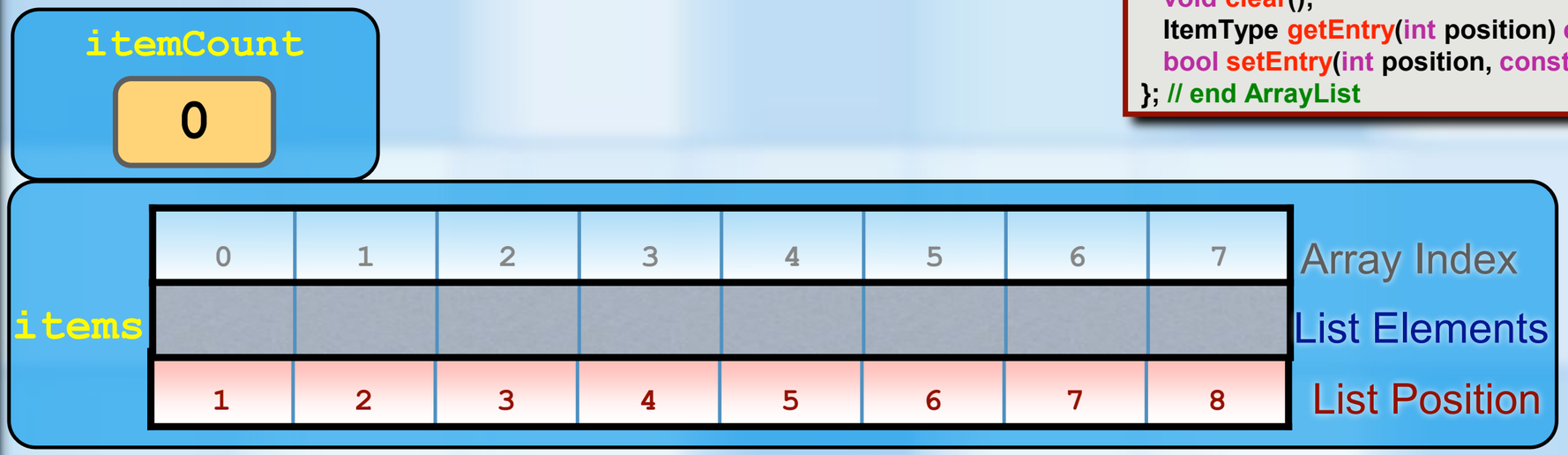
Overview of ArrayList Class

THE CLASS ARRAYLIST

- Data Fields
- Constructor

```
ListInterface<string>* groceryList =
                    new ArrayList<string>();
```

```
template<class ItemType>
class ArrayList : public ListInterface<ItemType>
{
private:
  static const int DEFAULT_CAPACITY = 8;
  ItemType items[DEFAULT_CAPACITY];
  int itemCount;
  int maxItems;

public:
  ArrayList();
  bool isEmpty() const;
  int getLength() const;
  bool insert(int position, const ItemType& newEntry);
  bool remove(int position);
  void clear();
  ItemType getEntry(int position) const;
  bool setEntry(int position, const ItemType& newEntry);
}; // end ArrayList
```



THE CLASS ARRAYLIST

- Data Fields
- Constructors
- Inserting items to the list

```
groceryList->insert(1, "Apples");
groceryList->insert(2, "Oranges");
groceryList->insert(3, "Pears");
groceryList->insert(4, "Tomatoes");
groceryList->insert(5, "Nachos");
```

```
template<class ItemType>
bool ArrayList<ItemType>::insert(int newPosition,
                                 const ItemType& newEntry)
{
  bool ableToInsert = (newPosition >= 1) &&
                      (newPosition <= itemCount + 1) &&
                      (itemCount < maxItems);
  if (ableToInsert)
  {
    // 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 index = itemCount; index >= newPosition; index--)
      items[index] = items[index - 1];

    // Insert new entry
    items[newPosition - 1] = newEntry;
    itemCount++;  // Increase count of entries
  }  // end if

  return ableToInsert;
}  // end insert
```

