CS302 - Data Structures

using C++

Topic: List Implementations

Kostas Alexis

The class ArrayList

- Data Fields

The class ArrayList

- Data Fields

```
template<class ItemType>
class ArrayList : public ListInterface<ItemType>
{
private:
     static const int DEFAULT_CAPACITY = 100; // Default list
     capacity
     ItemType items[DEFAULT_CAPACITY + 1]; // Array of list items
     (ignore                                         // items[0])
     int itemCount; // Current count of list items
     int maxCount; // Maximum capacity of the list
```

The class ArrayList

- Data Fields
- Constructor

```
template<class ItemType>
class ArrayList : public ListInterface<ItemType>
{
private:
    static const int DEFAULT_CAPACITY = 100; // Default list
    capacity
    ItemType items[DEFAULT_CAPACITY + 1]; // Array of list items
    (ignore                                          // items[0])
    int itemCount; // Current count of list items
    int maxCount; // Maximum capacity of the list
public:
    ArrayList();
    // Copy constructor and destructor are supplied by compiler
```

The class ArrayList

- Data Fields
- Constructor

```
ListInterface<string>* groceryList = newArrayList<string>();
```

```
itemCount
    0
```

```
template<class ItemType>
class ArrayList : public ListInterface<ItemType>
{
private:
     static const int DEFAULT_CAPACITY = 100; // Default list
     capacity
     ItemType items[DEFAULT_CAPACITY + 1]; // Array of list items
     (ignore                                            // items[0])
     int itemCount; // Current count of list items
     int maxCount; // Maximum capacity of the list
public:
     ArrayList();
     // Copy constructor and destructor are supplied by compiler
```

The class ArrayList

- Data Fields
- Constructor

```
ListInterface<string>* groceryList = newArrayList<string>();
```

itemCount
0

Items	0	1	2	3	4	5	6	7
	1	2	3	4	5	6	7	8

```
template<class ItemType>
class ArrayList : public ListInterface<ItemType>
{
private:
      static const int DEFAULT_CAPACITY = 100; // Default list
      capacity
      ItemType items[DEFAULT_CAPACITY + 1]; // Array of list items
      (ignore                                          // items[0])
      int itemCount; // Current count of list items
      int maxCount; // Maximum capacity of the list
public:
      ArrayList();
      // Copy constructor and destructor are supplied by compiler
```

The class ArrayList

- Data Fields
- Constructor

```
ListInterface<string>* groceryList = newArrayList<string>();
```

itemCount
0

Items

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

List Elements

```
template<class ItemType>
class ArrayList : public ListInterface<ItemType>
{
private:
     static const int DEFAULT_CAPACITY = 100; // Default list
     capacity
     ItemType items[DEFAULT_CAPACITY + 1]; // Array of list items
     (ignore                                         // items[0])
     int itemCount; // Current count of list items
     int maxCount; // Maximum capacity of the list
public:
     ArrayList();
     // Copy constructor and destructor are supplied by compiler
```

The class ArrayList

- Data Fields
- Constructor

```
ListInterface<string>* groceryList = newArrayList<string>();
```

itemCount
0

Items

ArrayIndex	0	1	2	3	4	5	6	7
List Elements								
	1	2	3	4	5	6	7	8

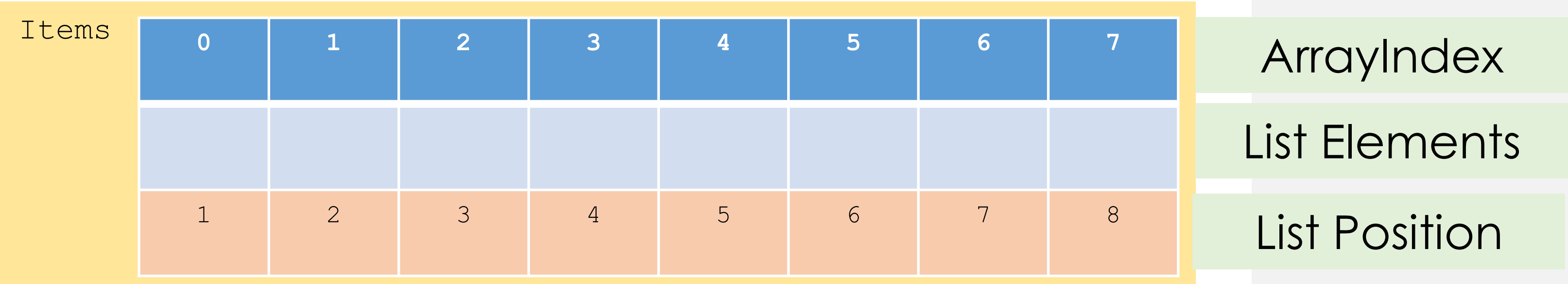
```
template<class ItemType>
class ArrayList : public ListInterface<ItemType>
{
private:
     static const int DEFAULT_CAPACITY = 100; // Default list
     capacity
     ItemType items[DEFAULT_CAPACITY + 1]; // Array of list items
     (ignore                                        // items[0])
     int itemCount; // Current count of list items
     int maxCount; // Maximum capacity of the list
public:
     ArrayList();
     // Copy constructor and destructor are supplied by compiler
```

The class ArrayList

- Data Fields
- Constructor

```
ListInterface<string>* groceryList = newArrayList<string>();
```

```
itemCount
0
```



Items								
ArrayIndex	0	1	2	3	4	5	6	7
List Elements								
List Position	1	2	3	4	5	6	7	8

```
template<class ItemType>
class ArrayList : public ListInterface<ItemType>
{
private:
     static const int DEFAULT_CAPACITY = 100; // Default list
     capacity
     ItemType items[DEFAULT_CAPACITY + 1]; // Array of list items
     (ignore                                      // items[0])
     int itemCount; // Current count of list items
     int maxCount; // Maximum capacity of the list
public:
     ArrayList();
     // Copy constructor and destructor are supplied by compiler
```

Array-based Implementation of ADT List

- Operations in UML form

```
+isEmpty(): boolean
+getLength(): integer
+insert(newPosition: integer, newEntry: ItemType): boolean
+remove(position: integer): boolean
+clear(): void
+getEntry(position: integer): ItemType
+replace(position: integer, newEntry: ItemType): ItemType
```

The class ArrayList

- Data Fields
- Constructor

```
ListInterface<string>* groceryList = newArrayList<string>();
```

itemCount
0

Items

0	1	2	3	4	5	6	7	ArrayIndex
								List Elements
1	2	3	4	5	6	7	8	List Position

```
template<class ItemType>
class ArrayList : public ListInterface<ItemType>
{
private:
      static const int DEFAULT_CAPACITY = 100; // Default list
      capacity
      ItemType items[DEFAULT_CAPACITY + 1]; // Array of list items
      (ignore                                         // items[0])
      int itemCount; // Current count of list items
      int maxCount; // Maximum capacity of the list
public:
      ArrayList();
      // Copy constructor and destructor are supplied by compiler
```

The class ArrayList

- Data Fields
- Constructor

```
ListInterface<string>* groceryList = newArrayList<string>();
```

itemCount
0

Items

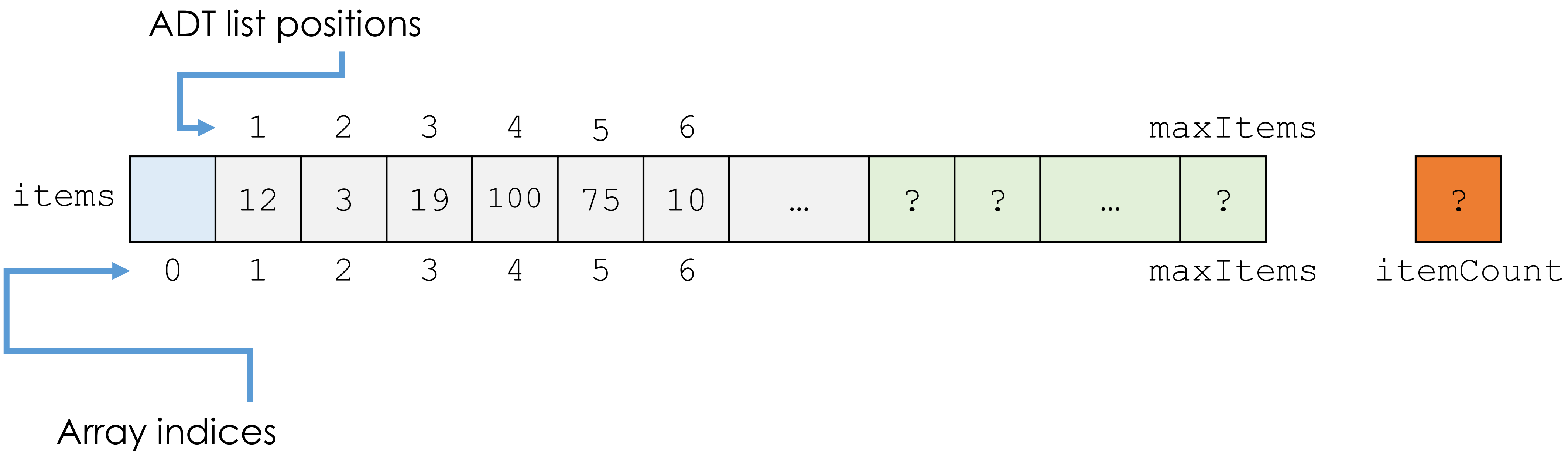
0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

```
template<class ItemType>
class ArrayList : public ListInterface<ItemType>
{
private:
      static const int DEFAULT_CAPACITY = 100; // Default list
      capacity
      ItemType items[DEFAULT_CAPACITY + 1]; // Array of list items
      (ignore                                          // items[0])
      int itemCount; // Current count of list items
      int maxCount; // Maximum capacity of the list
public:
      ArrayList();
      // Copy constructor and destructor are supplied by compiler
      bool isEmpty() const;
      int getLength() const;
      bool remove(int position);
      void clear();
      ItemType getEntry(int position) const
            throw(PrecondViolatedExcept);
       ItemType replace(int position, const ItemType& newEntry)
            throw(PrecondViolatedExcept);
};    // end ArrayList
```

Array-based Implementation of ADT List

- Array-based implementation is a natural choice
  - Both an array and a list identify their items by number
- However
  - ADT list has operations such as **`getLength`** that an array does not
  - Must keep track of number of entries

