List Implementations

Chapter 9

Array-Based Implementation

of the ADT List

```
+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
```

List operations in their UML form

Array-Based Implementation of the 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
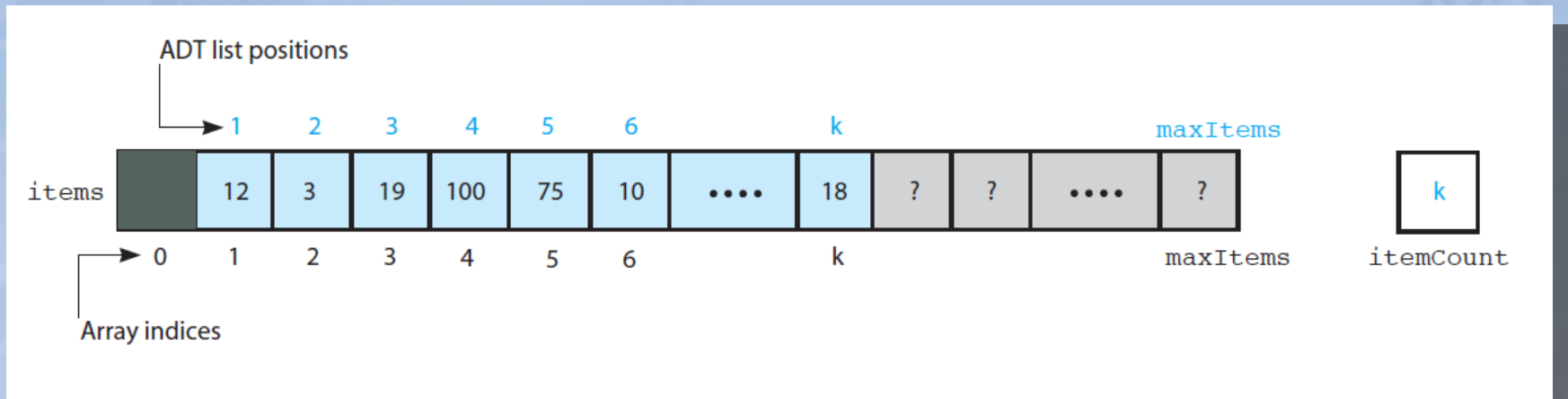 - Must keep track of number of entries

The Header File

FIGURE 9-1 An array-based implementation of the ADT list

The Header File

```
1    /** ADT list: Array-based implementation.
2     @file ArrayList.h */
3
4    #ifndef ARRAY_LIST_
5    #define ARRAY_LIST_
6
7    #include "ListInterface.h"
8    #include "PrecondViolatedExcept.h"
9
10   template<class ItemType>
11   class ArrayList : public ListInterface<ItemType>
12   {
13   private:
14      static const int DEFAULT_CAPACITY = 100; // Default capacity of the list
15      ItemType items[DEFAULT_CAPACITY + 1];      // Array of list items (ignore items[0])
16      int itemCount;                             // Current count of list items
17      int maxItems;                              // Maximum capacity of the list
18
```

LISTING 9-1 The header file for the class ArrayList

The Header File

```
18
19  public:
20     ArrayList();
21     // Copy constructor and destructor are supplied by compiler
22
23     bool isEmpty() const;
24     int getLength() const;
25     bool insert(int newPosition, const ItemType& newEntry);
26     bool remove(int position);
27     void clear();
28
```

LISTING 9-1 The header file for the class ArrayList

The Header File

```cpp
29       /** @throw  PrecondViolatedExcept if position < 1 or position > getLength(). */
30      ItemType getEntry(int position) const throw(PrecondViolatedExcept);
31
32       /** @throw  PrecondViolatedExcept if position < 1 or position > getLength(). */
33      ItemType replace(int position, const ItemType& newEntry)
34                                      throw(PrecondViolatedExcept);
35  }; // end ArrayList
36
37  #include "ArrayList.cpp"
38  #endif
```

LISTING 9-1 The header file for the class ArrayList

The Implementation File

```cpp
template<class ItemType>
ArrayList<ItemType>::ArrayList() : itemCount(0), maxItems(DEFAULT_CAPACITY)
{
}   // end default constructor
```

```cpp
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
```