itemCount: 5

items

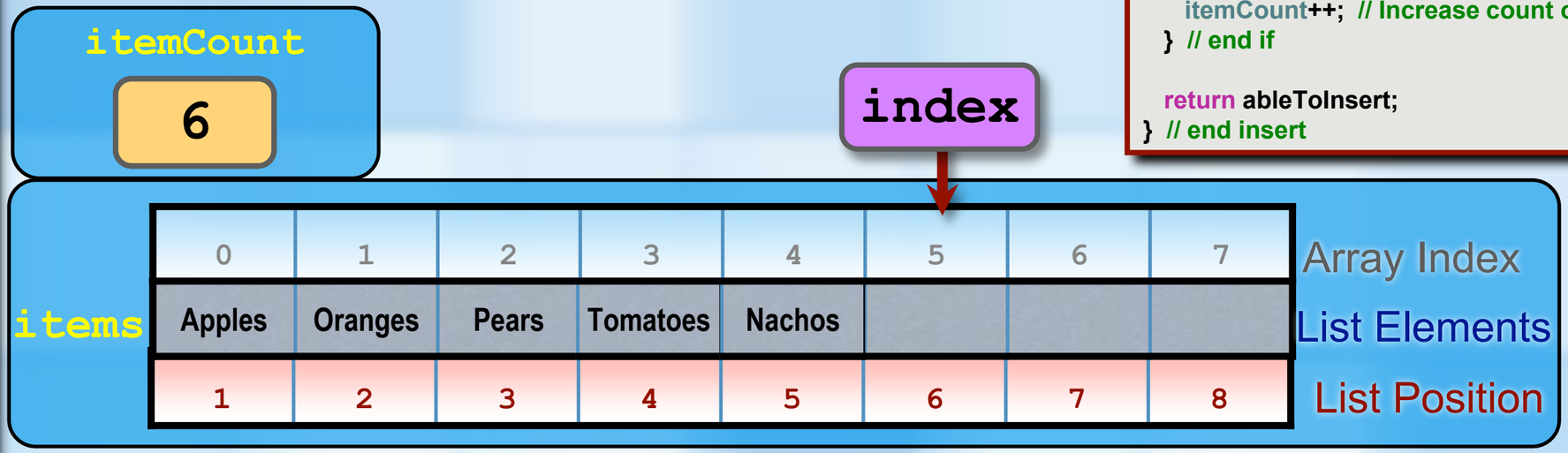
0	1	2	3	4	5	6	7	Array Index
Apples	Oranges	Pears	Tomatoes	Nachos				List Elements
1	2	3	4	5	6	7	8	List Position

THE CLASS ARRAYLIST

- Data Fields
- Constructors
- Inserting items to the list

```
groceryList->insert(4,"Cheese");
```

```
template<class ItemType>
bool ArrayList<ItemType>::insert(int newPosition,
                                 const ItemType& newEntry)
{
  bool ableToInsert = (newPosition >= 1) &&
                      (newPosition <= itemCount + 1) &&
                      (itemCount < maxItems);
  if (ableToInsert)
  {
    // 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 index = itemCount; index >= newPosition; index--)
      items[index] = items[index - 1];

    // Insert new entry
    items[newPosition - 1] = newEntry;
    itemCount++;  // Increase count of entries
  }  // end if

  return ableToInsert;
}  // end insert
```

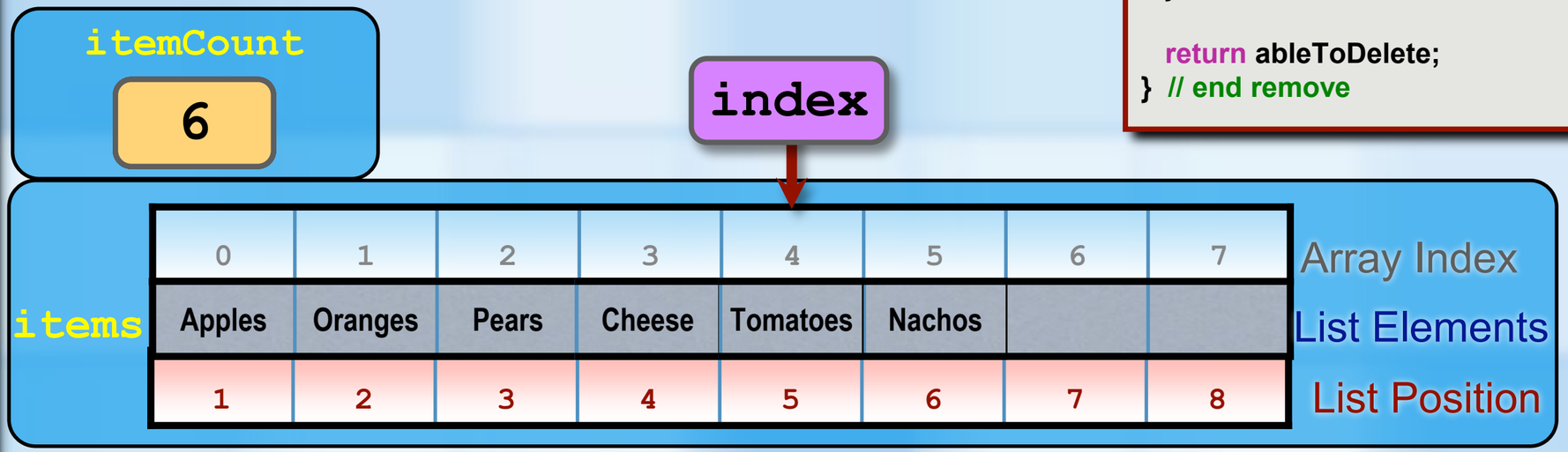


THE CLASS ARRAYLIST

- Data Fields
- Constructors
- Inserting items to the list
- Removing an item

```
groceryList->remove(4);
```

```
template<class ItemType>
bool ArrayList<ItemType>::remove(int position)
{
  bool ableToDelete = (position >= 1) &&
                                          (position <= itemCount);
  if (ableToDelete)
  {
    // Delete entry by shifting all entries after the one at
    // position toward the beginning of the array
    // (no shift if position == itemCount)
    for (int index = position; index < itemCount; index++)
      items[index - 1] = items[index];

    itemCount--;  // Decrease count of entries
  }  // end if

  return ableToDelete;
}  // end remove
```

