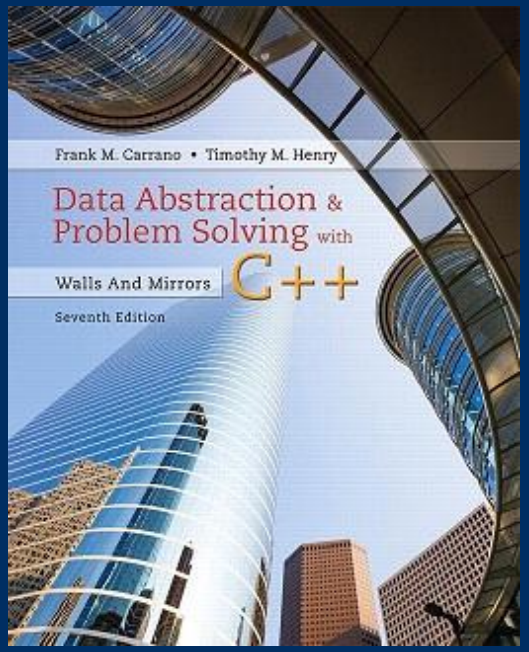
Chapter 8
Lists

CS 302 - Data Structures

M. Abdullah Canbaz



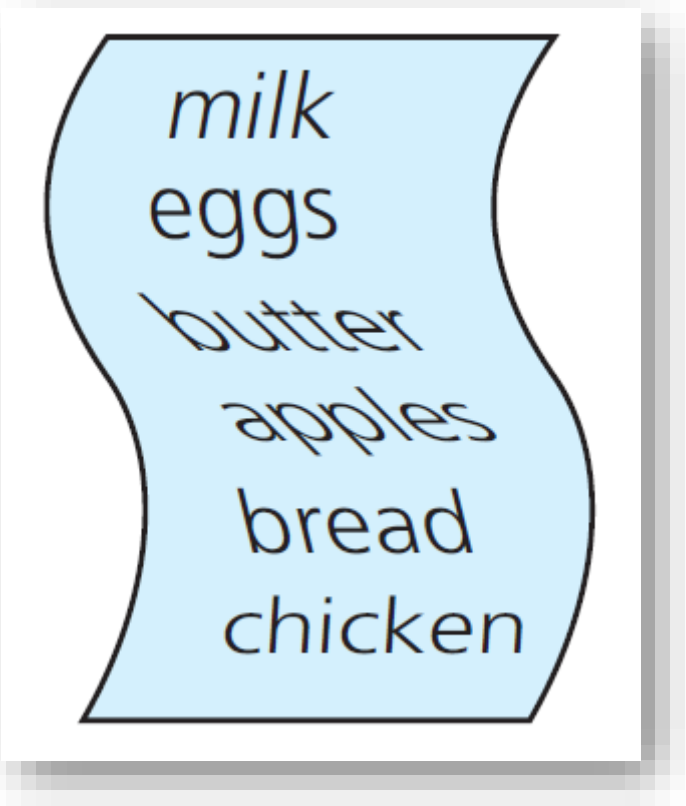
Modified from authors' slides

Specifying the ADT List

- Things you make lists of
  - Tasks
  - Addresses
  - Groceries
- Lists contain items of the same type
- Operations
  - Count items
  - Add, remove items
  - Retrieve

Specifying the ADT List

- A grocery list



ADT List Operations

- Test whether a list is empty.
- Get number of entries on a list.
- Insert entry at given position on list.
- Remove entry at given position from list.
- Remove all entries from list.
- Look at (get) entry at given position on list.
- Replace (set) entry at given position on list.

Specifying the ADT List

- UML diagram for the ADT list

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

Specifying the ADT List

- Definition: ADT List
  - Finite number of objects
  - Not necessarily distinct
  - Same data type
  - Ordered by position as determined by client

Axioms for ADT List

1. (List()).isEmpty() = true
2. (List()).getLength() = 0
3. aList.getLength() = (aList.insert(i, item)).getLength() - 1
4. aList.getLength() = (aList.remove(i)).getLength() + 1
5. (aList.insert(i, item)).isEmpty() = false
6. (List()).remove(i) = false
7. (aList.insert(i, item)).remove(i) = true
8. (aList.insert(i, item)).remove(i) = aList
9. (List()).getEntry(i) => error
10. (aList.insert(i, item)).getEntry(i) = item
11. aList.getEntry(i) = (aList.insert(i, item)).getEntry(i + 1)
12. aList.getEntry(i + 1) = (aList.remove(i)).getEntry(i)
13. (List()).replace(i, item) => error
14. (aList.replace(i, item)).getEntry(i) = item