Array-based Implementation of ADT List



The Header File

```
/** ADT list: Array-based implementation
    @file ArrayList.h */

#ifndef ARRAY_LIST_
#define ARRAY_LIST_

#include “ListInterface.h”
#include “PrecondViolatedExcept.h”

template<class ItemType>
class ArrayList : public ListInterface<ItemType>
{
private:
      static const int DEFAULT_CAPACITY = 100; // Default list capacity
      ItemType items[DEFAULT_CAPACITY + 1]; // Array of list items (ignore
                                            // items[0])
      int itemCount; // Current count of list items
      int maxCount; // Maximum capacity of the list
public:
      ArrayList();
      // Copy constructor and destructor are supplied by compiler

      bool isEmpty() const;
      int getLength() const;
      bool remove(int position);
      void clear();
      ItemType getEntry(int position) const
            throw(PrecondViolatedExcept);
       ItemType replace(int position, const ItemType& newEntry)
            throw(PrecondViolatedExcept);
};    // end ArrayList

#include “ArrayList.cpp”
#endif
```

The Implementation File

Method **insert**

```
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
        // positions from itemCount down to newPosition
        // (no shift if newPosition == itemCount + 1)
        for (int pos = itemCount; pos >= newPosition; pos--)
                item[pos + 1] = items[pos];
        // Insert new Entry
        items[newPosition] = newEntry;
        itemCount++; // Increase count of entries
    } // end if
    return ableToInsert;
} // end getEntry
```

The Implementation File

Method **insert**

```
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
           // positions from itemCount down to newPosition
           // (no shift if newPosition == itemCount + 1)
           for (int pos = itemCount; pos >= newPosition; pos--)
                      item[pos + 1] = items[pos];
           // Insert new Entry
           items[newPosition] = newEntry;
           itemCount++; // Increase count of entries
     } // end if
     return ableToInsert;
} // end getEntry
```

The Implementation File

Method **`insert`**

```cpp
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
          // positions from itemCount down to newPosition
          // (no shift if newPosition == itemCount + 1)
          for (int pos = itemCount; pos >= newPosition; pos--)
                    item[pos + 1] = items[pos];
          // Insert new Entry
          items[newPosition] = newEntry;
          itemCount++; // Increase count of entries
     } // end if
     return ableToInsert;
} // end getEntry
```

Move items out of the way

The Implementation File

Method **`insert`**

```cpp
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
          // positions from itemCount down to newPosition
          // (no shift if newPosition == itemCount + 1)
          for (int pos = itemCount; pos >= newPosition; pos--)
                    item[pos + 1] = items[pos];
          // Insert new Entry
          items[newPosition] = newEntry;
          itemCount++; // Increase count of entries
     } // end if
     return ableToInsert;
} // end getEntry
```

Move items out of the way

The Implementation File

Method **insert**

```
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
           // positions from itemCount down to newPosition
           // (no shift if newPosition == itemCount + 1)
           for (int pos = itemCount; pos >= newPosition; pos--)
                      item[pos + 1] = items[pos];
           // Insert new Entry
           items[newPosition] = newEntry;
           itemCount++; // Increase count of entries
     } // end if
     return ableToInsert;
} // end getEntry
```

Move items out of the way

Increase itemCount

The Implementation

Items	0	1	2	3	4	5	6	7
	Apples	Orange	Cheese	Steaks	Nachos			
	1	2	3	4	5	6	7	8

Method **`insert`**

```
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
          // positions from itemCount down to newPosition
          // (no shift if newPosition == itemCount + 1)
          for (int pos = itemCount; pos >= newPosition; pos--)
                    item[pos + 1] = items[pos];
          // Insert new Entry
          items[newPosition] = newEntry;
          itemCount++; // Increase count of entries
     } // end if
     return ableToInsert;
} // end getEntry
```

Move items out of the way

Increase itemCount

The Implementation File

Method **`insert`**

Say we want to insert item at position 4

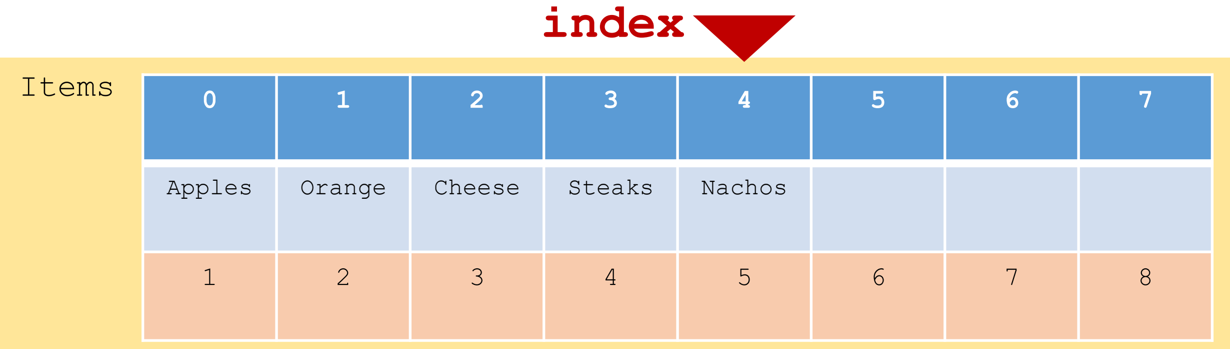
```
template<class ItemType>
bool ArrayList<ItemType>::insert(int newPosition, const ItemType& newEntry)
{
    bool ableToInsert = (newPosition >= 1) && (newPosition <= itemCount + 1) && (itemCount < maxItems);
    if (ableToInsert)
    {
        // Make room for new entry by shifting all entries
        // positions from itemCount down to newPosition
        // (no shift if newPosition == itemCount + 1)
        for (int pos = itemCount; pos >= newPosition; pos--)
                item[pos + 1] = items[pos];
        // Insert new Entry
        items[newPosition] = newEntry;
        itemCount++; // Increase count of entries
    } // end if
    return ableToInsert;
} // end getEntry
```

Move items out of the way

Increase itemCount

The Implementation

Method **insert**



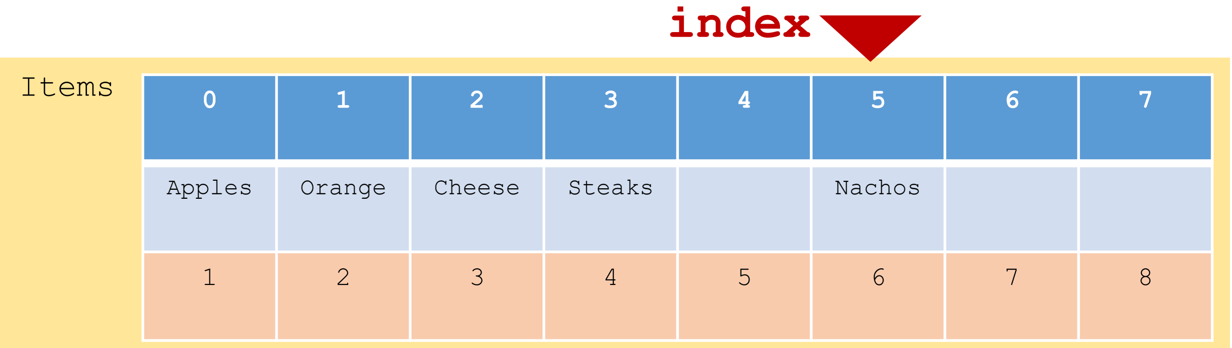
Items	0	1	2	3	4	5	6	7
	Apples	Orange	Cheese	Steaks	Nachos			
	1	2	3	4	5	6	7	8

```
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
        // positions from itemCount down to newPosition
        // (no shift if newPosition == itemCount + 1)
        for (int pos = itemCount; pos >= newPosition; pos--)
                item[pos + 1] = items[pos];
        // Insert new Entry
        items[newPosition] = newEntry;
        itemCount++; // Increase count of entries
    } // end if
    return ableToInsert;
} // end getEntry
```

Executed because the number of items is greater than the position we want to insert to so they have to be moved out of the way.

The Implementation

Method **insert**



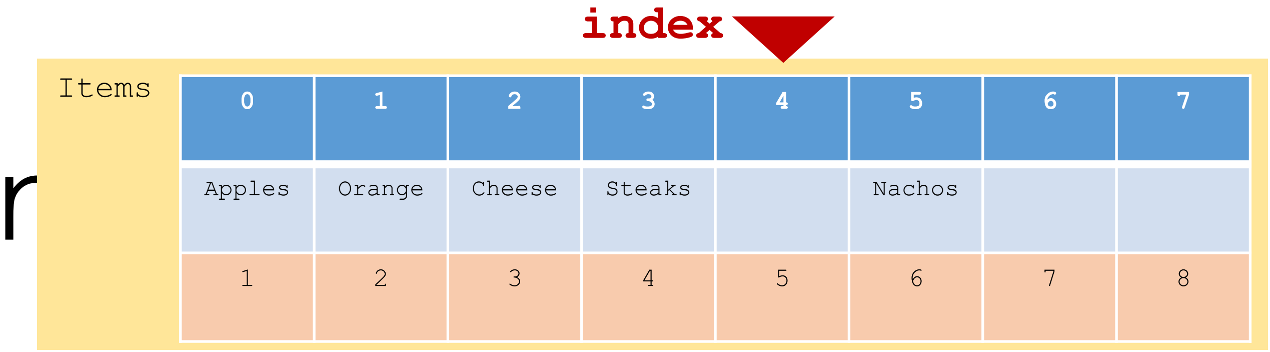
Items	0	1	2	3	4	5	6	7
	Apples	Orange	Cheese	Steaks		Nachos		
	1	2	3	4	5	6	7	8

```
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
        // positions from itemCount down to newPosition
        // (no shift if newPosition == itemCount + 1)
        for (int pos = itemCount; pos >= newPosition; pos--)
                item[pos + 1] = items[pos];
        // Insert new Entry
        items[newPosition] = newEntry;
        itemCount++; // Increase count of entries
    } // end if
    return ableToInsert;
} // end getEntry
```

Executed because the number of items is greater than the position we want to insert to so they have to be moved out of the way.