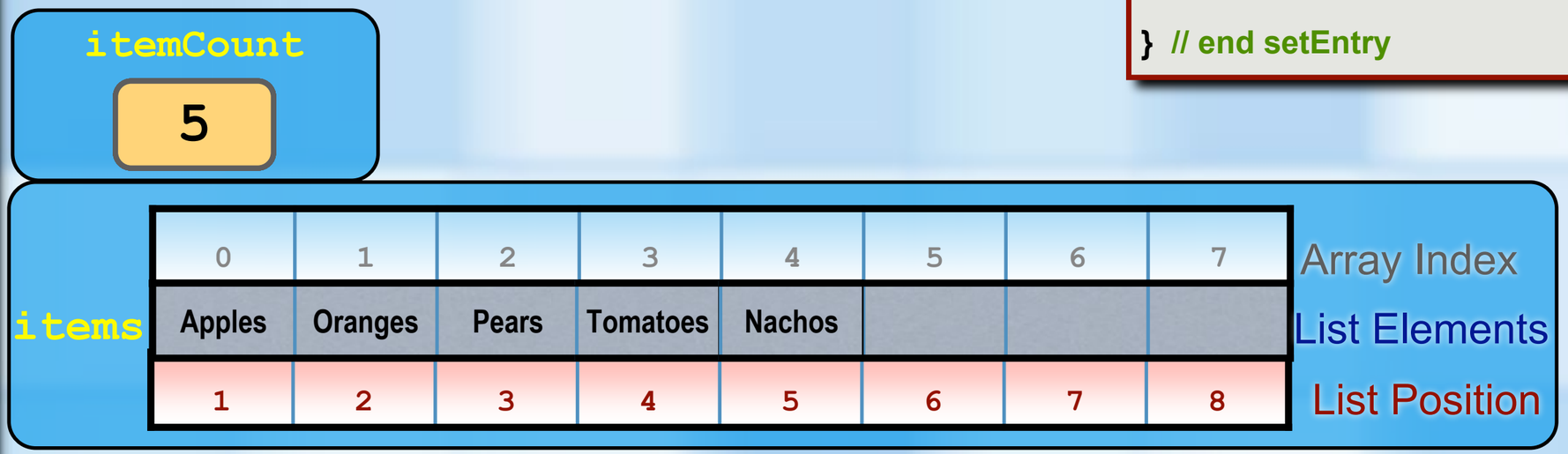


THE CLASS ARRAYLIST

- **Replacing an entry**

```
groceryList->setEntry(4,"Cheese");
```

```
template<class ItemType>
void ArrayList<ItemType>::setEntry(int position,
                                   const ItemType& newEntry)
{
  // Enforce precondition
  bool ableToSet = (position >= 1) &&
                   (position <= itemCount);

  if (ableToSet)
    items[position - 1] = newEntry;

  return ableToSet;

}  // end setEntry
```



THE CLASS ARRAYLIST

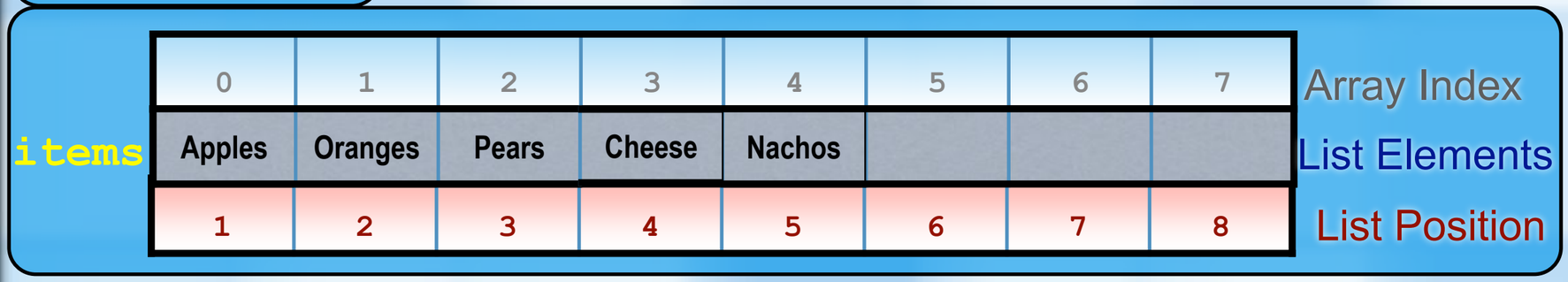
- Replacing an entry
- Getting an entry at a specific position

```
groceryList->getEntry(4);
```