Interface Template for ADT List

- A C++ interface for lists

```
1  /** Interface for the ADT list
2   @file ListInterface.h */
3
4  #ifndef LIST_INTERFACE_
5  #define LIST_INTERFACE_
6
7  template<class ItemType>
8  class ListInterface
9
10 {
11 public:
12    /** Sees whether this list is empty.
13     @return  True if the list is empty; otherwise returns false. */
14    virtual bool isEmpty() const = 0;
15
16    /** Gets the current number of entries in this list.
17     @return  The integer number of entries currently in the list. */
18    virtual int getLength() const = 0;
```

Interface Template for ADT List

- A C++ interface for lists

```
19
20      /** Inserts an entry into this list at a given position.
21       @pre  None.
22       @post  If 1 <= position <= getLength() + 1 and the insertion is
23          successful, newEntry is at the given position in the list,
24          other entries are renumbered accordingly, and the returned
25          value is true.
26       @param newPosition  The list position at which to insert newEntry.
27       @param newEntry  The entry to insert into the list.
28       @return  True if the insertion is successful, or false if not. */
29      virtual bool insert(int newPosition, const ItemType& newEntry) = 0;
30
31      /** Removes the entry at a given position from this list.
32       @pre  None.
33       @post  If 1 <= position <= getLength() and the removal is successful,
34          the entry at the given position in the list is removed, other
35          items are renumbered accordingly, and the returned value is true.
36       @param position  The list position of the entry to remove.
37       @return  True if the removal is successful, or false if not. */
38      virtual bool remove(int position) = 0;
39
```

Interface Template for ADT List

- A C++ interface for lists

```
39
40      /** Removes all entries from this list.
41       @post  The list contains no entries and the count of items is 0. */
42      virtual void clear() = 0;
43
44      /** Gets the entry at the given position in this list.
45       @pre  1 <= position <= getLength().
46       @post  The desired entry has been returned.
47       @param position  The list position of the desired entry.
48       @return  The entry at the given position. */
49      virtual ItemType getEntry(int position) const = 0;
50
```

Interface Template for ADT List

- A C++ interface for lists

```
50
51      /** Replaces the entry at the given position in this list.
52       @pre  1 <= position <= getLength().
53       @post  The entry at the given position is newEntry.
54       @param position  The list position of the entry to replace.
55       @param newEntry  The replacement entry.
56       @return  The replaced entry. */
57      virtual ItemType replace(int position, const ItemType& newEntry) = 0;
58
59      /** Destroys this list and frees its assigned memory. */
60      virtual ~ListInterface() { }
61  }; // end ListInterface
62  #endif
```

Using the List Operations

- Displaying the items on a list.

```
// Displays the items on the list aList.
displayList(aList)
{
   for (position = 1 through aList.getLength())
   {
      dataItem = aList.getEntry(position)
      Display dataItem
   }
}
```

Using the List Operations

- Replacing an item.

```
// Replaces the ith entry in the list aList with newEntry.
// Returns true if the replacement was successful; otherwise return false.
replace(aList, i, newEntry)
{
   success = aList.remove(i)
   if (success)
      success = aList.insert(i, newEntry)

   return success
}
```

Sorted and Unsorted Lists

UNSORTED LIST

Elements are placed into the list in no particular order

SORTED LIST

List elements are in an order that is sorted in some way

- **either numerically,**
- **alphabetically by the elements themselves, or by a component of the element**
  - **called a KEY member**

Complexities

- the order of the operation that determines if an item is in
  - a list in a sorted, array-based implementation — O(log N)
  - a list in an unsorted, array-based implementation — O(N)
  - a list in a sorted, linked implementation — O(N)
  - a list in an unsorted, linked implementation — O(N)

Allocation of memory

STATIC ALLOCATION

Static allocation is the allocation of memory space at compile time

DYNAMIC ALLOCATION

Dynamic allocation is the allocation of memory space at run time by using operator new

3 Kinds of Program Data

- **STATIC DATA: memory allocation exists throughout execution of program**

```
static long SeedValue;
```

- **AUTOMATIC DATA: automatically created at function entry, resides in activation frame of the function, and is destroyed when returning from function**

- **DYNAMIC DATA: explicitly allocated and deallocated during program execution by C++ instructions written by programmer using unary operators `new` and `delete`**

Why is a destructor needed?

- **When a local list variable goes out of scope, the memory space for data member `pointer` is deallocated**

- **But the nodes to which `pointer` points are not deallocated**

- **A class destructor is used to deallocate the dynamic memory pointed to by the data member**