The Implementation

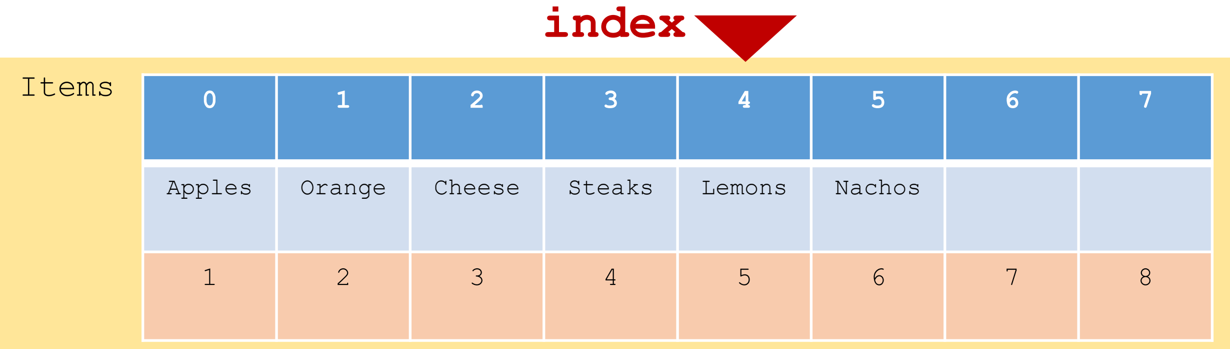
Method **insert**



Items	0	1	2	3	4	5	6	7
	Apples	Orange	Cheese	Steaks		Nachos		
	1	2	3	4	5	6	7	8

```
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
        // positions from itemCount down to newPosition
        // (no shift if newPosition == itemCount + 1)
        for (int pos = itemCount; pos >= newPosition; pos--)
                item[pos + 1] = items[pos];
        // Insert new Entry
        items[newPosition] = newEntry;
        itemCount++; // Increase count of entries
    } // end if
    return ableToInsert;
} // end getEntry
```

Executed because the number of items is greater than the position we want to insert to so they have to be moved out of the way.



Items	0	1	2	3	4	5	6	7
	Apples	Orange	Cheese	Steaks	Lemons	Nachos		
	1	2	3	4	5	6	7	8

The Implementation

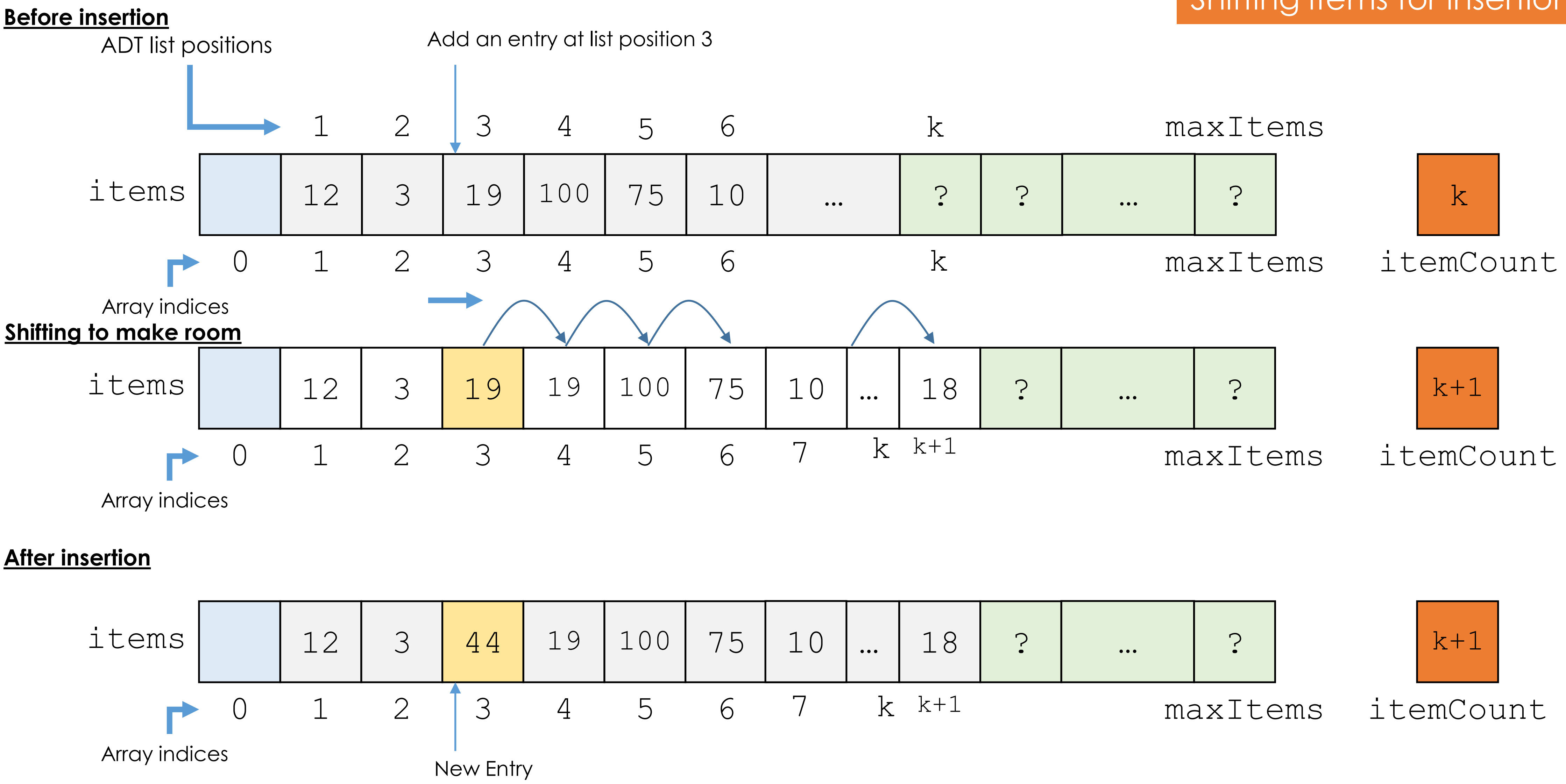
Method **insert**

```
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
          // positions from itemCount down to newPosition
          // (no shift if newPosition == itemCount + 1)
          for (int pos = itemCount; pos >= newPosition; pos--)
                    item[pos + 1] = items[pos];
          // Insert new Entry
          items[newPosition] = newEntry;
          itemCount++; // Increase count of entries
     } // end if
     return ableToInsert;
} // end getEntry
```

Executed because the number of items is greater than the position we want to insert to so they have to be moved out of the way.

The Implementation File

Shifting items for insertion



Before insertion

ADT list positions

Add an entry at list position 3

		1	2	3	4	5	6		k			maxItems	
items		12	3	19	100	75	10	…	?	?	…	?	k
	0	1	2	3	4	5	6		k			maxItems	itemCount

Array indices

Shifting to make room

items		12	3	19	19	100	75	10	…	18	?	…	?	k+1
	0	1	2	3	4	5	6	7	k	k+1			maxItems	itemCount

Array indices

After insertion

items		12	3	44	19	100	75	10	…	18	?	…	?	k+1
	0	1	2	3	4	5	6	7	k	k+1			maxItems	itemCount

Array indices

New Entry

The Implementation File

Method **`remove`**

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

        itemCount--; // Decrease count of entries
    } // end if
    return ableToRemove
} // end remove
```

The Implementation File

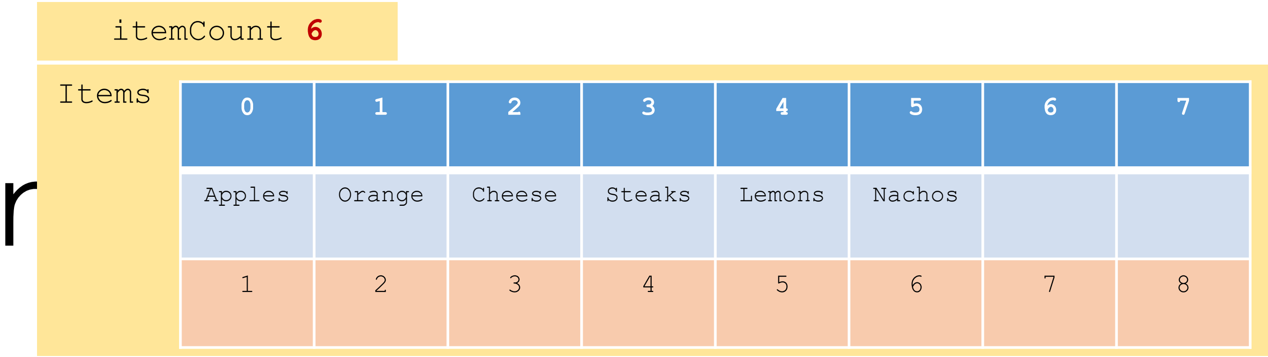
Method **`remove`**

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

        itemCount--; // Decrease count of entries
    } // end if
    return ableToRemove
} // end remove
```

The Implementatior

Method **remove**



itemCount **6**

Items

0	1	2	3	4	5	6	7
Apples	Orange	Cheese	Steaks	Lemons	Nachos		
1	2	3	4	5	6	7	8

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

            itemCount--; // Decrease count of entries
      } // end if
      return ableToRemove
} // end remove
```

The Implementation

Method **remove**

itemCount **6**

Items	0	1	2	3	4	5	6	7
	Apples	Orange	Cheese	Steaks	Lemons	Nachos		
	1	2	3	4	5	6	7	8

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

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

        itemCount--; // Decrease count of entries
    } // end if
    return ableToRemove
} // end remove
```

The Implementation

Method **remove**

itemCount 6

Items	0	1	2	3	4	5	6	7
	Apples	Orange	Cheese	Steaks	Lemons	Nachos		
	1	2	3	4	5	6	7	8