return to client

```
template<class ItemType>
ItemType ArrayList<ItemType>::getEntry(int position) const
{
  assert((position >= 1) && (position <= itemCount));
  return items[position - 1];
}  // end getEntry
```

itemCount: 5



THE CLASS ARRAYLIST

- Replacing a
- Getting an
  specific po

```
groceryList->getEn
```

Client Code

```
ArrayList< std:: string> groceryList =
                                    new ArrayList< std::string>();

// statements that add items to the bag .....
std::string someItem;
try
{
  someItem = groceryList->getEntry(4);
}
catch (PrecondViolationExcept except)
{
  std:: cout << except.what() << std:: endl;
}
```

itemCount: 5

items

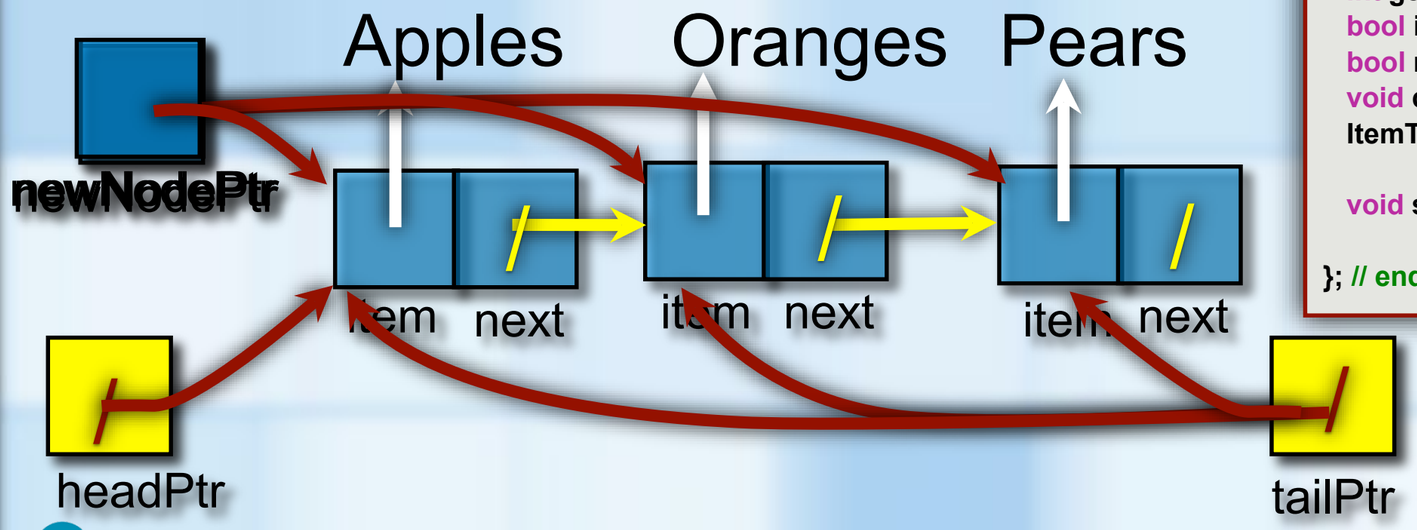
0	1	2	3	4	5	6	7	Array Index
Apples	Oranges	Pears	Cheese	Nachos				List Elements
1	2	3	4	5	6	7	8	List Position

Pearson

Linked List Implementation

THE CLASS LINKEDLIST

- **Data Fields**
  - **`headPtr`**
    - Reference to the first node in the list
  - **`tailPtr`**
    - reference to the last node in the list
  - **`itemCount`**
    - number of entries in the list