Chapter 9

List Implementations

Array-Based Implementation of the ADT List

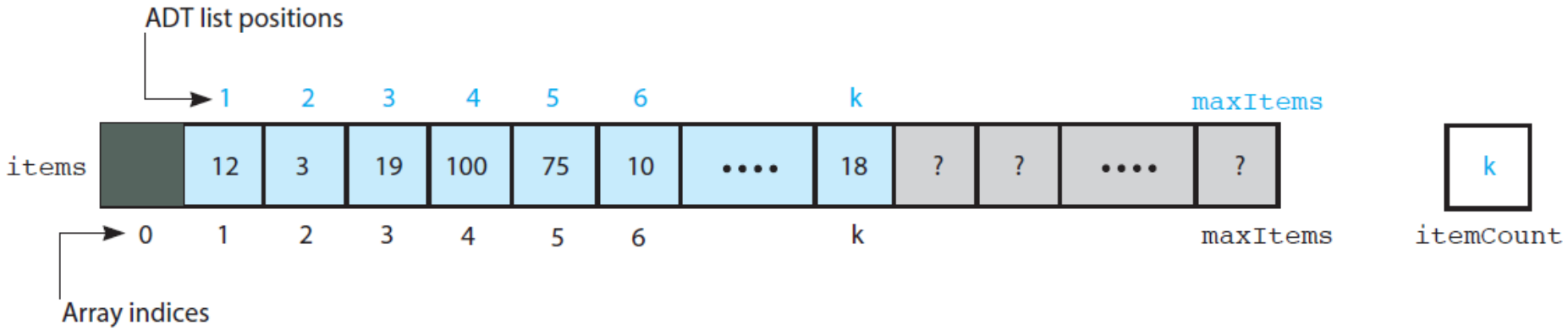
```
+isEmpty(): boolean
+getLength(): integer
+insert(newPosition: integer, newEntry: ItemType): boolean
+remove(position: integer): boolean
+clear(): void
+getEntry(position: integer): ItemType
+replace(position: integer, newEntry: ItemType): ItemType
```

- 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



- 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
```

- 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
```

- The header file for the class ArrayList

The Header File

```
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
```

- The header file for the class ArrayList

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

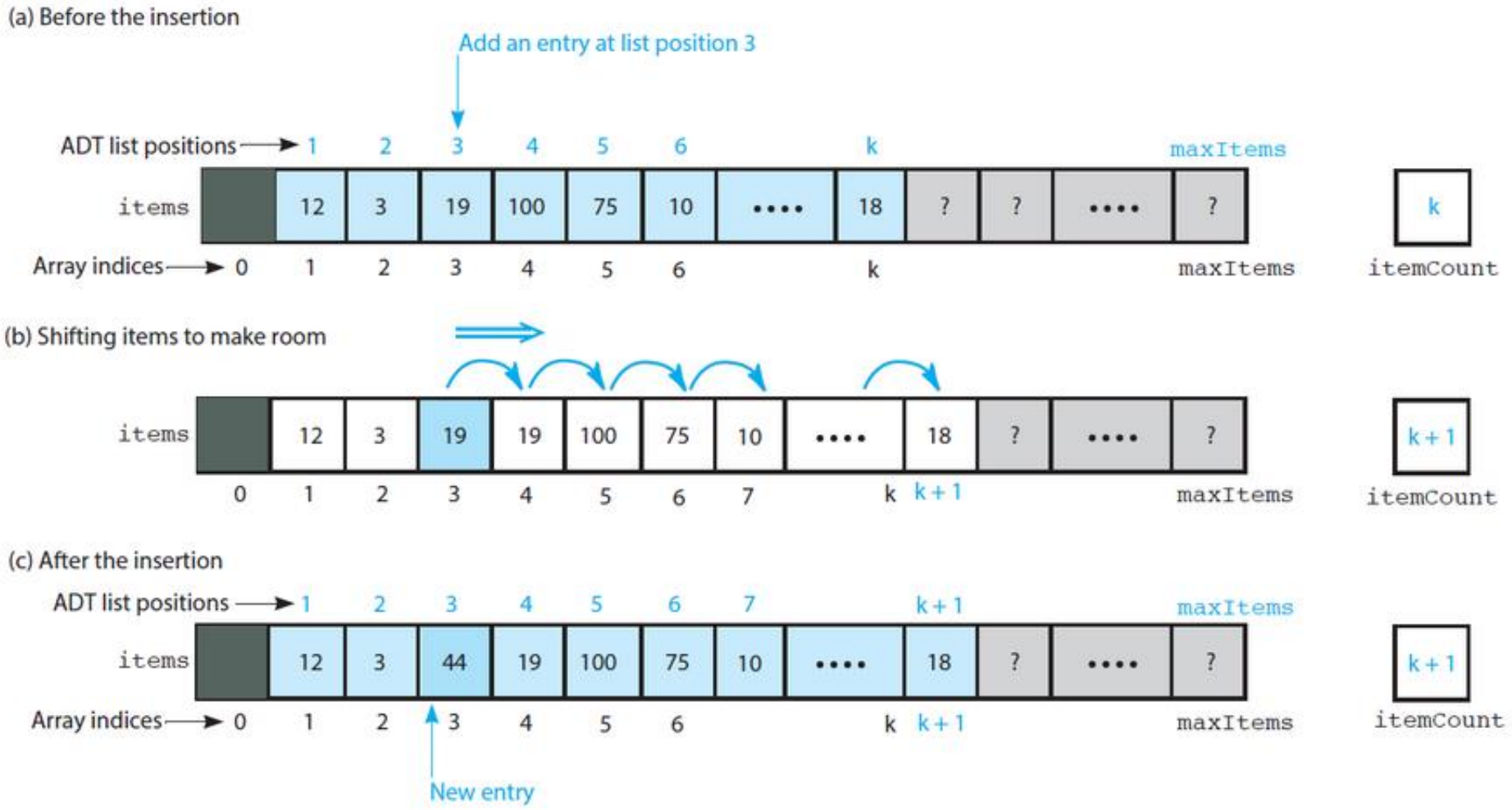
```
template<class ItemType>
ItemType ArrayList<ItemType>::getEntry(int position) const
                                throw(PrecondViolatedExcept)
{
   // Enforce precondition
   bool ableToGet = (position >= 1) && (position <= itemCount);
   if (ableToGet)
      return items[position];
   else
  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

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 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
```