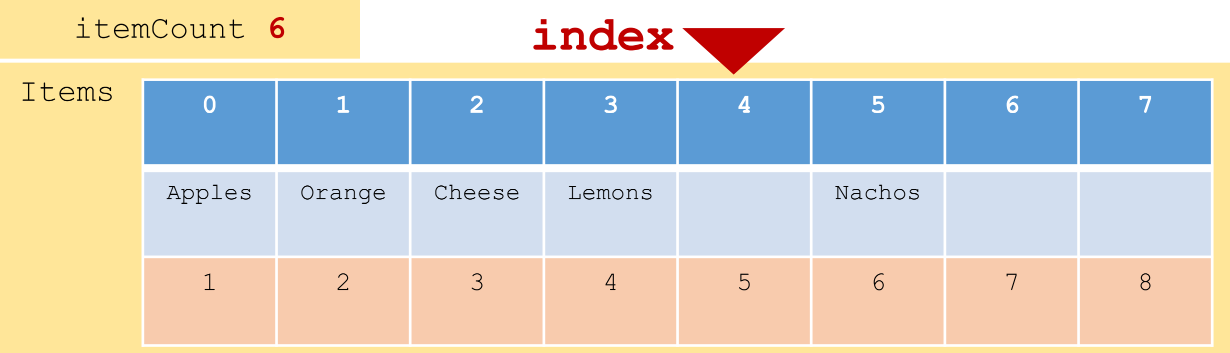
```
groceryList->remove(4);
```

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

        itemCount--; // Decrease count of entries
    } // end if
    return ableToRemove
} // end remove
```

The Implementation

Method **remove**



itemCount 6

index

Items	0	1	2	3	4	5	6	7
	Apples	Orange	Cheese	Lemons		Nachos		
	1	2	3	4	5	6	7	8

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

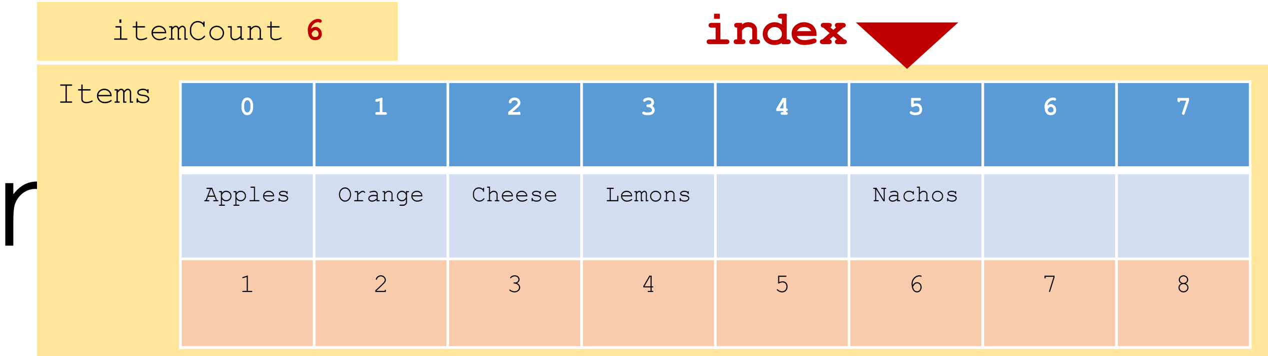
```
template<class ItemType>
bool ArrayList<ItemType>::remove(int position)
{
      bool ableToRemove = (position >= 1) && (position <= itemCount);
      if (ableToRemove)
      {
            // Remove entry by shifting all entries after the one at
            // position toward the beginning of the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

            itemCount--; // Decrease count of entries
      } // end if
      return ableToRemove
} // end remove
```

The Implementation

Method **remove**

`groceryList->remove(4);`



itemCount 6

index

Items

0	1	2	3	4	5	6	7
Apples	Orange	Cheese	Lemons		Nachos		
1	2	3	4	5	6	7	8

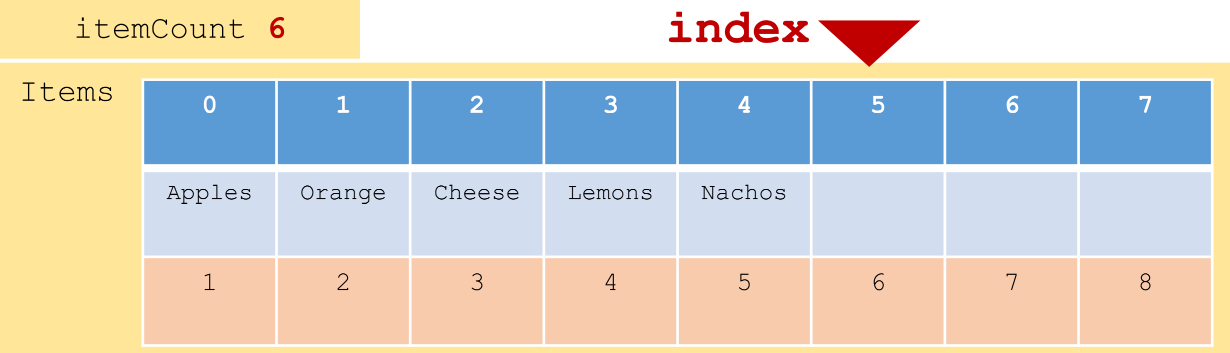
```cpp
template<class ItemType>
bool ArrayList<ItemType>::remove(int position)
{
    bool ableToRemove = (position >= 1) && (position <= itemCount);
    if (ableToRemove)
    {
        // Remove entry by shifting all entries after the one at
        // position toward the beginning of the array
    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

        itemCount--; // Decrease count of entries
    } // end if
    return ableToRemove
} // end remove
```

The Implementation

Method **remove**

`groceryList->remove(4);`



itemCount 6

index

Items

0	1	2	3	4	5	6	7
Apples	Orange	Cheese	Lemons	Nachos			
1	2	3	4	5	6	7	8

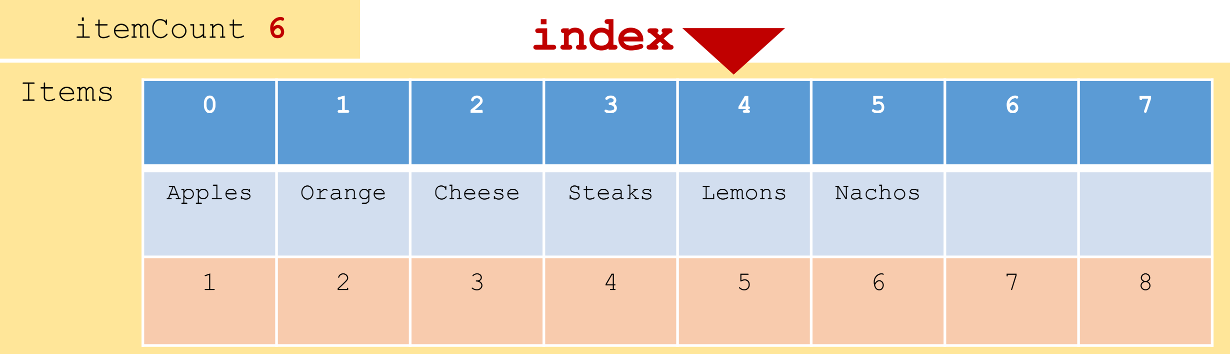
```
template<class ItemType>
bool ArrayList<ItemType>::remove(int position)
{
    bool ableToRemove = (position >= 1) && (position <= itemCount);
    if (ableToRemove)
    {
        // Remove entry by shifting all entries after the one at
        // position toward the beginning of the array
       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

        itemCount--; // Decrease count of entries
    } // end if
    return ableToRemove
} // end remove
```

The Implementation

Method **remove**

`groceryList->remove(4);`



itemCount 6

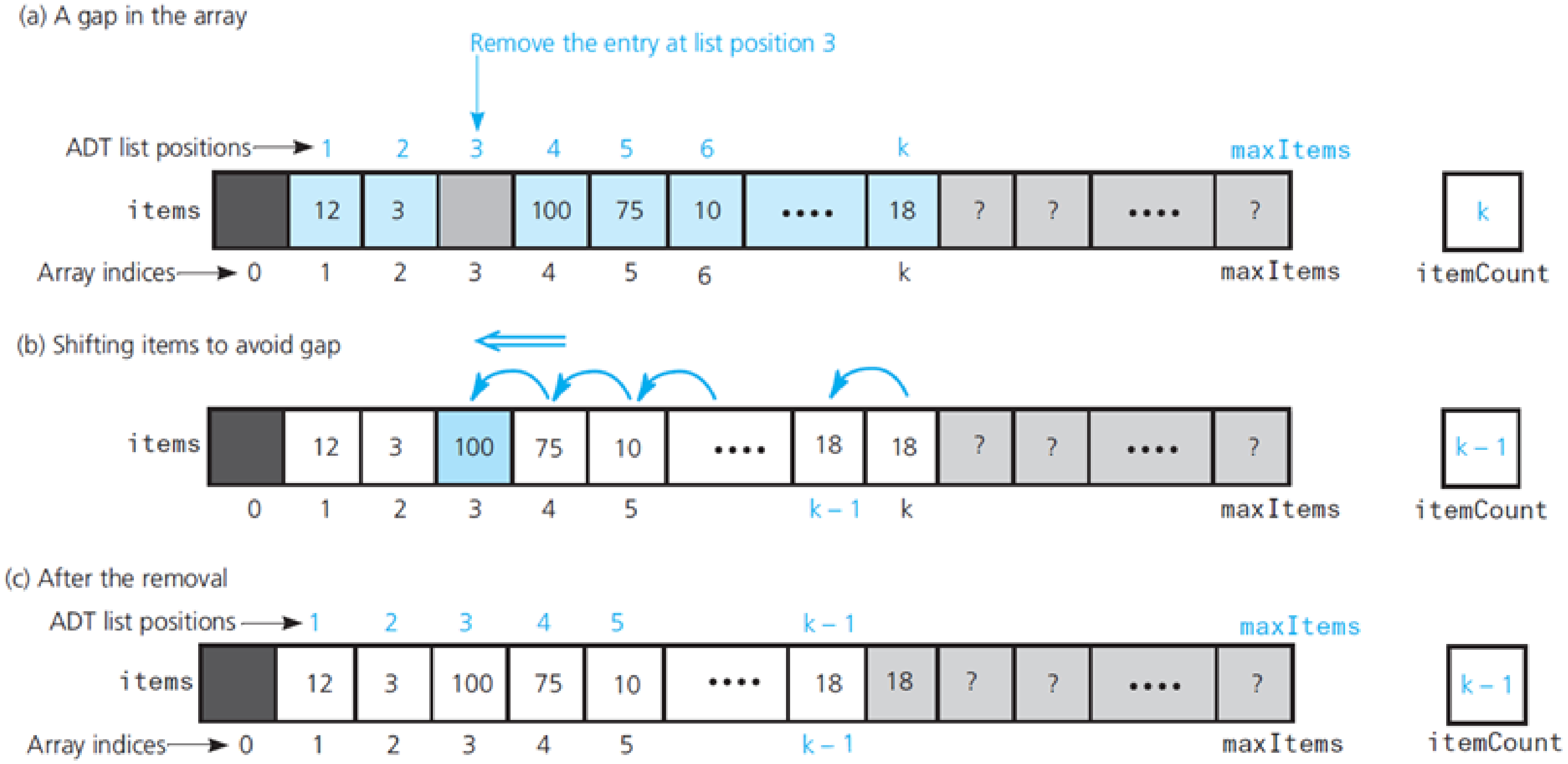
Items	0	1	2	3	4	5	6	7
	Apples	Orange	Cheese	Steaks	Lemons	Nachos		
	1	2	3	4	5	6	7	8

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

        itemCount--; // Decrease count of entries
    } // end if
    return ableToRemove
} // end remove
```