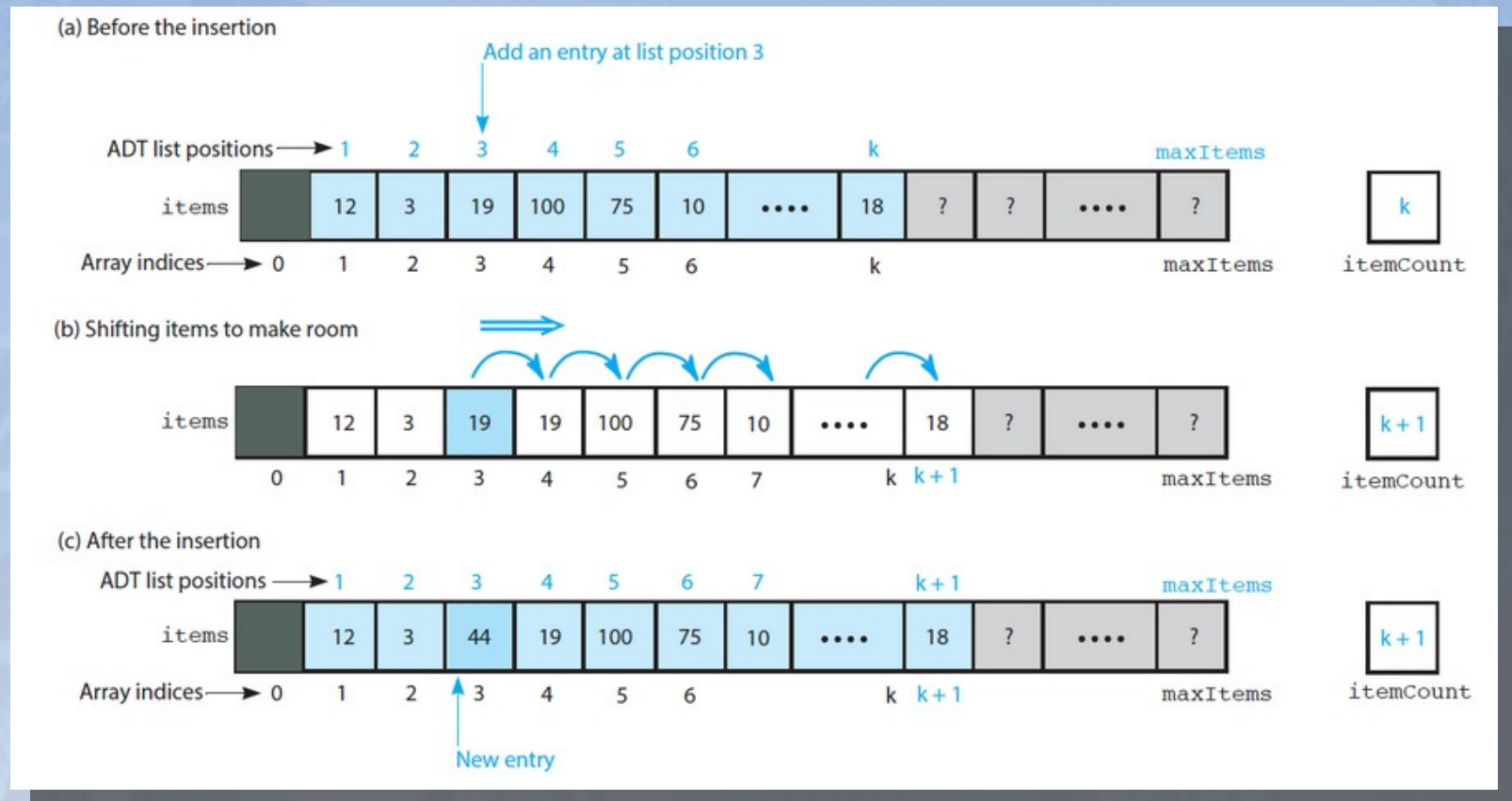Constructor, methods isEmpty and getLength

The Implementation File

```cpp
template<class ItemType>
ItemType ArrayList<ItemType>::getEntry(int position) 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
```

Method getEntry

The Implementation File

```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 at
        // positions from itemCount down to newPosition
        // (no shift if newPosition == itemCount + 1)
        for (int pos = itemCount; pos >= newPosition; pos--)
            items[pos + 1] = items[pos];

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

    return ableToInsert;
} // end insert
```

Method insert

The Implementation File

FIGURE 9-2 Shifting items for insertion

The Implementation File

```cpp
template<class ItemType>
ItemType ArrayList<ItemType>::getEntry(int position) 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
```

Method getEntry

The Implementation File

```cpp
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
```