The Implementation File



Shifting items for insertion

The Implementation File

- Method getEntry

```
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
```

The Implementation File

- Method replace

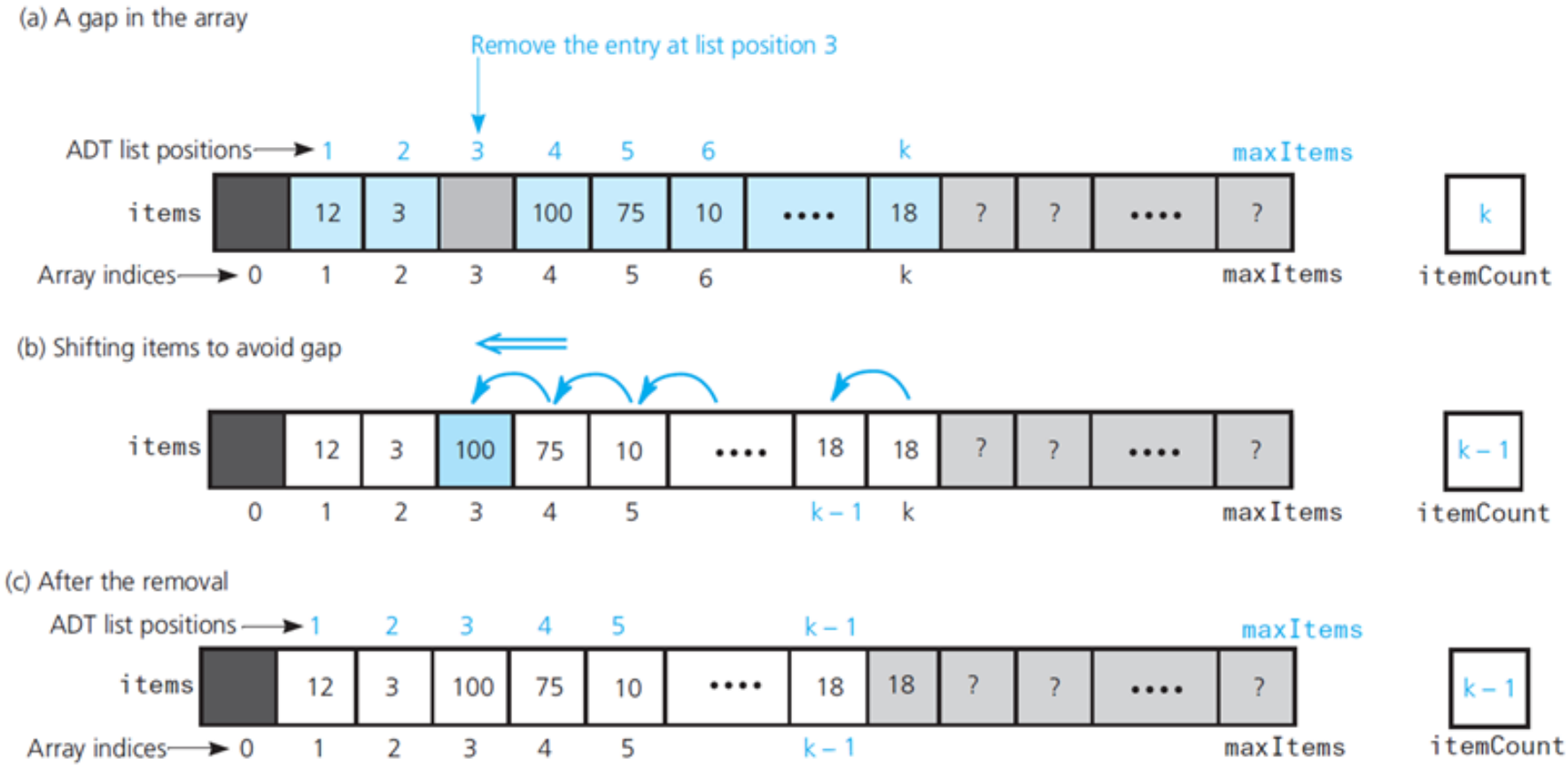
```
template<class ItemType>
ItemType ArrayList<ItemType>::replace(int position, const ItemType& newEntry)
                              throw(PrecondViolatedExcept)
{
   // Enforce precondition
   bool ableToSet = (position >= 1) && (position <= itemCount);
   if (ableToSet)
   {
      ItemType oldEntry = 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

- 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



Shifting items to remove an entry

The Implementation File

- Method clear

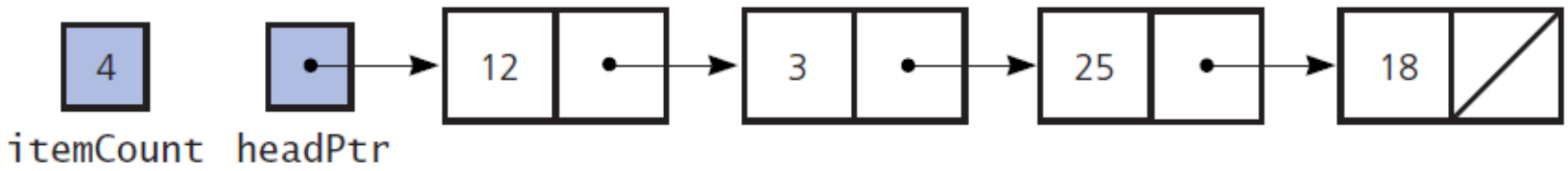
```
template<class ItemType>
void ArrayList<ItemType>::clear()
{
   itemCount = 0;
}  // end clear
```

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

- 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.
```