The Implementation File

Shifting items to remove



The Implementation File

Method **replace**

```
template<class ItemType>
ItemType ArrayList<ItemType>::replace(int position, const ItemType& newEntry) throw(PrecondViolatedExcept)
{
      // Enforce precondition
      bool ableToSet = (position >= 1) && (position <= itemCount);
      if (ableToSet)
      {
            ItemType oldEntry = items[position];
            items[position] = newEntry;
            return oldEntry;
      }
       else
      {
            std::string message = “replace() called with an empty list or”;
            message = message + “invalid position.”;
            throw(PrecondViolatedExcept(message));
      } // end if
} // end replace
```

The Implementation File

Constructor, methods **isEmpty** and **getLength**

```
template<class ItemType>
ArrayList<ItemType>::ArrayList() : itemCount(0), maxItems(DEFAULT_CAPACITY)
{
} // end default constructor
{
    template<class ItemType>
    bool ArrayList<ItemType>::isEmpty() const
    {
        return itemCount == 0;
    } // end isEmpty
    template<class ItemType>
    int ArrayList<ItemType>::getLength() const
    {
        return itemCount;
    } // end getLength
```

The Implementation File

Method **getEntry**

```
template<class ItemType>
ItemType ArrayList<ItemType>::getEntry(int newPosition) const throw(PrecondViolatedExcept)
{
     // Enforce precondition
     bool ableToGet = (position >= 1) && (position <= itemCount);
     if (ableToGet)
     {
          return items[position];
      else
     {
          std::string message = “getEntry() called with an empty list or”;
          message = message + “invalid position.”;
          throw(PrecondViolatedExcept(message));
     } // end if
} // end getEntry
```

The Implementation File

Method **clear**

```
template<class ItemType>
void ArrayList<ItemType>::clear()
{
     itemCount = 0;
} // end clear
```

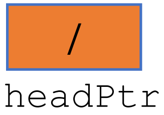
The Class LinkedList

- Data Fields
  - **`headPtr`**
    - Reference to the first node in the list
  - **`itemCount`**
    - Number of entries in the list

The Class LinkedList

- Data Fields
    - **headPtr**
        - Reference to the first 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; // optional – check implementation
     int itemCount; // Current count of list items
     Node<ItemType>* getNodeAt(int position) const;
public:
     LinkedList();
     LinkedList(const LinkedList<ItemType>& aList);
     virtual ~LinkedList();
     bool isEmpty() const;
     int getLength() const;
     bool remove(int position);
     void clear();
     ItemType getEntry(int position) const
          throw(PrecondViolatedExcept);
      ItemType replace(int position, const ItemType& newEntry)
          throw(PrecondViolatedExcept);
};    // end LinkedList
```

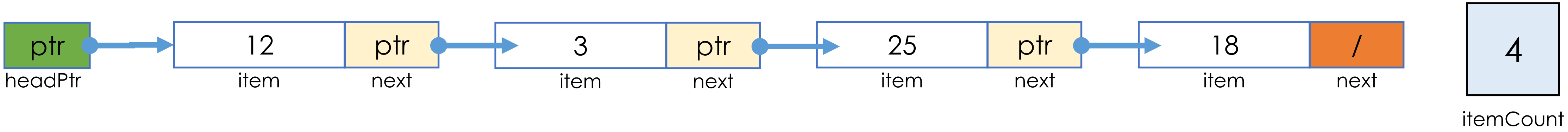


Link-based Implementation of ADT List

- We can use C++ pointers instead of an array to implement the ADT list
  - Link-based implementation does not shift items during insertion and removal operations
  - We need to represent items in the list and its length

Link-based Implementation of ADT List

- A link-based implementation of the ADT list



The Header File

```
/** ADT list: Linked-based implementation
    @file LinkedList.h */

#ifndef LINKED_LIST_
#define LINKED_LIST_

#include “ListInterface.h”
#include “Node.h”
#include “PrecondViolatedExcept.h”

template<class ItemType>
class LinkedList : public ListInterface<ItemType>
{
private:
      Node<ItemType>* headPtr; // Pointer to first node in chain
                               // (contains the first entry in the list)
      int itemCount; // Current count of list items
      // Locates a specified node in a linked list
      // @pre position is the number of the desired node;
      //      position >= 1 and position <= itemCount
      // @post The node is found and a pointer to it is returned
      // @param position The number of the node to locate
      //@return A pointer to the node at the given position
      Node<ItemType>* getNodeAt(int position) const;
```

```
public:
      LinkedList();
      LinkedList(const LinkedList<ItemType>& aList);
      virtual ~LinkedList();
      bool isEmpty() const;
      int getLength() const;
      bool remove(int position);
      void clear();
      ItemType getEntry(int position) const
            throw(PrecondViolatedExcept);
       ItemType replace(int position, const ItemType& newEntry)
            throw(PrecondViolatedExcept);
};    // end LinkedList

#include “LinkedList.cpp”
#endif
```

The Implementation File

Constructor

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

The Implementation File

Method **getEntry**

```
template<class ItemType>
ItemType LinkedList<ItemType>::getEntry(int position) const throw(PrecondViolatedExcept)
{
     // enforce precondition
     bool ableToGet = (position >= 1) && (position <= itemCount);
     if (ableToGet)
          Node<ItemType>* nodePtr = getNodeAt(position)
          return nodePtr->getItem();
     else
     {
          std::string message = “getEntry() called with an empty list or “;
          message = message + “invalid position.”;
          throw(PrecondViolatedExcept(message));
     } // end if
} // end getEntry
```

The Implementation File

Method **getNodeAt**

```
template<class ItemType>
Node<ItemType>* LinkedList<ItemType>::getNodeAt(int position) const
{
      // debugging check of precondition
      assert( (position >= 1) && (position <= itemCount) );
      // Count from the beginning of the chain
      Node<ItemType>* curPtr = headPtr;
      for (int skip = 1; skip < position; skip++)
            curPtr = curPtr->getNext();
      return curPtr;
} // end getNodeAt
```

The Implementation File

- The Insertion process requires three high-level steps
  - Create a new node and store the new data in it.
  - Determine the point of insertion.
  - Connect the new node to the linked chain by changing pointers

The Implementation File

Method **insert**

```
template<class ItemType>
bool LinkedList<ItemType>::insert(int newPosition, const ItemType& newEntry)
{
    bool ableToInsert = (newPosition >= 1) && (newPosition <= itemCount + 1)
    if (ableToInsert)
    {
        // Create a new node containing the new entry
        Node<ItemType>* newNodePtr = new Node<ItemType>(newEntry);
        // Attach a new node to chain
        if (newPosition == 1)
        {
            // Insert new node at beginning of chain
            newNodePtr->setNext(headPtr);
            headPtr = newNodePtr;

        }
        else
```

The Implementation File

Method **insert**

```
        else
        {
            // Find node that will be before new node
            Node<ItemType>* prevPtr = getNodeAt(newPosition - 1);
            // Insert new node after node to which prevPtr points
            newNodePtr->setNext(prevPtr->getNext());
            prevPtr->setNext(newNodePtr);
        } // end if
        itemCount++; // Increase count of entries
    } // end if
    return ableToInsert;
} // end insert
```

The Implementation File

Method **insert**

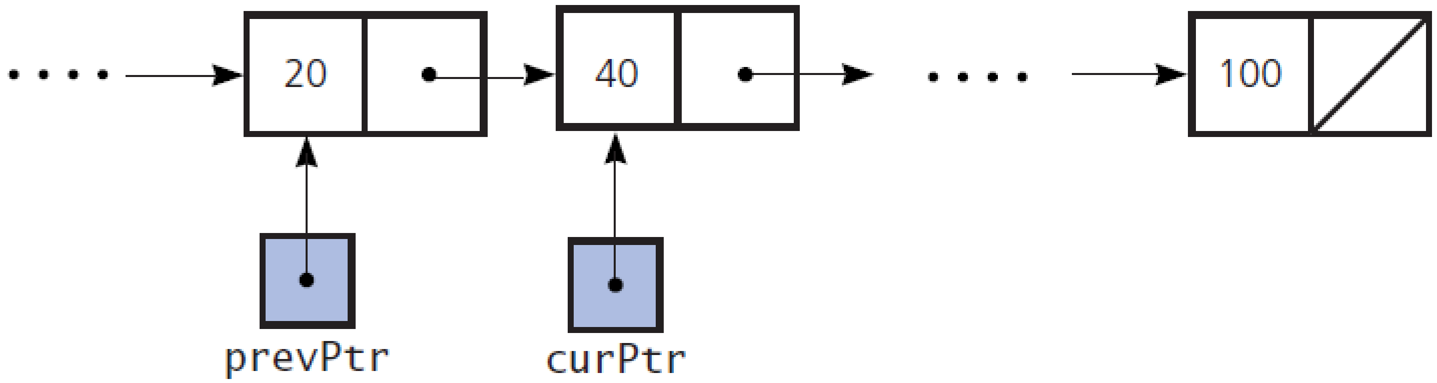
```
template<class ItemType>
bool LinkedList<ItemType>::insert(int newPosition, const ItemType& newEntry)
{
        bool ableToInsert = (newPosition >= 1) && (newPosition <= itemCount + 1)
        if (ableToInsert)
        {
               // Create a new node containing the new entry
               Node<ItemType>* newNodePtr = new Node<ItemType>(newEntry);
               // Attach a new node to chain
               if (newPosition == 1)
               {
                      // Insert new node at beginning of chain
                      newNodePtr->setNext(headPtr);
                      headPtr = newNodePtr;

