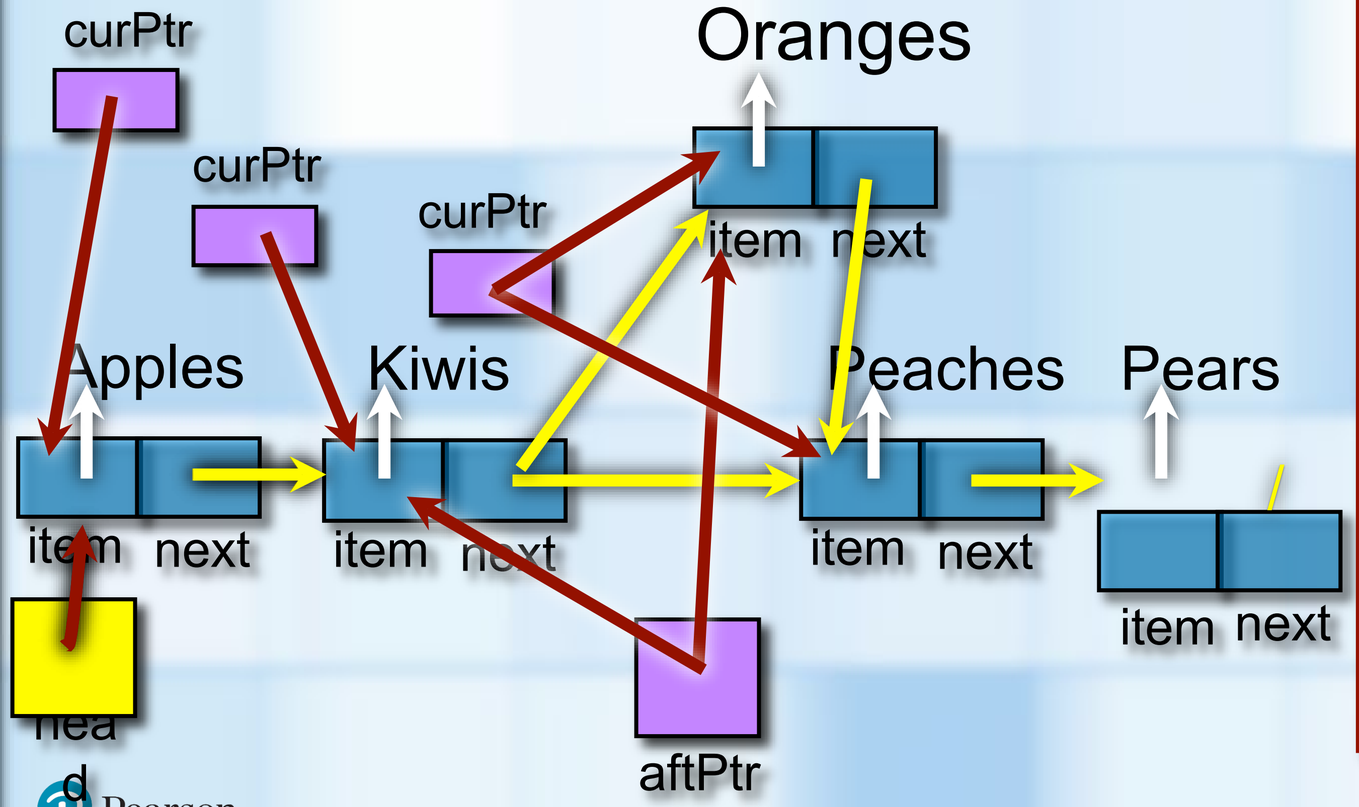
LinkedSortedList.cpp

Pearson

A Linked Sorted List

- Only uses a reference to the first node
- Adding a node to the sorted list
  - Recursive implementation

```
grocerySortedList.add("Oranges");
```




```
template<class ItemType>
void LinkedSortedList<ItemType>::
          insertSorted(const ItemType& newEntry)
{
  headPtr = insertSorted(newEntry, headPtr);
  itemCount++;
}

template<class ItemType>
void LinkedSortedList<ItemType>::
          insertSorted(const ItemType& newEntry,
                       Node<ItemType>* curPtr)
{
  if (currentNode == nullptr ||
          newEntry < curPtr->getItem())
  {
     curPtr = new Node<ItemType>(newEntry, curPtr);
  }
  else
  {
    Node<ItemType>* aftPtr =
        insertSorted(newEntry, curPtr->getNext());
    curPtr->setNext(aftPtr);
  } // end if

  return curPtr;
} // end insertSorted
```

Pearson

- ADT SortedList Implementations

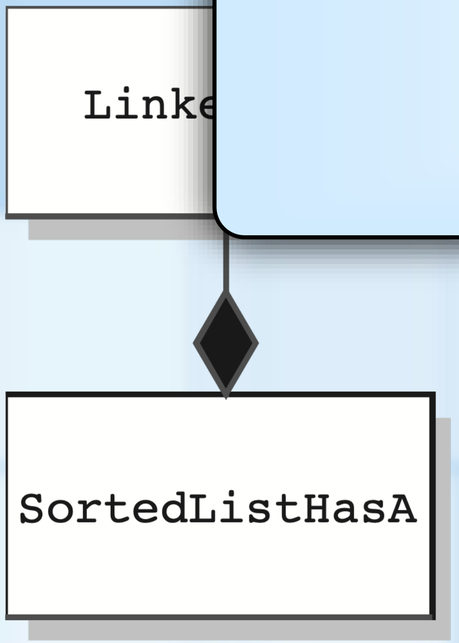
Test Subtitle

Pearson

LIST-BASED SORTED LIST

- **Reusing Previous Work**
  - Composition
    - **`Has-A`** Relationship
    - Our **`SortedLis`** data field