- The header file for the class LinkedList

The Header File

```
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();
```

- 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
```

- The header file for the class LinkedList

The Implementation File

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

- Constructor

The Implementation File

```
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

- 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

- 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

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

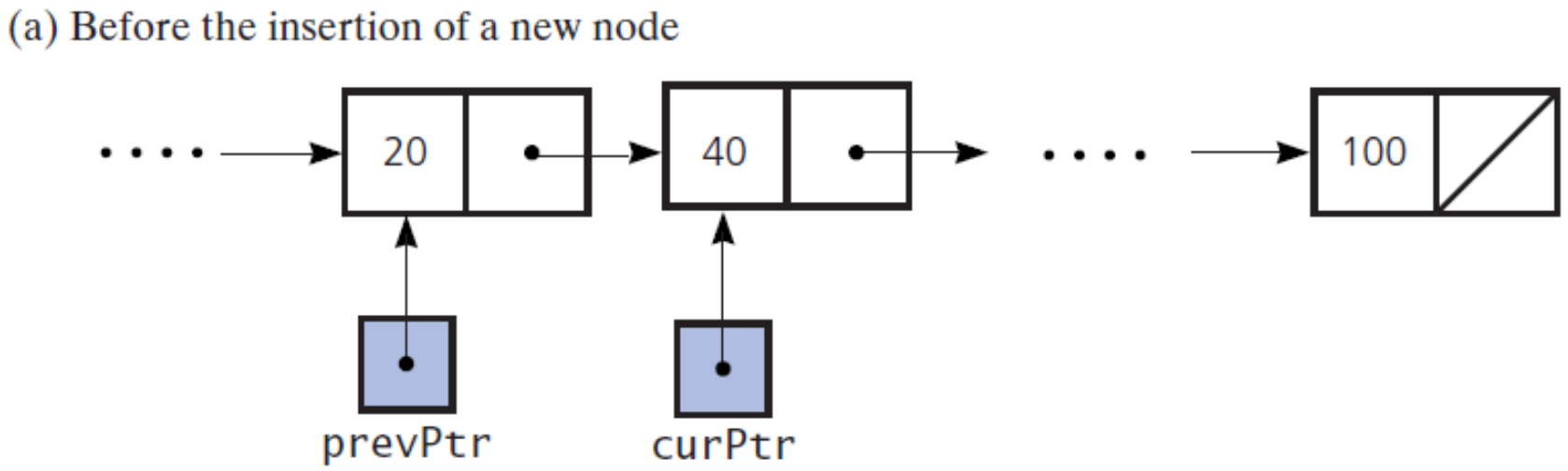
The Implementation File

- 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
```

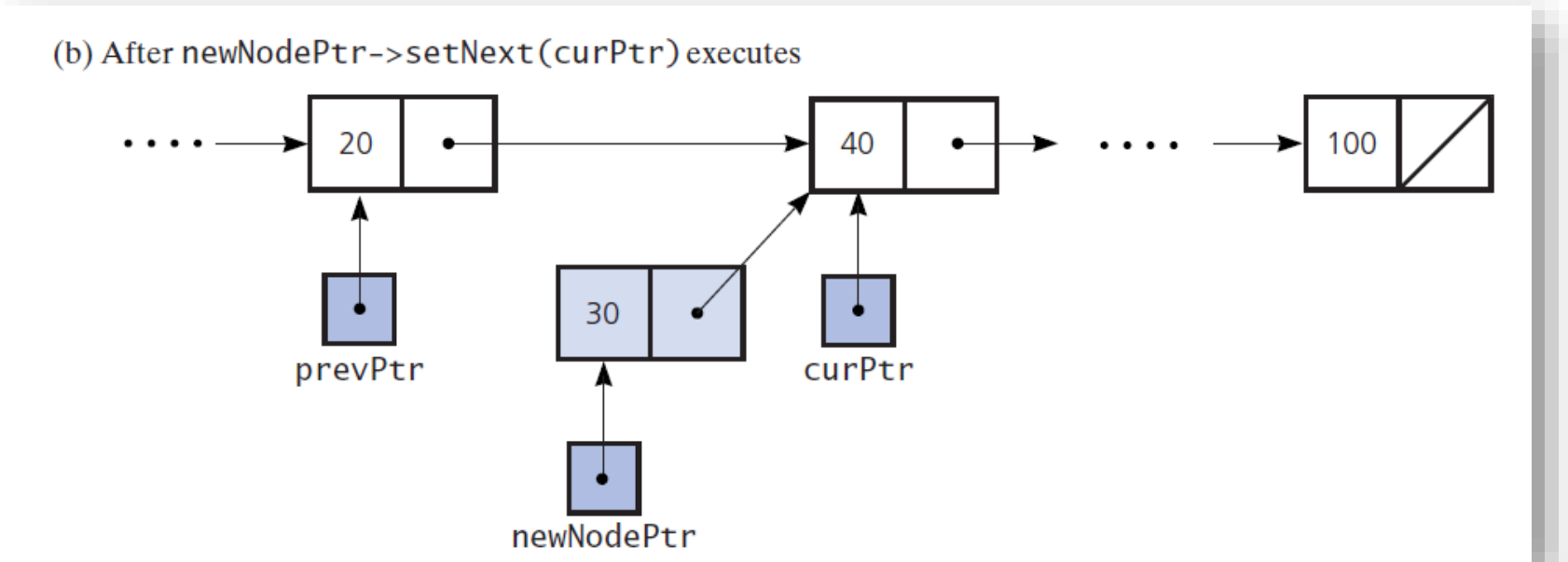
The Implementation File

- Inserting a new node between existing nodes of a linked chain



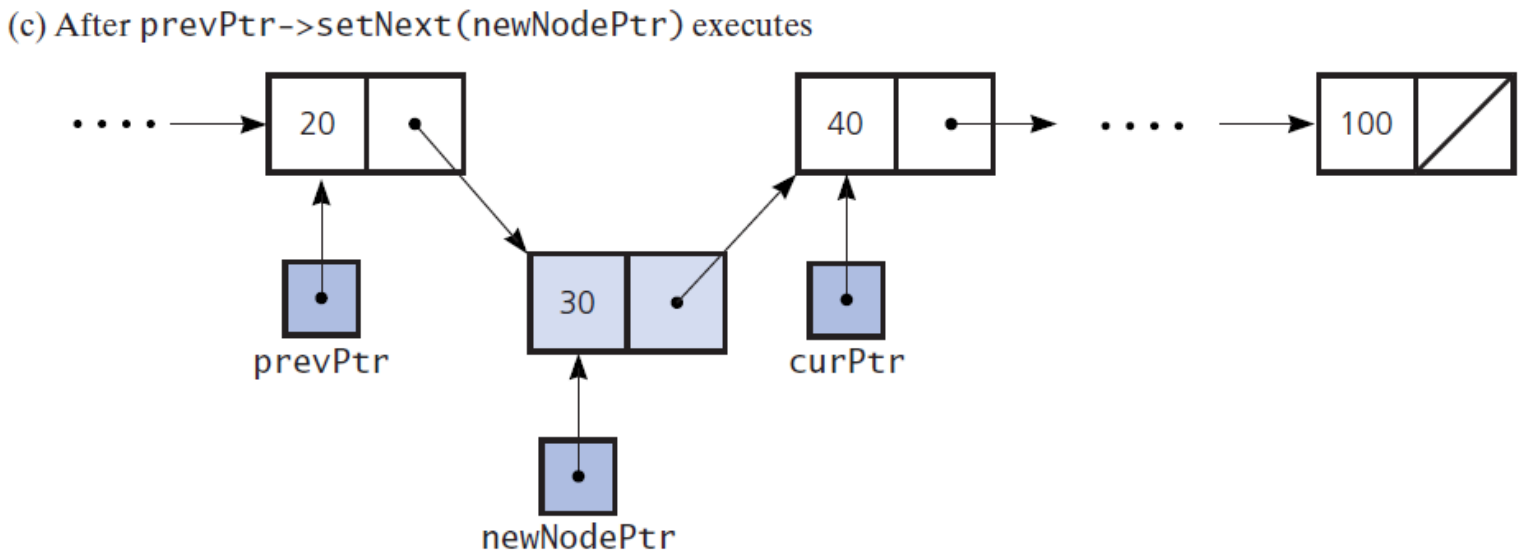
The Implementation File

- Inserting a new node between existing nodes of a linked chain



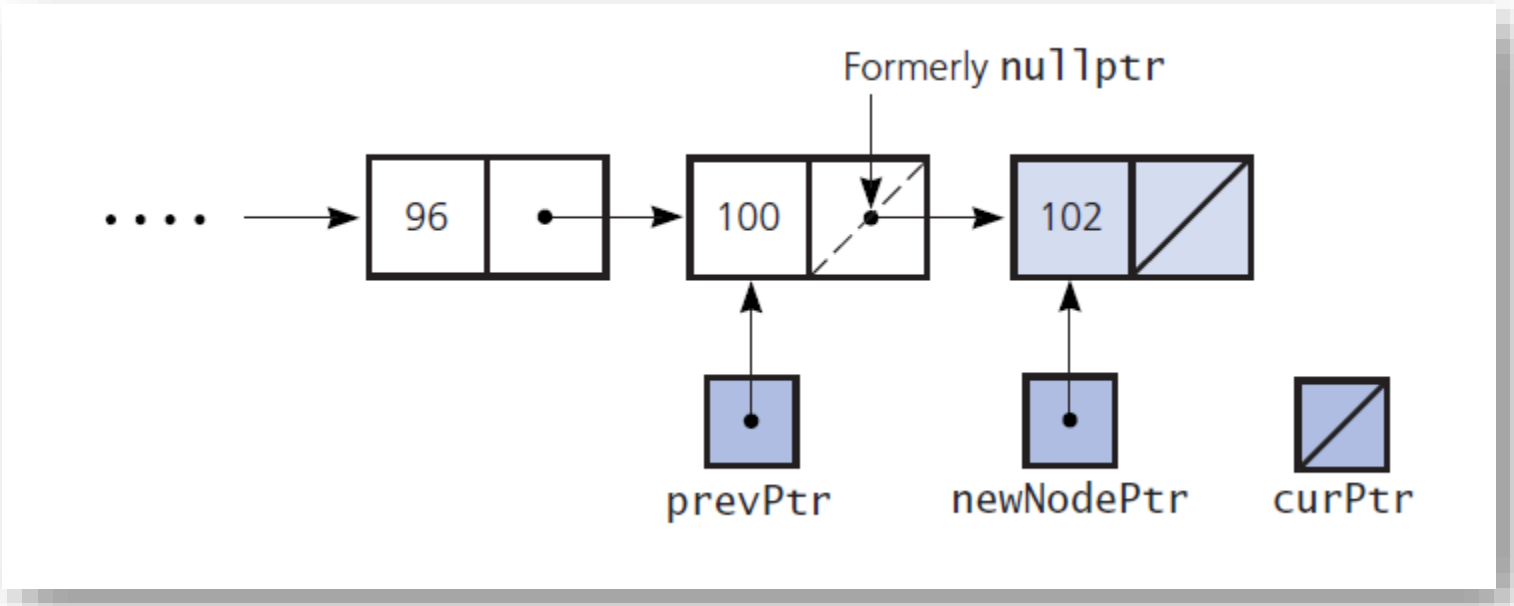