Method replace

The Implementation File

```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
```
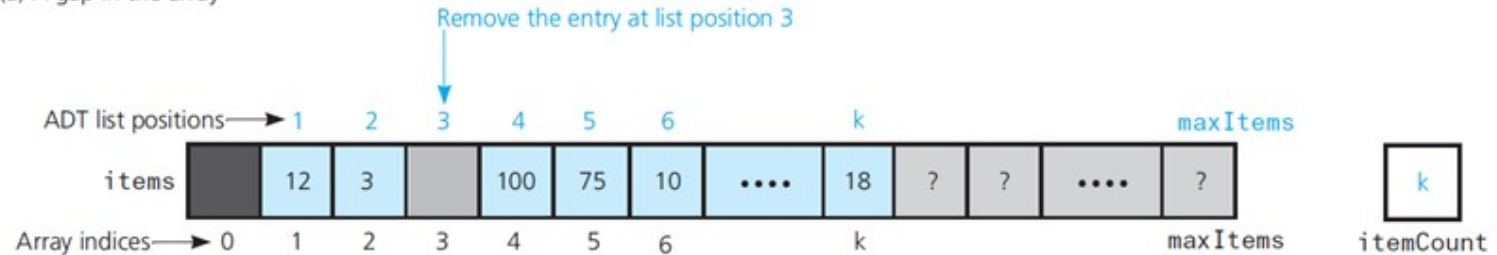
Method remove
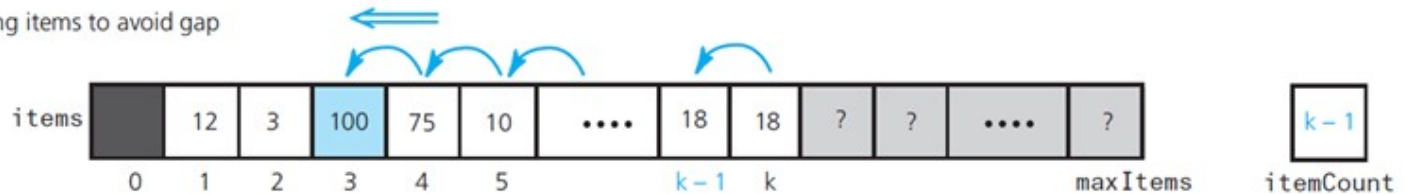
The Implementation File
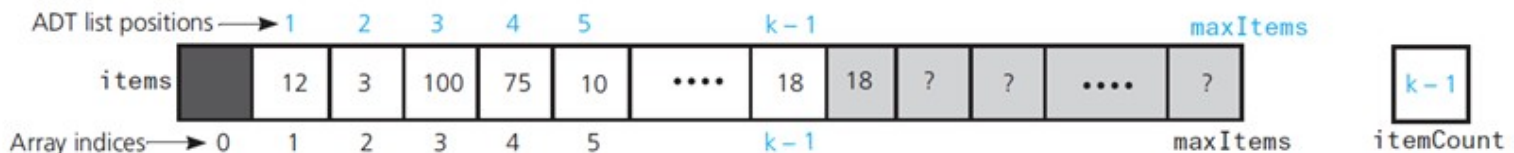
FIGURE 9-3 Shifting items to remove an entry

The Implementation File

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

Method clear

Link-Based Implementation of the ADT List

- We can use C++ pointers instead of an array to implement 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 the ADT List
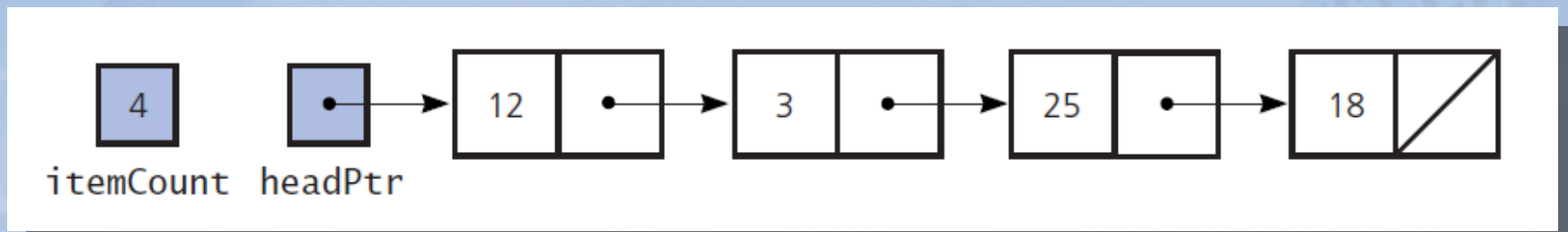
FIGURE 9-4 A link-based implementation of the ADT list