               }
               else
               {
                      // Find node that will be before new node
                      Node<ItemType>* prevPtr = getNodeAt(newPosition - 1);
                      // Insert new node after node to which prevPtr points
                      newNodePtr->setNext(prevPtr->getNext());
                      prevPtr->setNext(newNodePtr);
               } // end if
               itemCount++; // Increase count of entries
        } // end if
        return ableToInsert;
} // end insert
```

The Implementation File

Method **`insert`**

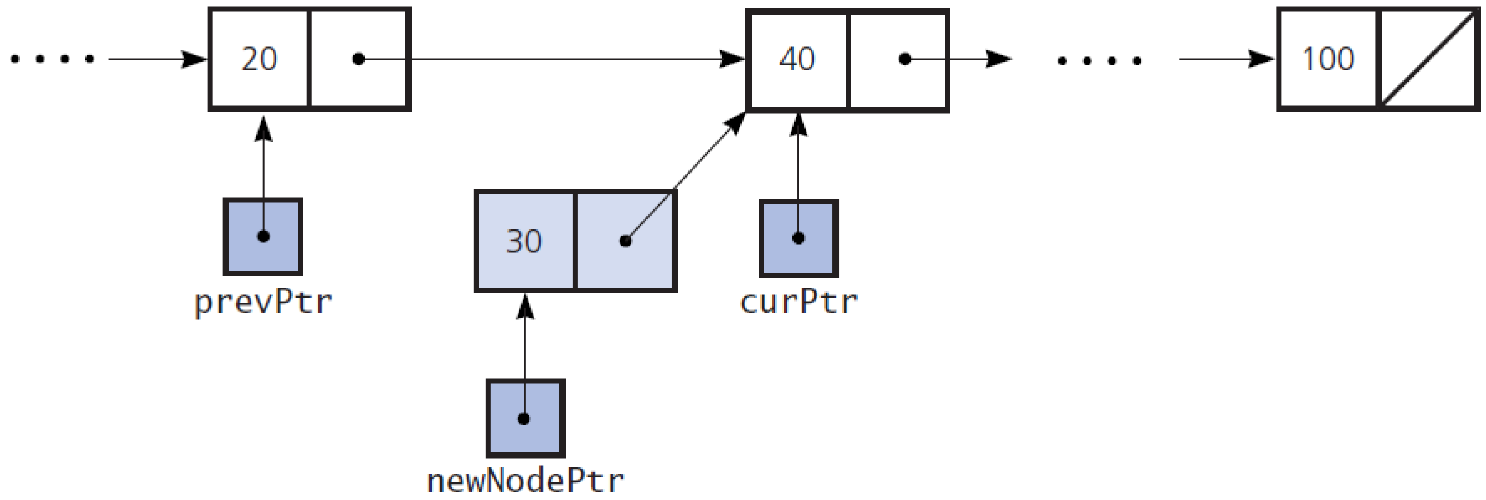
(a) Before the insertion of a new node



The Implementation File

Method **insert**

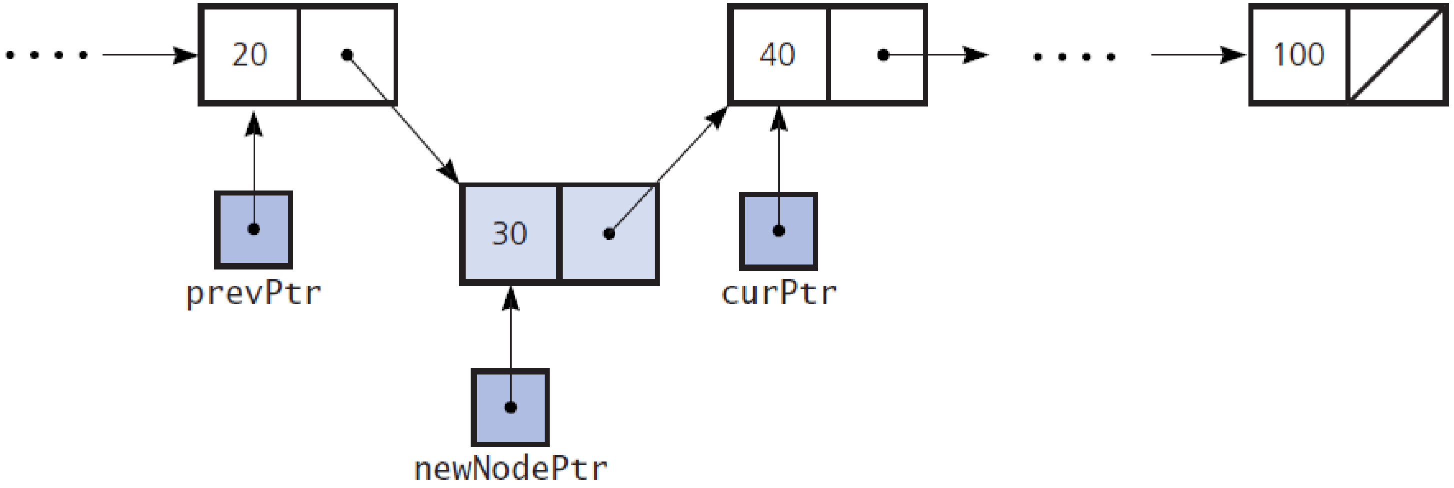
(b) After newNodePtr->setNext(curPtr) executes



The Implementation File

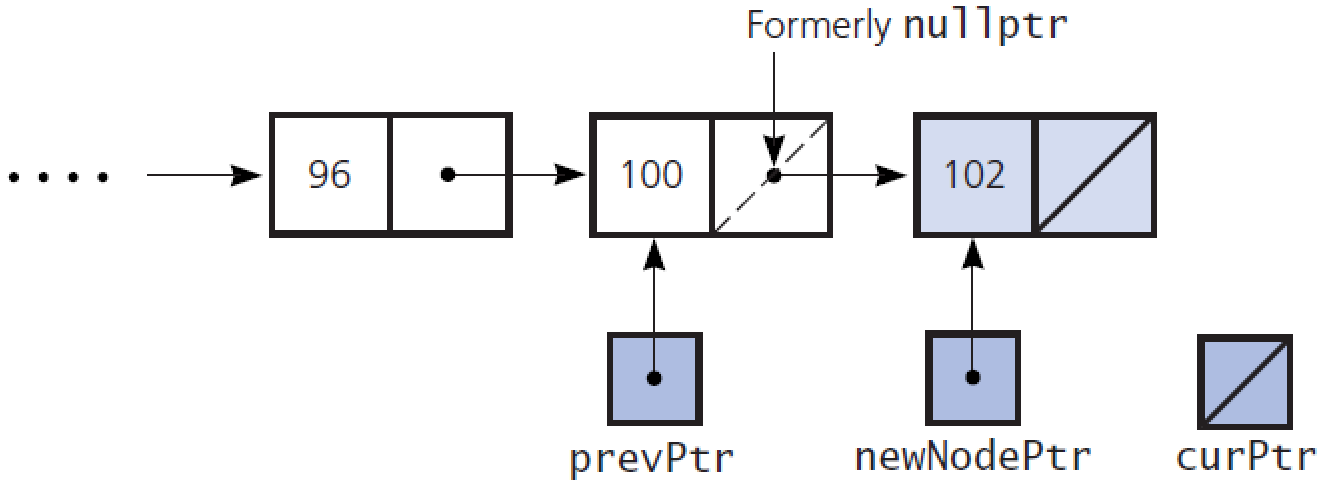
Method **`insert`**

(c) After `prevPtr->setNext(newNodePtr)` executes



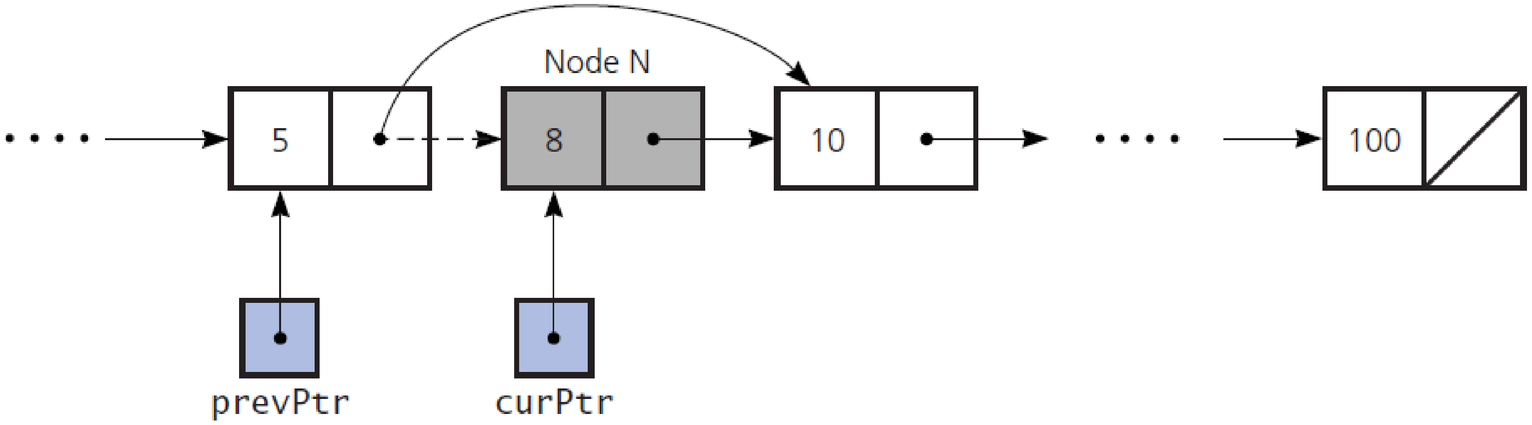
The Implementation File

Method **`insert`**



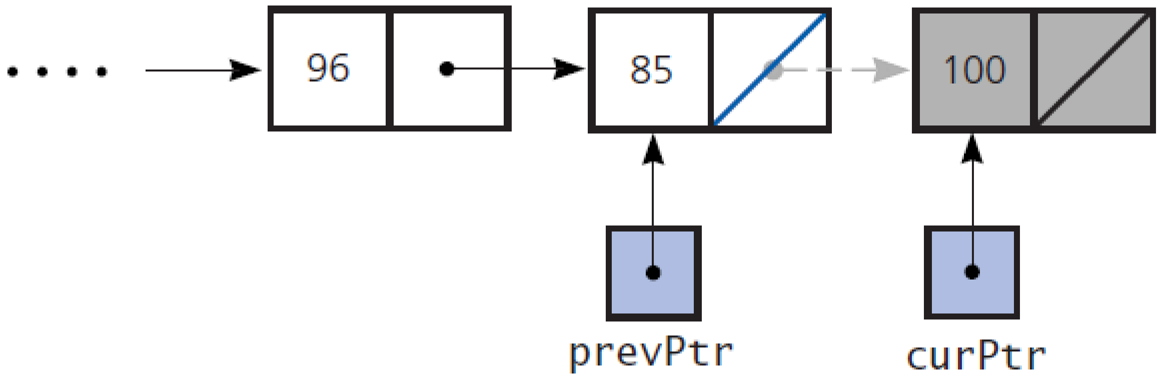
The Implementation File

Method **remove**



The Implementation File

Method **`remove`**



The Implementation File

Method **remove**

```
template<class ItemType>
bool LinkedList<ItemType>::remove(int position)
{
     bool ableToRemove = (position >= 1) && (position <= itemCount);
     if (ableToRemove)
     {
          Node<ItemType>* curPtr = nullptr;
          if (position == 1)
          {
             // Remove the first node in th chain
             curPtr = headPtr; // Save pointer to node // save pointer to next node
             headPtr = headPtr->getNext();
          }
          else
```

The Implementation File

Method **remove**