The Implementation File

- Inserting a new node between existing nodes of a linked chain



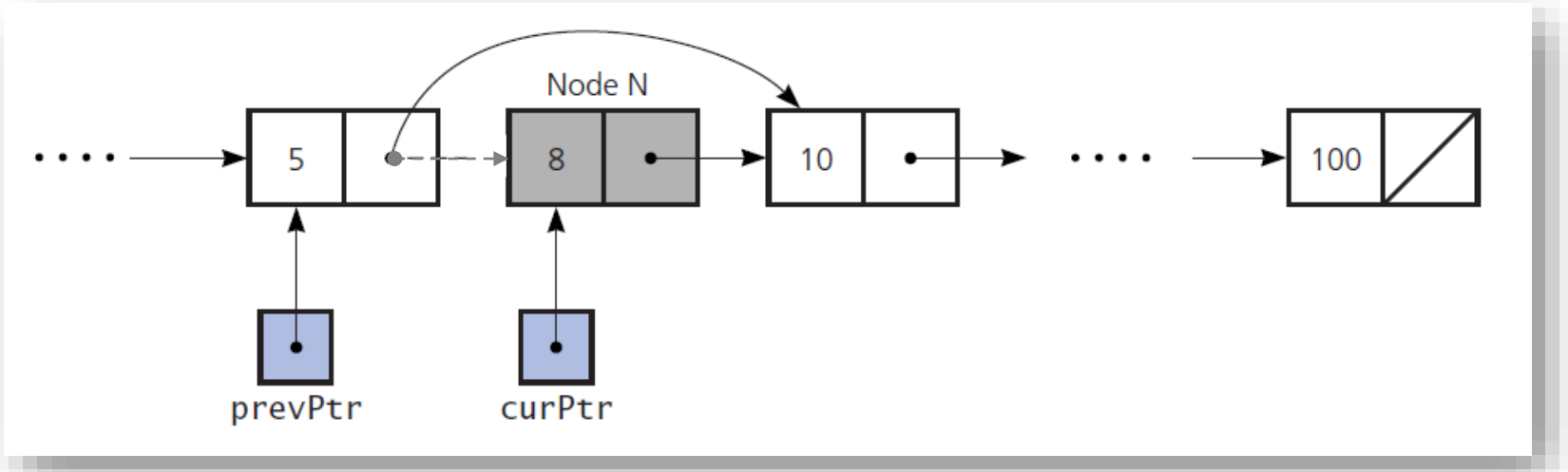
The Implementation File

- Inserting a new node at the end of a chain of linked nodes



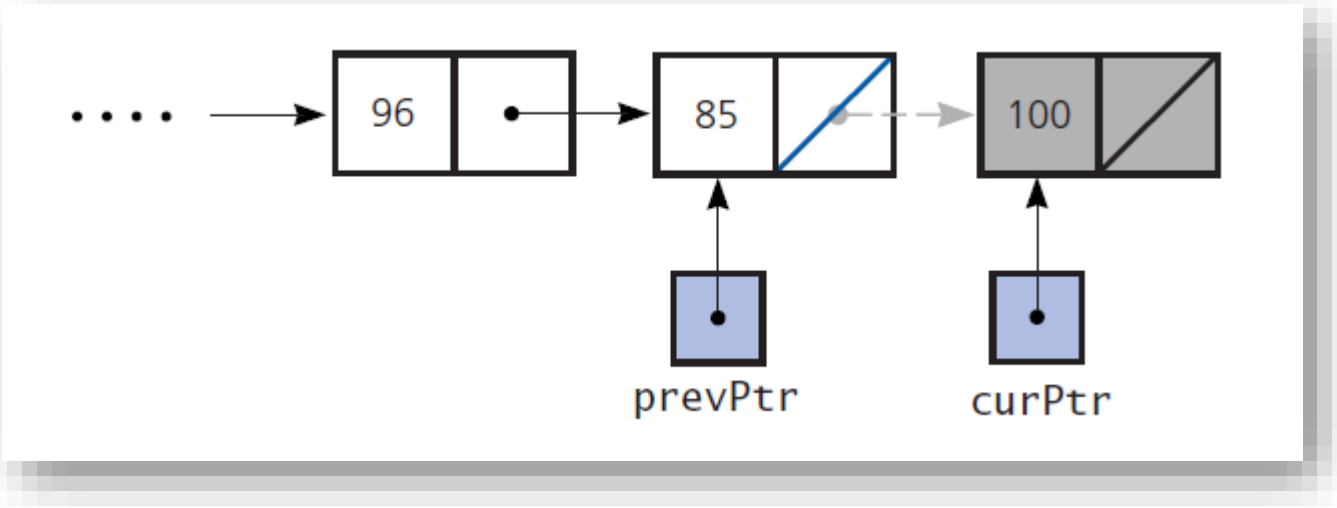
The Implementation File

- Removing a node from a chain



The Implementation File

- Removing the last node



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 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);
```