The Header File

```
1   /** ADT list: Link-based implementation.
2    @file LinkedList.h */
3
4   #ifndef LINKED_LIST_
5   #define LINKED_LIST_
6
7   #include "ListInterface.h"
8   #include "Node.h"
9   #include "PrecondViolatedExcept.h"
10
11  template<class ItemType>
12  class LinkedList : public ListInterface<ItemType>
13  {
14  private:
15     Node<ItemType>* headPtr; // Pointer to first node in the chain
16                              // (contains the first entry in the list)
17     int itemCount;          // Current count of list items
18     // Locates a specified node in a linked list.
```

LISTING 9-2 The header file for the class LinkedList

The Header File

```cpp
17      int itemCount;              // Current count of list items
18      // Locates a specified node in a linked list.
19      // @pre   position is the number of the desired node;
20      //        position >= 1 and position <= itemCount.
21      // @post  The node is found and a pointer to it is returned.
22      // @param position  The number of the node to locate.
23      // @return  A pointer to the node at the given position.
24      Node<ItemType>* getNodeAt(int position) const;
25
26   public:
27      LinkedList();
28      LinkedList(const LinkedList<ItemType>& aList);
29      virtual ~LinkedList();
30
31      bool isEmpty() const;
32      int getLength() const;
33      bool insert(int newPosition, const ItemType& newEntry);
34      bool remove(int position);
35      void clear();
```

LISTING 9-2 The header file for the class LinkedList

The Header File

```
34    bool remove(int position);
35    void clear();
36
37    /** @throw  PrecondViolatedExcept if position < 1 or
38                                      position > getLength(). */
39    ItemType getEntry(int position) const throw(PrecondViolatedExcept);
40
41    /** @throw  PrecondViolatedExcept if position < 1 or
42                                      position > getLength(). */
43    ItemType replace(int position, const ItemType& newEntry)
44                                   throw(PrecondViolatedExcept);
45  }; // end LinkedList
46
47  #include "LinkedList.cpp"
48  #endif
```

LISTING 9-2 The header file for the class LinkedList

The Implementation File

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

Constructor

The Implementation File

```cpp
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
```

Method getEntry

The Implementation File

```cpp
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
```

Method getNodeAt

The Implementation File

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

The Implementation File

```cpp
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 new node to chain
        if (newPosition == 1)
        {
            // Insert new node at beginning of chain
            newNodePtr->setNext(headPtr);
            headPtr = newNodePtr;
        }
        else
        {
```

Method insert

The Implementation File

```
        }
        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
```
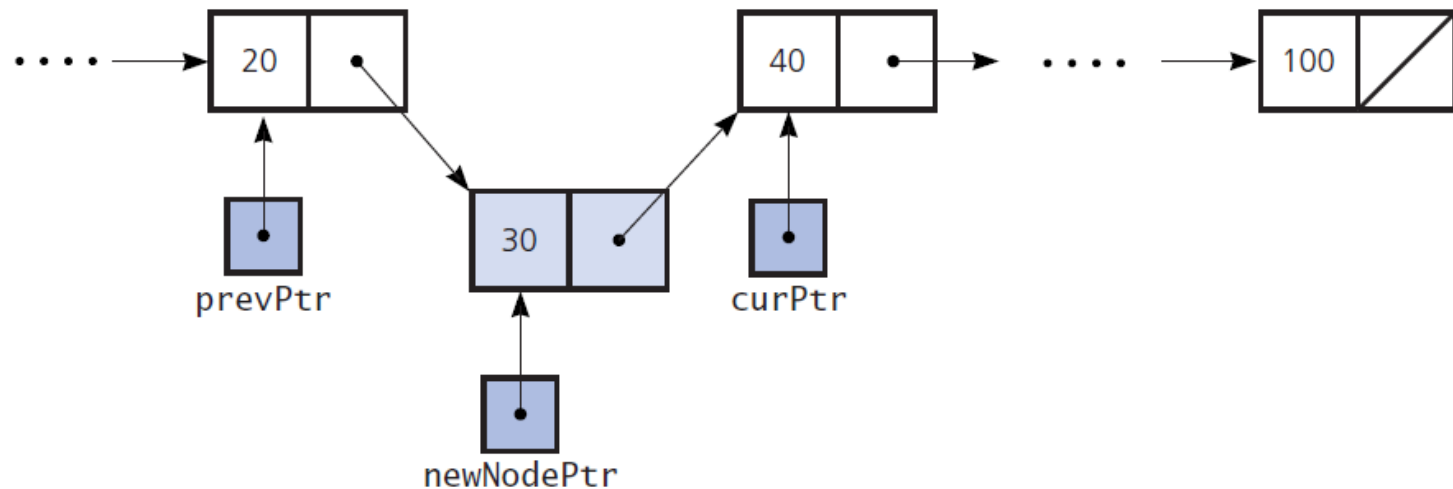
Method insert