`SortedListHasA` is a wrapper or adapter for **`LinkedList`**



`SortedListHasA.h`

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

public:
                                                    ls
```

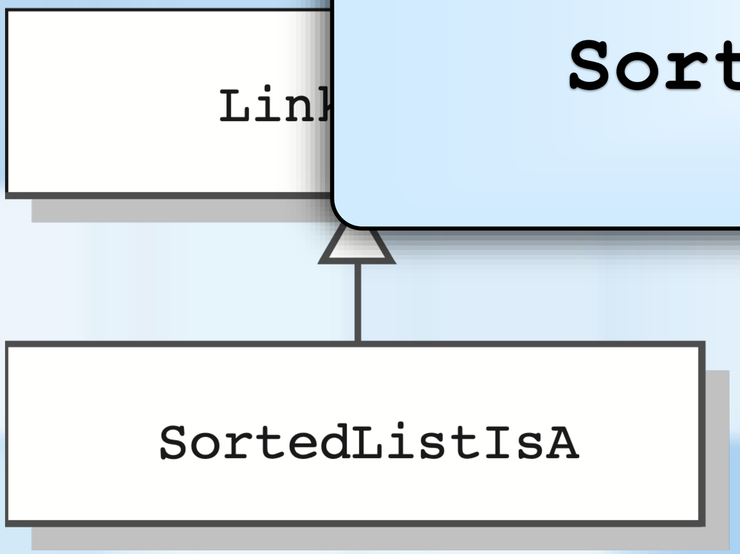
`SortedListHasA.cpp`

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

Pearson

LIST-BASED SORTED LIST

- **Reusing Previous Work**
  - Public Inheritance
    - **`Is-A`** Relations
    - Our **`SortedLi`** **`LinkedList`**



Must override **`LinkedList`** **`insert`** and **`setEntry`** to protect integrity of **`SortedList`**'s sorted entry order.

`SortedListIsA.h`

```
template<class ItemType>
class SortedListIsA : public LinkedList<ItemType>
{

public:
   // Constructors, destructor and SortedListInterface methods
```

```
template<class ItemType>
void SortedListIsA<ItemType>::insertSorted(const ItemType& newEntry)
{
  int newPosition = fabs(getPosition(newEntry));
  LinkedList<ItemType>::insert(newPosition, newEntry);

} // end insertSorted
```

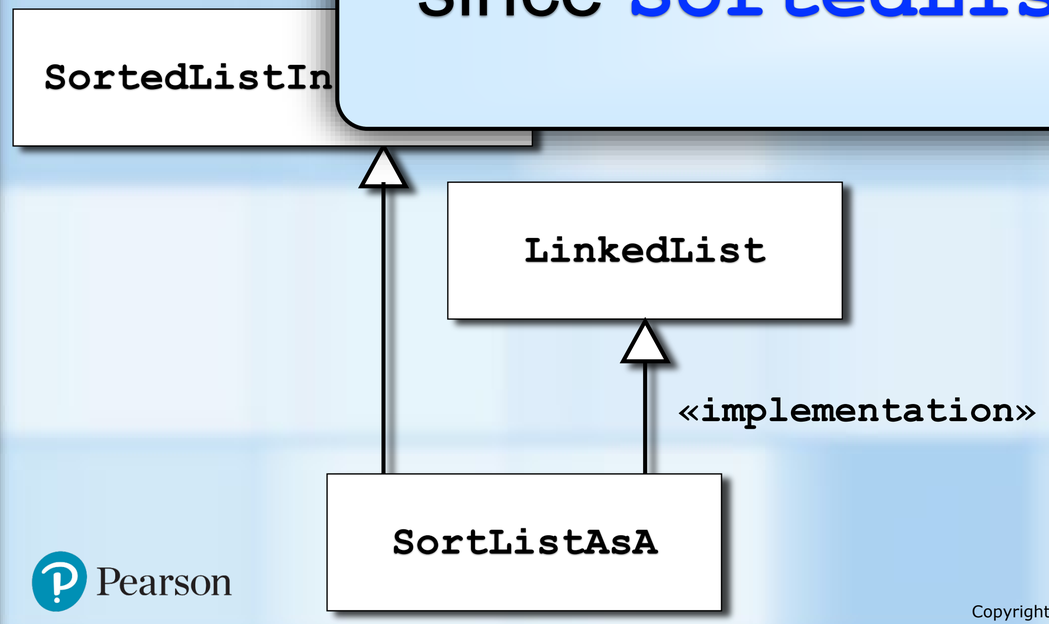
`SortedListIsA.cpp`

Pearson

LIST-BASED SORTED LIST

- **Reusing Previous Work**
  - Private Inheritance
    - **As-A** Relationship
    - The **Lin**
      clients c

SortedListAsA only receives code from **LinkedList** in this example of multiple inheritance since **SortedListInterface** is an abstract base class.



SortedListIsA.h

```
template<class ItemType>
class SortedListAsA : public SortedListInterface<ItemType>,
                      private LinkedList<ItemType>
{

public:
   // Constructors, destructor and SortedListInterface methods
```

SortedListIsA.cpp

```
template<class ItemType>
void SortedListAsA<ItemType>::insertSorted(const ItemType& newEntry)
{
  int newPosition = fabs(getPosition(newEntry));
  LinkedList<ItemType>::insert(newPosition, newEntry);

}  // end insertSorted
```

Pearson