```
           else
           {
              // Find node that is before the one to remove
              Node<ItemType>* prevPtr = getNodeAt(position - 1);
              // Point to node to remove
              curPtr = prevPtr->getNext();
              // Disconnect indicated node from chain by connecting the prior node with the one after
              prevPtr->setNext(curPtr->getNext());
           } // end if
           curPtr->getNext(nullptr);
           delete curPtr;
           curPtr = nullptr;
           itemCount--; // Decrease count of entries
     } // end if
     return ableToRemove;
} // end remove
```

The Implementation File

Method **remove**

```
template<class ItemType>
bool LinkedList<ItemType>::remove(int position)
{
        bool ableToRemove = (position >= 1) && (position <= itemCount);
        if (ableToRemove)
        {
                Node<ItemType>* curPtr = nullptr;
                if (position == 1)
                {
                     // Remove the first node in th chain
                     curPtr = headPtr; // Save pointer to node // save pointer to next node
                     headPtr = headPtr->getNext();
                }
                else
                {
                     // Find node that is before the one to remove
                     Node<ItemType>* prevPtr = getNodeAt(position - 1);
                     // Point to node to remove
                     curPtr = prevPtr->getNext();
                     // Disconnect indicated node from chain by connecting the prior node with the one after
                     prevPtr->setNext(curPtr->getNext());
                } // end if
                curPtr->getNext(nullptr);
                delete curPtr;
                curPtr = nullptr;
                itemCount--; // Decrease count of entries
        } // end if
        return ableToRemove;
} // end remove
```

The Implementation File

Method **`clear`**

```
template<class ItemType>
ItemType LinkedList<ItemType>::clear()
{
    while(!isEmpty())
        remove(1);
} // end clear
```

The Implementation File

Destructor

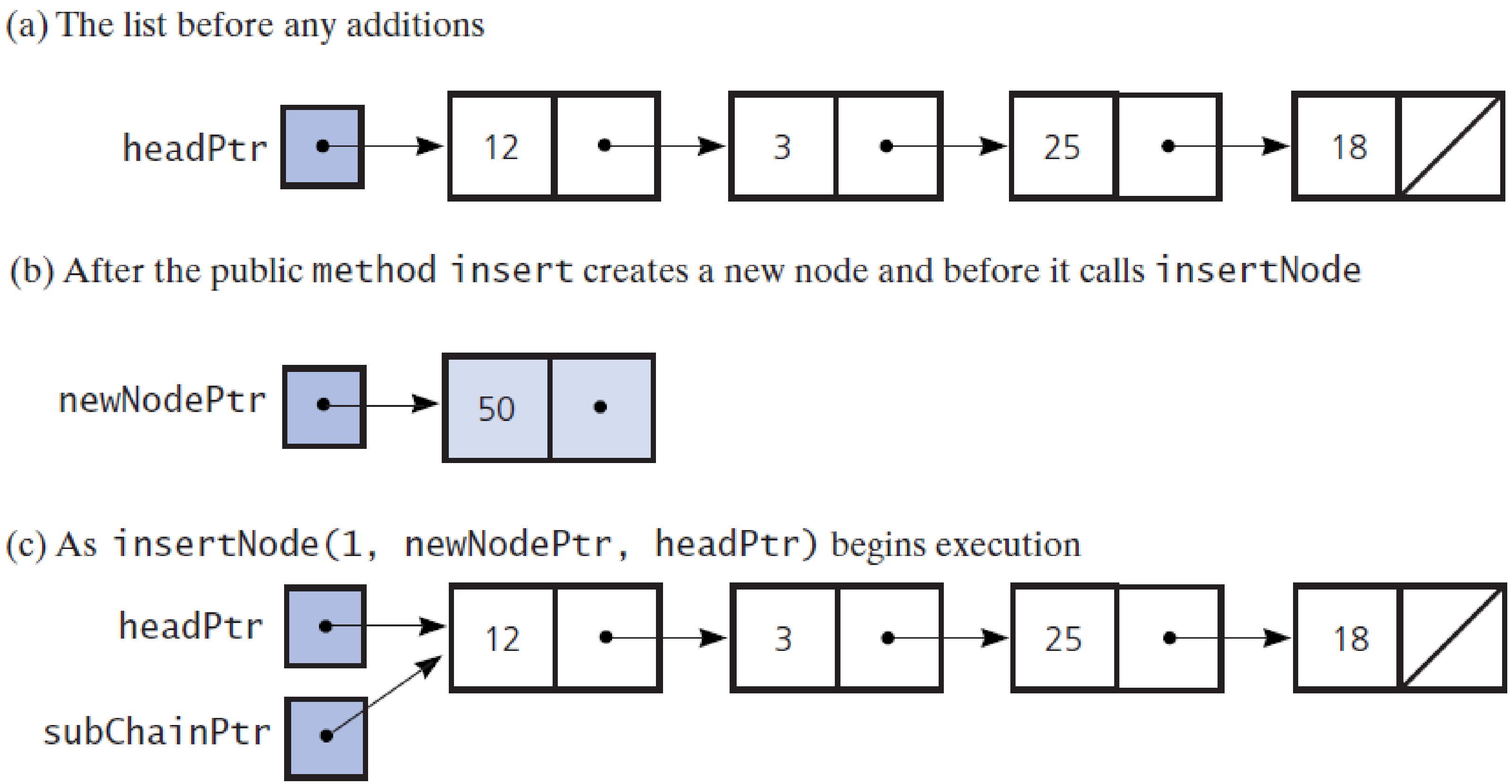
```
template<class ItemType>
ItemType LinkedList<ItemType>::~LinkedList()
{
     clear();
} // end destructor
```

Using Recursion in LinkedList Methods

- Possible to process a linked chain by
  - Processing its first node and
  - Then the rest of the chain recursively
- Logic used to add a node