The Implementation File

FIGURE 9-5 Inserting a new node between existing nodes of a linked chain

The Implementation File

FIGURE 9-5 Inserting a new node between existing nodes of a linked chain

The Implementation File

FIGURE 9-5 Inserting a new node between
existing nodes of a linked chain

The Implementation File
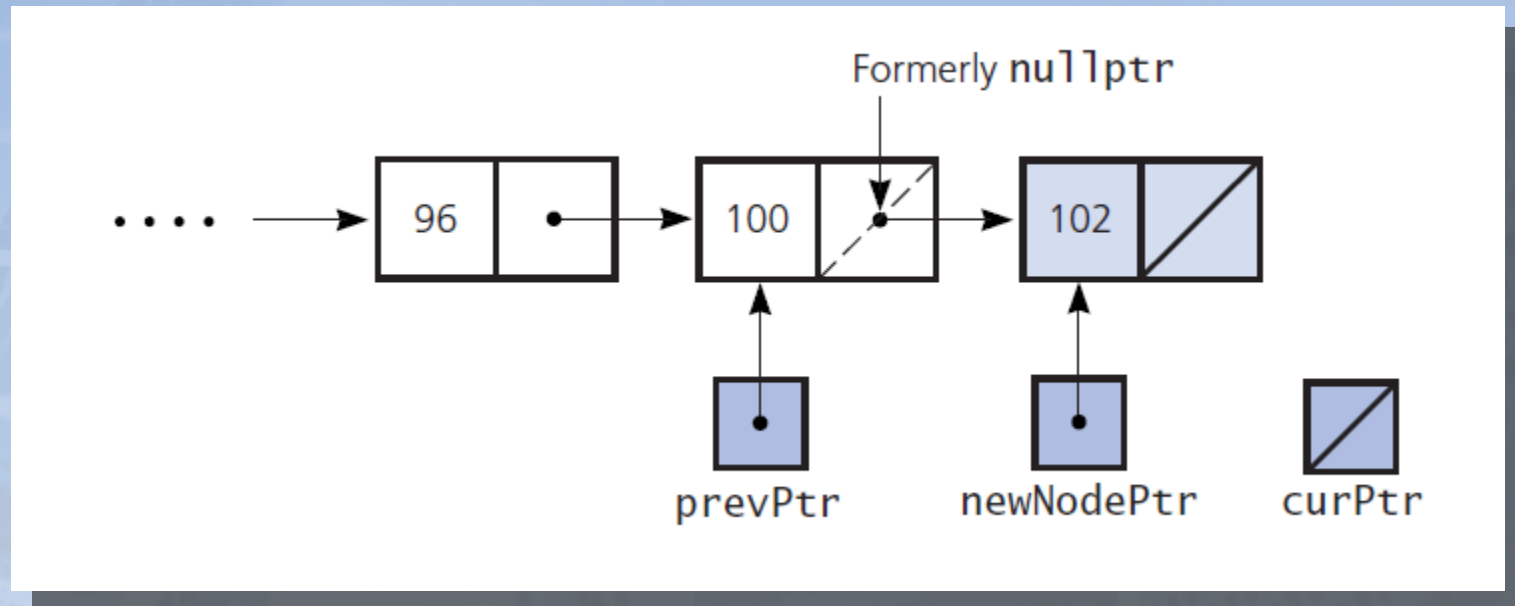
FIGURE 9-6 Inserting a new node at the end of a chain of linked nodes