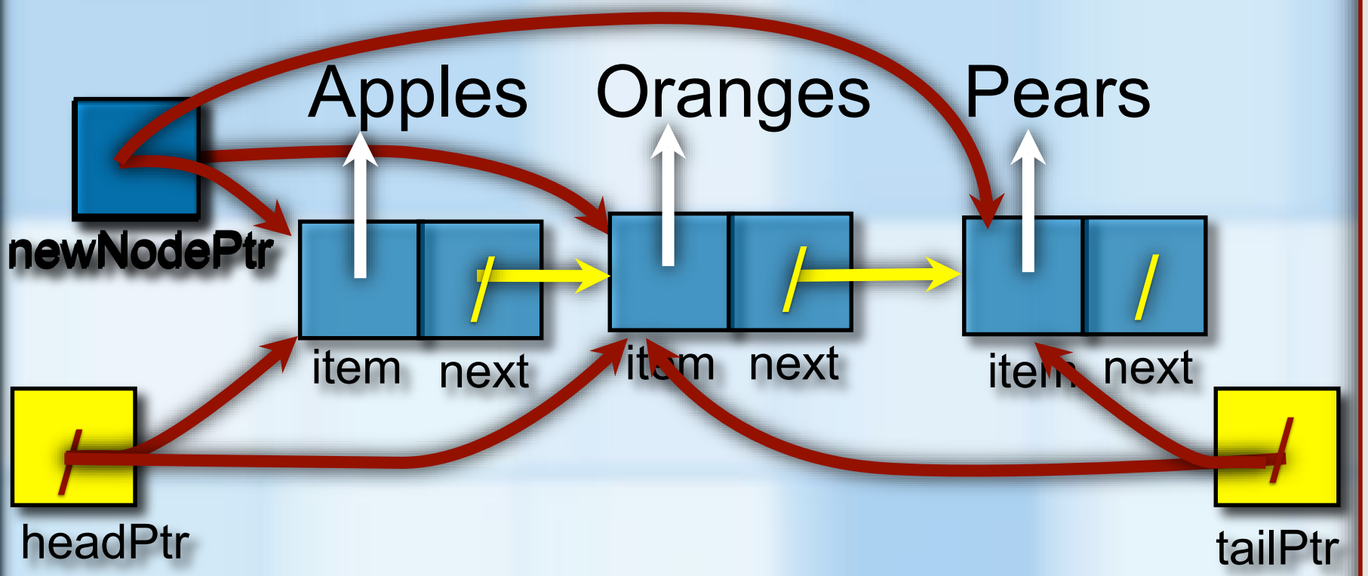

```
template<class ItemType>
class LinkedList : public ListInterface<ItemType>
{
private:
  Node<ItemType>* headPtr;
  Node<ItemType>* tailPtr;
  int itemCount;
  Node<ItemType>* getNodeAt(int position) const;

public:
  LinkedList();
  LinkedList(const LinkedList<ItemType>& aList);
  virtual ~LinkedList();

  bool isEmpty() const;
  int getLength() const;
  bool insert(int newPosition, const ItemType& someItem);
  bool remove(int position);
  void clear();
  ItemType getEntry(int position) const
                    throw(PrecondViolatedExcep);
  void setEntry(int position, const ItemType& someItem)
                    throw(PrecondViolatedExcep);
}; // end LinkedList
```

Pearson

Adding to a Linked List

- **Adding a node to a linked chain**
  - The chain is empty
  - Adding a node at the chain's beginning
  - Adding a node to the chain's tail
  - Adding a node between adjacent nodes

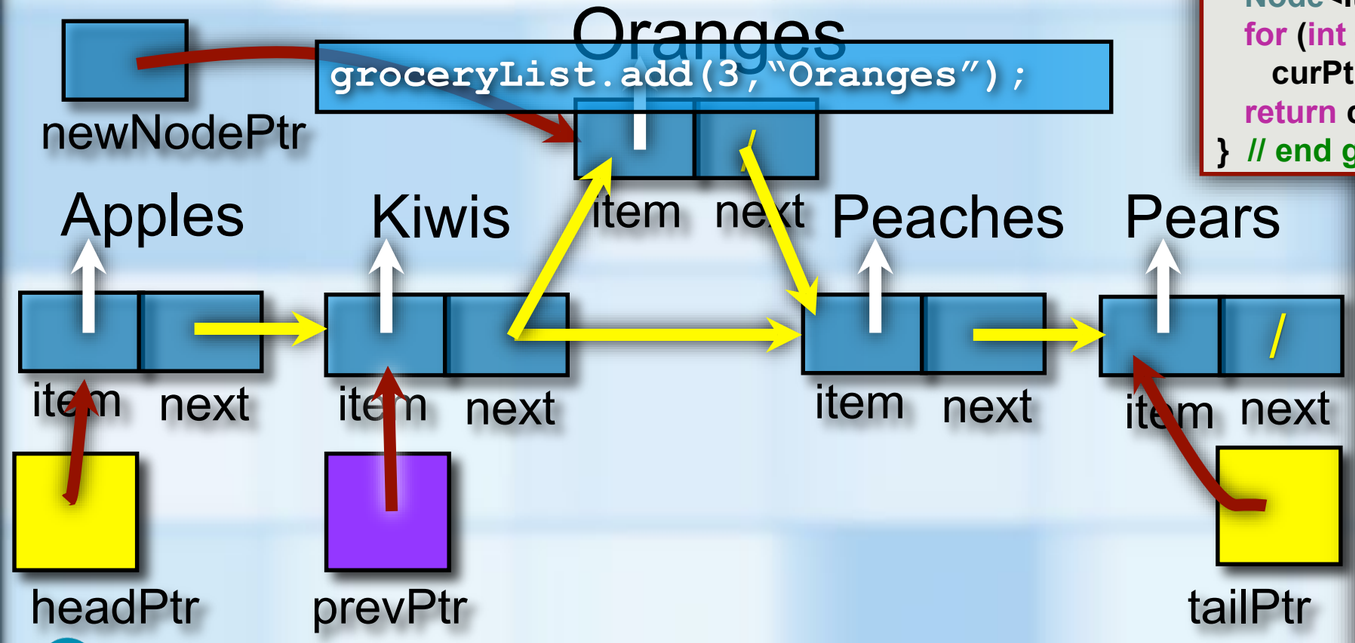
```
groceryList.add(3,"Pears");
```




```
template<class ItemType>
bool LinkedList<ItemType>::insert(int newPosition,
const ItemType& someItem)
{
  bool ableToInsert = (newPosition >= 1)
                      && (newPosition <= itemCount + 1);
  if (ableToInsert)
  {
    auto newNodePtr = new Node<ItemType>(someItem);

    if (isEmpty())
    {
      headPtr = newNodePtr;
      tailPtr = newNodePtr;
    }
    else if (newPosition == 1)
    {
      newNodePtr->setNext(headPtr);
      headPtr = newNodePtr;
    }
    else if (newPosition == itemCount + 1)
    {
      tailPtr->setNextNode(newNodePtr);
      tailPtr = newNodePtr;
    }
    else
    {
      Node<ItemType>* prevPtr =
                    getNodeAt(newPosition - 1);
      newNodePtr->setNext(prevPtr->getNext());
      prevPtr->setNext(newNodePtr);
    }  // end if
    itemCount++;  // Increase count of entries
  }  // end if
  return ableToInsert;
}  // end insert
```