```
if (the insertion position is 1)
          Add the new node to the beginning of the chain
else
          Ignore the first node and add the new node to the rest of the chain
```

Using Recursion in LinkedList Methods

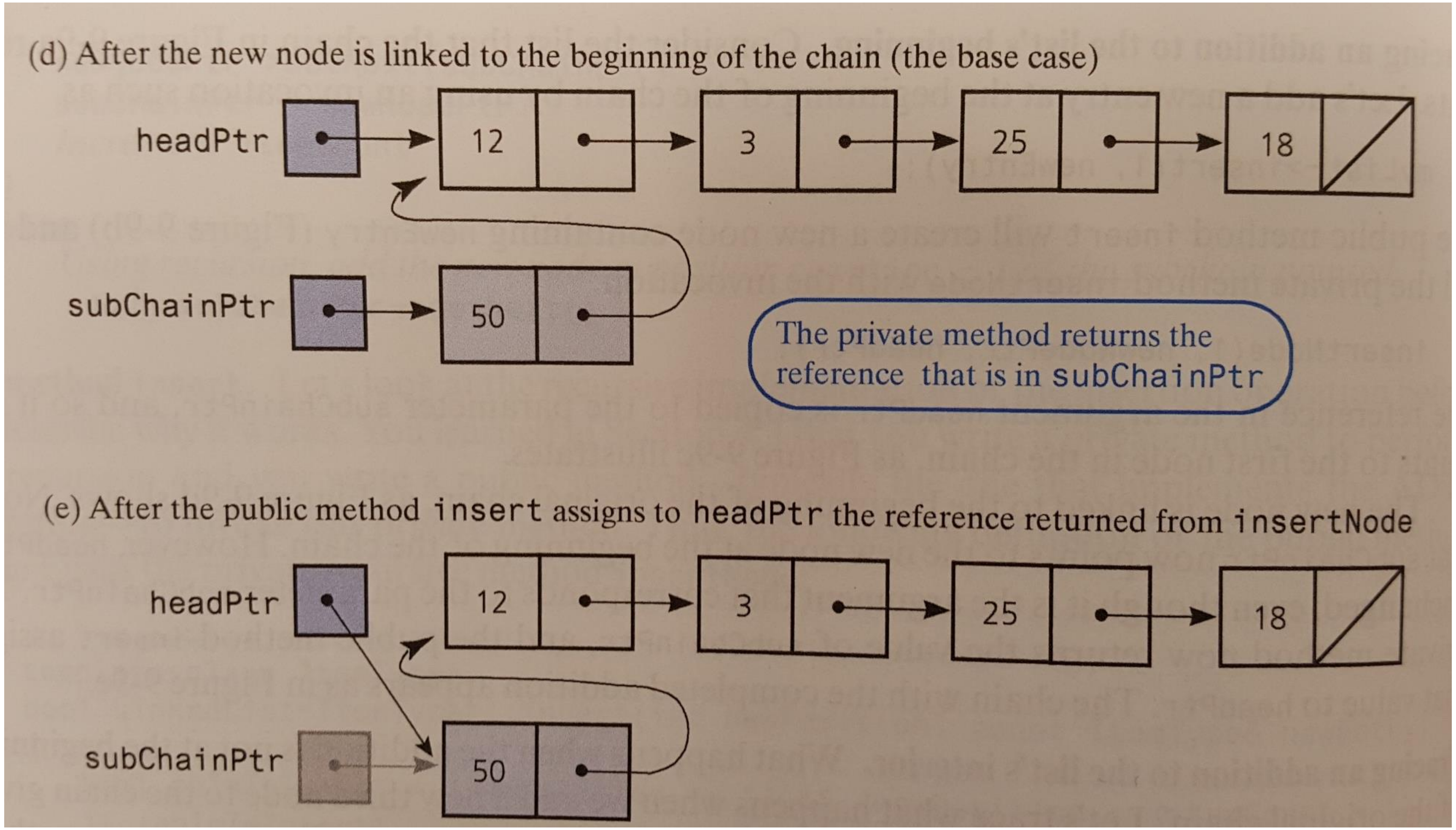


(a) The list before any additions

(b) After the public method insert creates a new node and before it calls insertNode

(c) As insertNode(1, newNodePtr, headPtr) begins execution

Using Recursion in LinkedList Methods



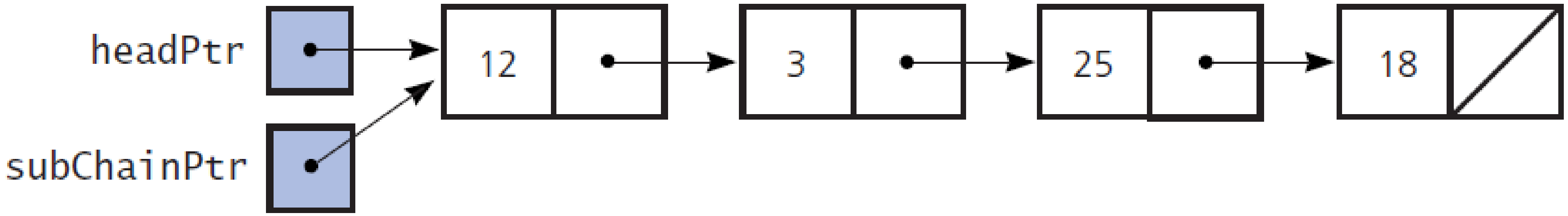
(d) After the new node is linked to the beginning of the chain (the base case)

The private method returns the reference that is in subChainPtr

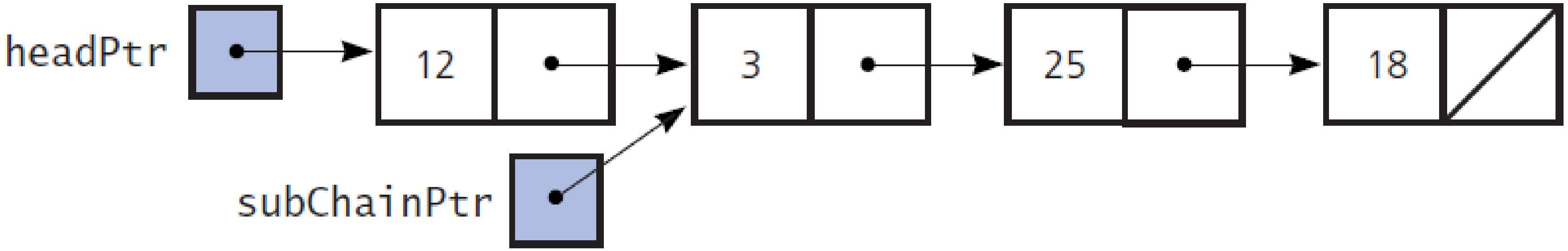
(e) After the public method insert assigns to headPtr the reference returned from insertNode

Using Recursion in LinkedList Methods

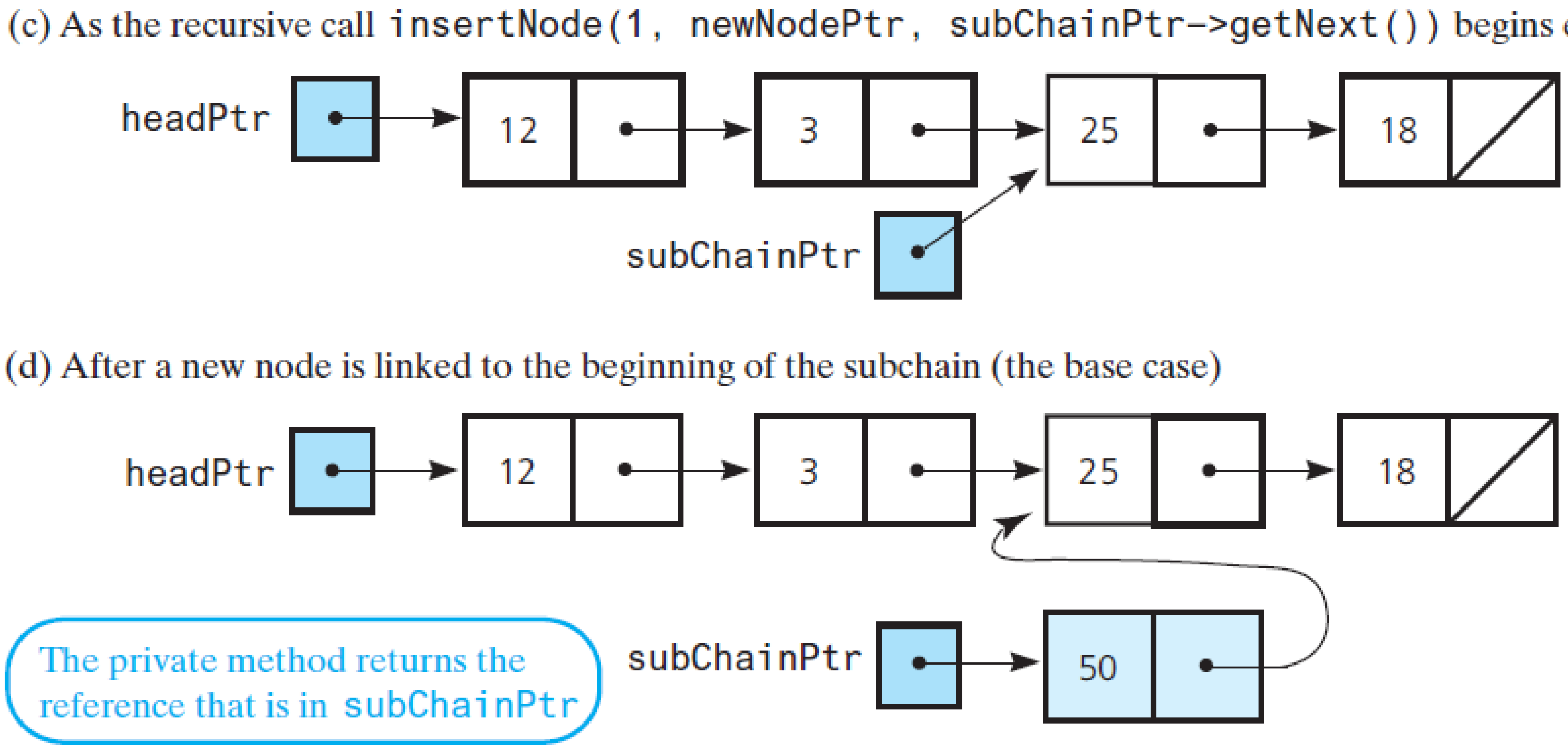
(a) As `insertNode(3, newNodePtr, headPtr)` begins execution



(b) As the recursive call `insertNode(2, newNodePtr, subChainPtr->getNext())` begins execution



Using Recursion in LinkedList Methods



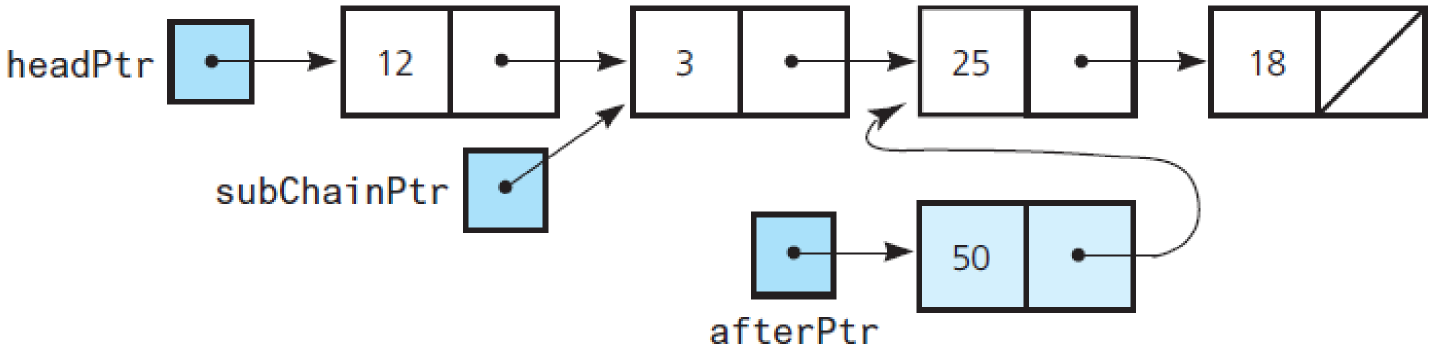
(c) As the recursive call `insertNode(1, newNodePtr, subChainPtr->getNext())` begins execution

(d) After a new node is linked to the beginning of the subchain (the base case)

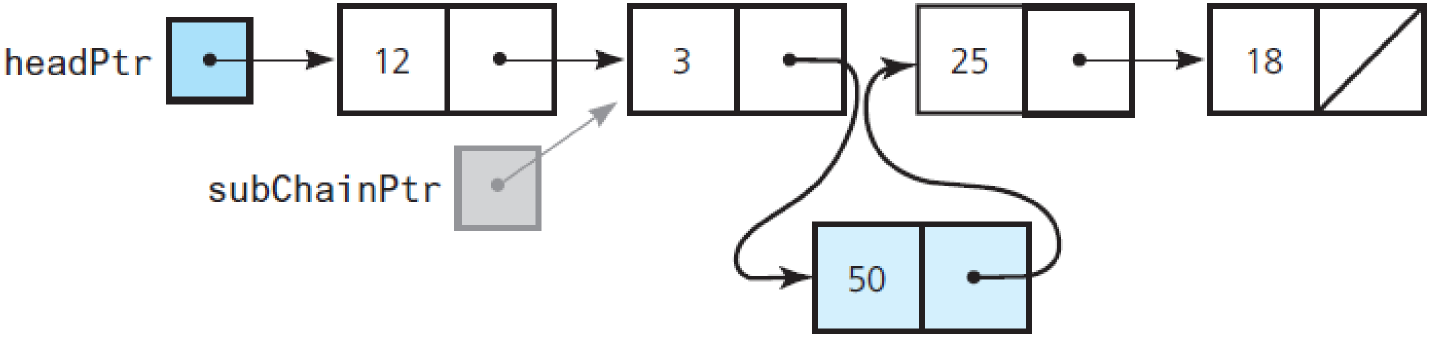
The private method returns the reference that is in `subChainPtr`

Using Recursion in LinkedList Methods

(e) After the returned reference is assigned to `afterPtr`



(f) After `subChainPtr->setNext(afterPtr)` executes



Comparing Implementations

- Time to access the i-th node in a chain of linked nodes depends on I
- You can access array items directly with equal access time
- Insertions and removals with link-based implementation
  - Do not require shifting data
  - Do require a traversal

Comparing Implementations

- Time to access the i-th node in a chain of linked nodes depends on I
- You can access array items directly with equal access time
- Insertions and removals with link-based implementation
  - Do not require shifting data
  - **Do require a traversal**

Let’s think of an alternative

- Data Fields
  - **`headPtr`**
    - Reference to the first node in the list
  - **`tailPtr`**
    - Reference to the last node in the list (efficiency reasons)
  - **`itemCount`**
    - Number of entries in the list

Thank you