The Implementation File
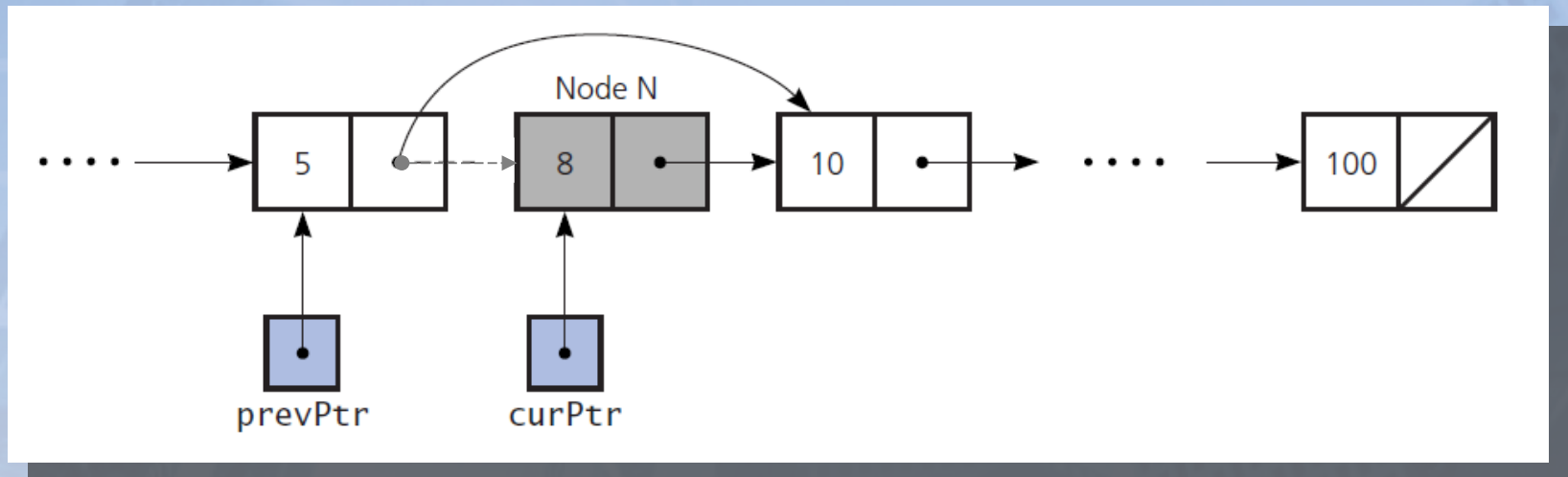
FIGURE 9-7 Removing a node from a chain

The Implementation File
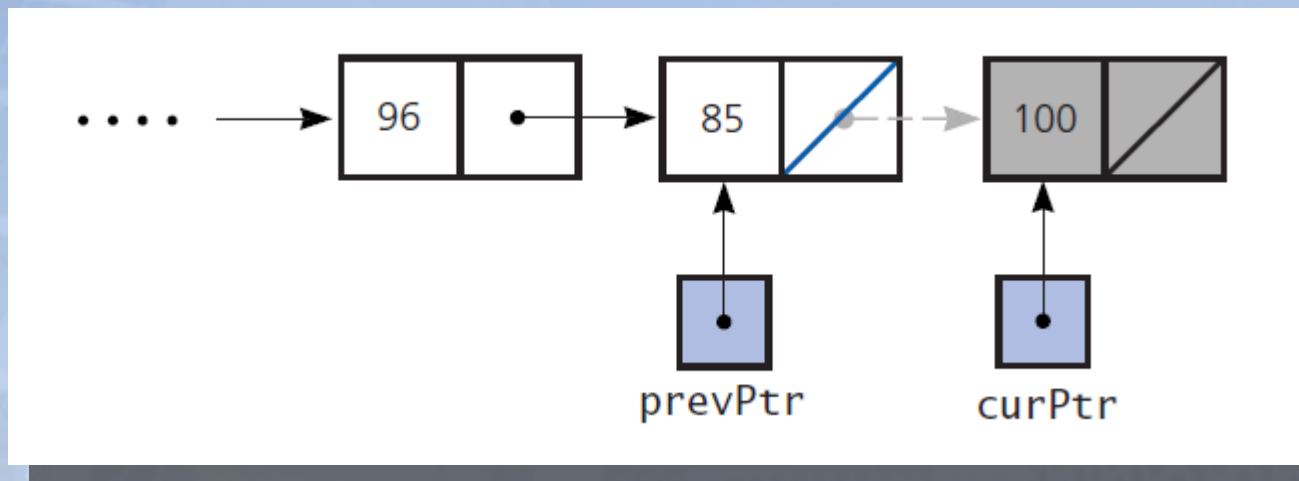
FIGURE 9-8 Removing the last node

The Implementation File

```cpp
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 the chain
            curPtr = headPtr; // Save pointer to node
            headPtr = headPtr->getNext();
        }
        else
        {
            // Find node that is before the one to remove
            Node<ItemType>* prevPtr = getNodeAt(position - 1);
```

Method remove

The Implementation File

```cpp
            // 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

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

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

    return ableToRemove;
}  // end remove
```