Adding to a Linked List

- **Adding a node to a linked chain**
  - The chain is empty
  - Adding a node at the chain's beginning
  - Adding a node to the chain's tail
  - Adding a node between adjacent nodes




```
template<class ItemType>
bool LinkedList<ItemType>::insert(int newPosition,
                                  const ItemType& newEntry)
{
  bool ableToInsert = (newPosition >= 1)
                      && (newPosition <= itemCount + 1);
  if (ableToInsert)
  {
    Node<ItemType>* newNodePtr =
                                  new Node<ItemType>(newEntry);
```

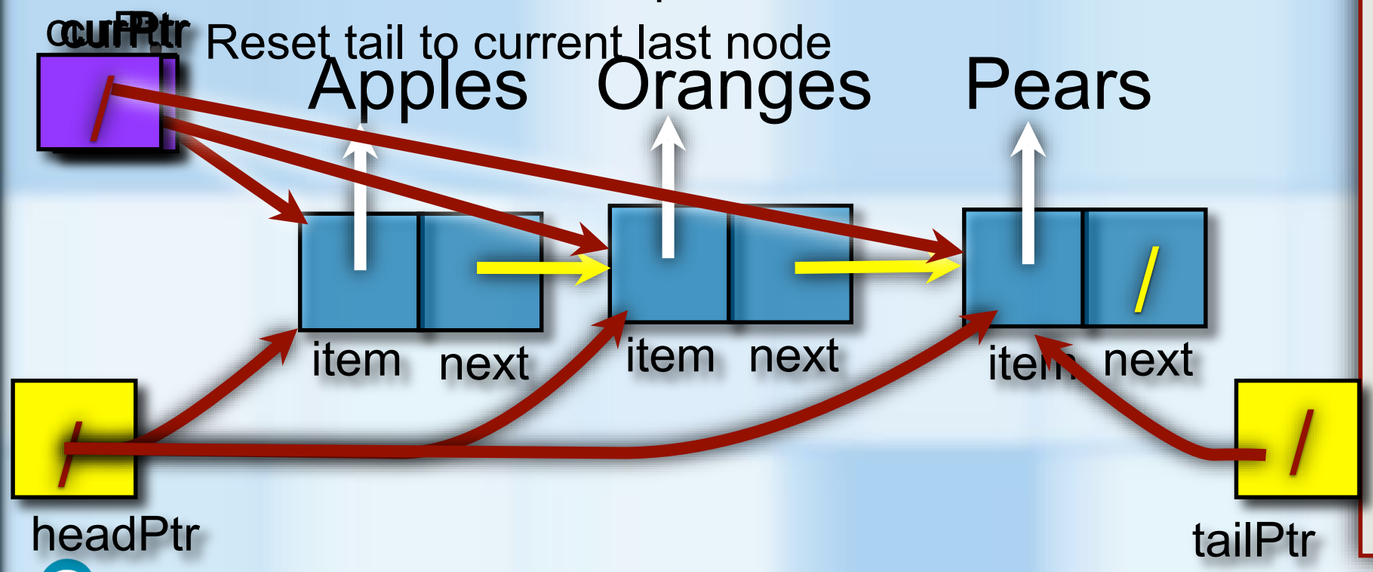
```
template<class ItemType>
Node<ItemType>* LinkedList<ItemType>::getNodeAt(int position) const
{
  assert( (position >= 1) && (position <= itemCount) );
  Node<ItemType>* curPtr = headPtr;
  for (int skip = 1; skip < position; skip++)
    curPtr = curPtr->getNext();
  return curPtr;
} // end getNodeAt
```

```
    {
      tailPtr->setNextNode(newNodePtr);
      tailPtr = newNodePtr;
    }
    else
    {
      Node<ItemType>* prevPtr =
                        getNodeAt(newPosition - 1);
      newNodePtr->setNext(prevPtr->getNext());
      prevPtr->setNext(newNodePtr);
    } // end if
    itemCount++;  // Increase count of entries
  } // end if
  return ableToInsert;
} // end insert
```

Pearson

Removing from a Linked List

- **Removing a node from a linked chain**
  - Removing the first entry
    - Check if list is now empty
      - Set tail to `nullptr`
  - Removing any other entry
    - Check if node in last position was removed
    - Reset tail to current last node

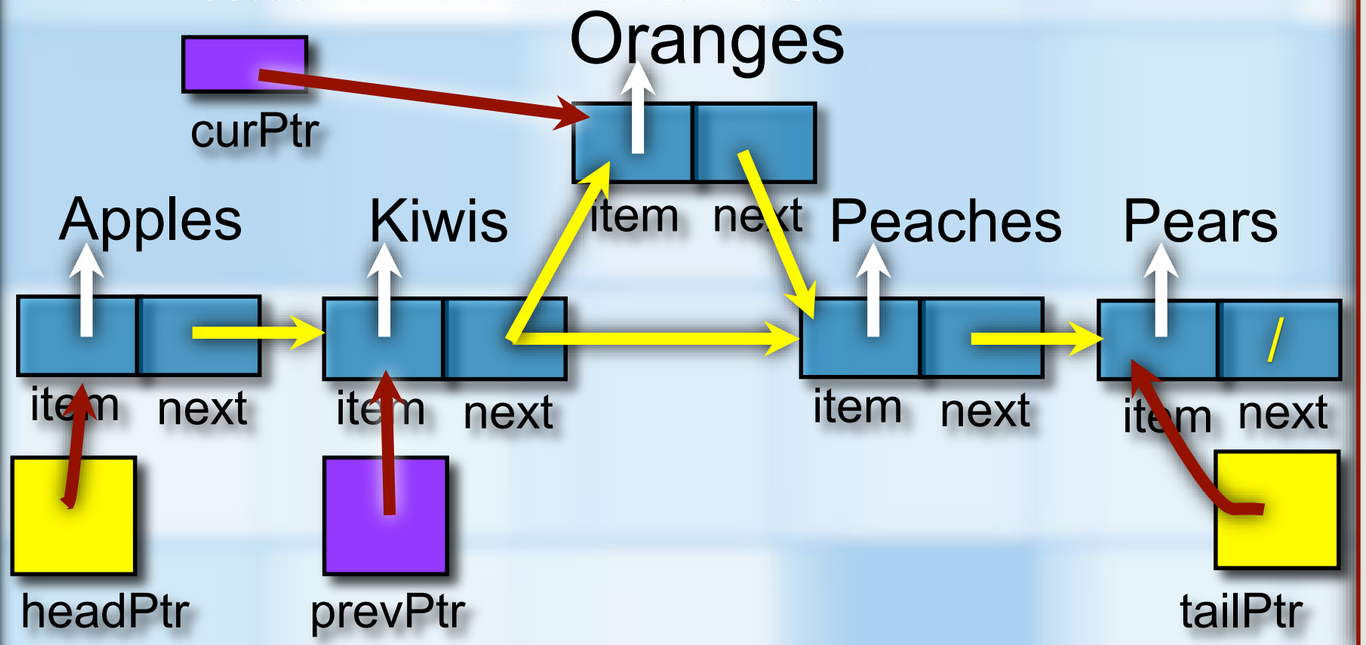



```
template<class ItemType>
bool LinkedList<ItemType>::remove(int position)
{
  bool ableToDelete = (position >= 1)
                                    && (position <= itemCount);
  if (ableToDelete)
  {
    Node<ItemType>* curPtr = nullptr;
    if (position == 1)
    {
      curPtr = headPtr; // save pointer to node
      headPtr = headPtr->getNext();
      if (itemCount == 1) tailPtr = nullptr;
    }
      else
      {
         // Find node that is before the one to delete
         Node<ItemType>* prevPtr = getNodeAt(position - 1);

         // Point to node to delete
         curPtr = prevPtr->getNext();
               prevPtr->setNext(curPtr->getNext());
         if (position == itemCount) tailPtr = prevPtr;
       } // end if

    // Return deleted node to system
    curPtr->setNext(nullptr);
    delete curPtr;
    curPtr = nullptr;

    itemCount--;  // Decrease count of entries
  } // end if

  return ableToDelete;
} // end remove
```

```
groceryList.remove(1);
```

Removing from a Linked List

- **Removing a node from a linked chain**
  - Removing the first entry
    - Check if list is now empty
      - Set tail to **`nullptr`**
  - Removing any other entry
    - Check if node in last position was removed
    - Reset tail to current last node