Method remove

The Implementation File

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

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

Method clear and the 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
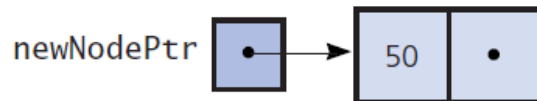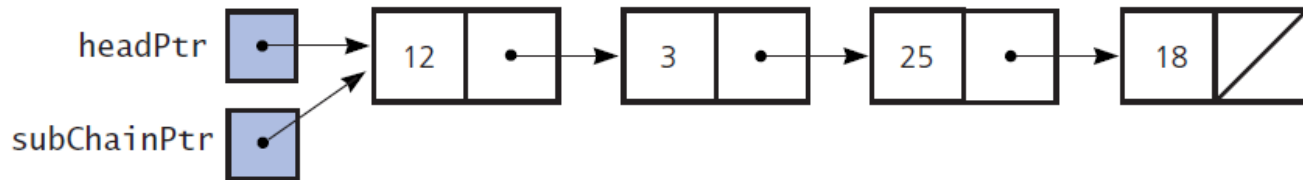
FIGURE 9-9 Recursively adding a node at the beginning of a chain

Using Recursion in LinkedList Methods

FIGURE 9-10 Recursively adding a node between existing nodes in a chain
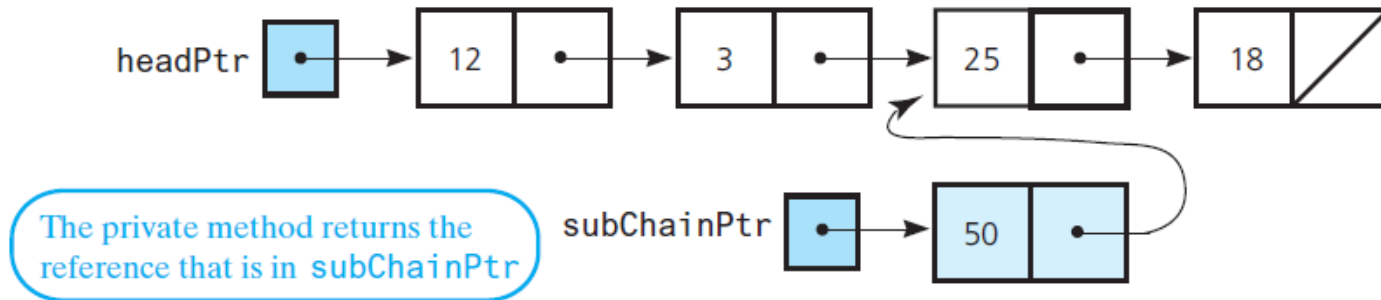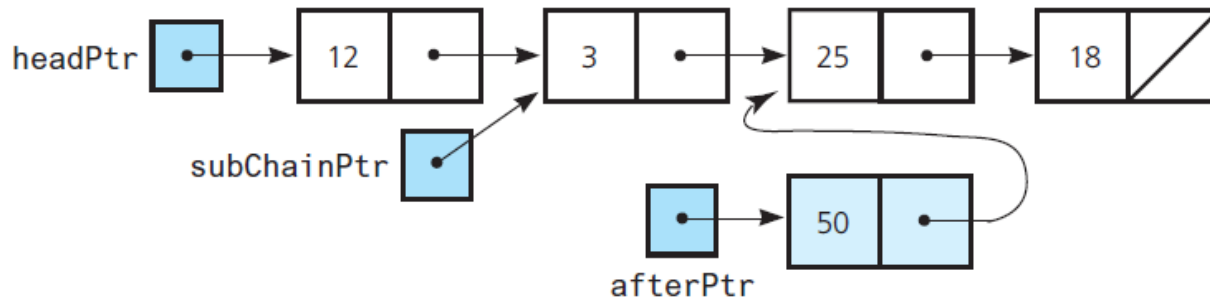
Using Recursion in LinkedList Methods