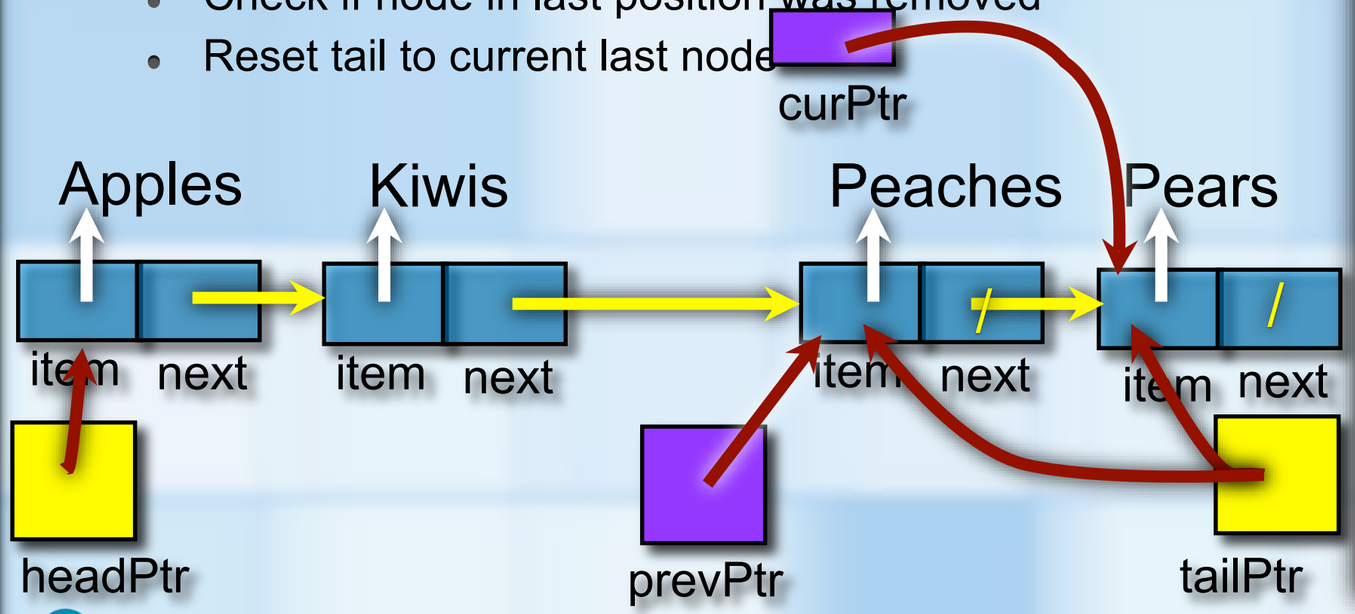



```
template<class ItemType>
bool LinkedList<ItemType>::remove(int position)
{
  bool ableToDelete = (position >= 1)
                                    && (position <= itemCount);
  if (ableToDelete)
  {
    Node<ItemType>* curPtr = nullptr;
    if (position == 1)
    {
      curPtr = headPtr; // save pointer to node
      headPtr = headPtr->getNext();
      if (itemCount == 1) tailPtr = nullptr;
    }
      else
      {
          // Find node that is before the one to delete
          Node<ItemType>* prevPtr = getNodeAt(position - 1);
          // Point to node to delete
          curPtr = prevPtr->getNext();
          prevPtr->setNext(curPtr->getNext());

        if (position == itemCount) tailPtr = prevPtr;
      } // end if

    // Return deleted node to system
    curPtr->setNext(nullptr);
    delete curPtr;
    curPtr = nullptr;

    itemCount--;  // Decrease count of entries
  } // end if

  return ableToDelete;
} // end remove
```

```
groceryList.remove(3);
```

Removing from a Linked List

- **Removing a node from a linked chain**
  - Removing the first entry
    - Check if list is now empty
      - Set tail to `nullptr`
  - Removing any other entry
    - Check if node in last position was removed
    - Reset tail to current last node




```
template<class ItemType>
bool LinkedList<ItemType>::remove(int position)
{
  bool ableToDelete = (position >= 1)
                                    && (position <= itemCount);
  if (ableToDelete)
  {
    Node<ItemType>* curPtr = nullptr;
    if (position == 1)
    {
      curPtr = headPtr; // save pointer to node
      headPtr = headPtr->getNext();
      if (itemCount == 1) tailPtr = nullptr;
    }
      else
      {
          // Find node that is before the one to delete
          Node<ItemType>* prevPtr = getNodeAt(position - 1);
          // Point to node to delete
          curPtr = prevPtr->getNext();
          prevPtr->setNext(curPtr->getNext());

          if (position == itemCount) tailPtr = prevPtr;
       } // end if

    // Return deleted node to system
    curPtr->setNext(nullptr);
    delete curPtr;
    curPtr = nullptr;

    itemCount--;  // Decrease count of entries
  } // end if

  return ableToDelete;
} // end remove
```

```
groceryList.remove(4);
```