FIGURE 9-10 Recursively adding a node between existing nodes in a chain
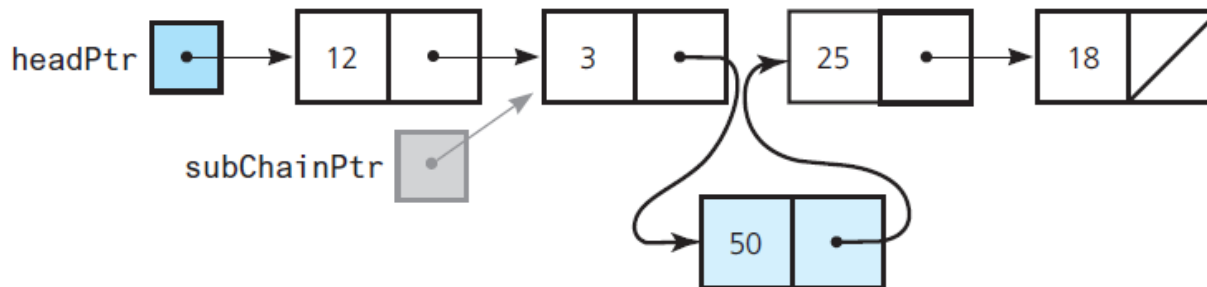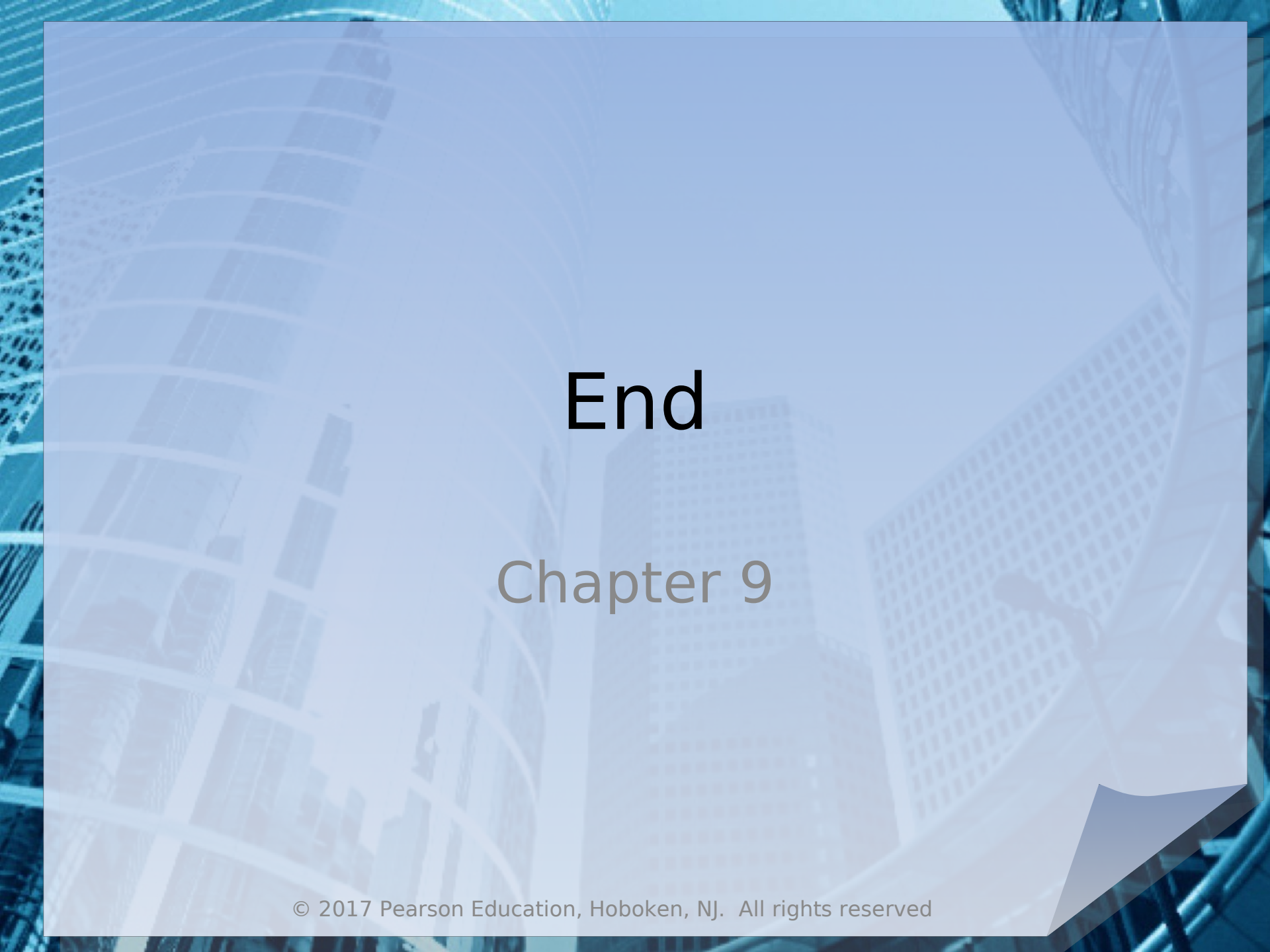
Using Recursion in LinkedList Methods

FIGURE 9-10 Recursively adding a node between existing nodes in a chain

Comparing Implementations

- Time to access the ith 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

End

Chapter 9