The Implementation File

- Method remove

```
        // 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
```

The Implementation File

- Method clear and the destructor

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

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

Using Recursion in LinkedList Methods

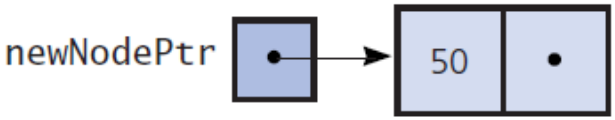
- Possible to process a linked chain by
  - Processing its first node and
  - Then the rest of the chain recursively
- Logic used to add a node

```
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
```

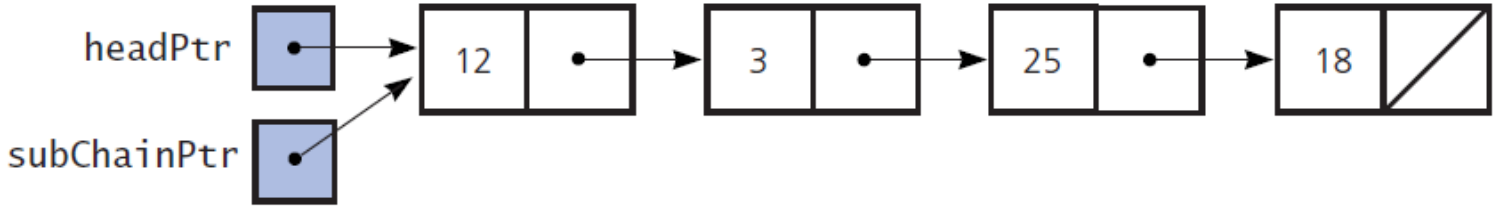
Using Recursion in LinkedList Methods

(a) The list before any additions

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

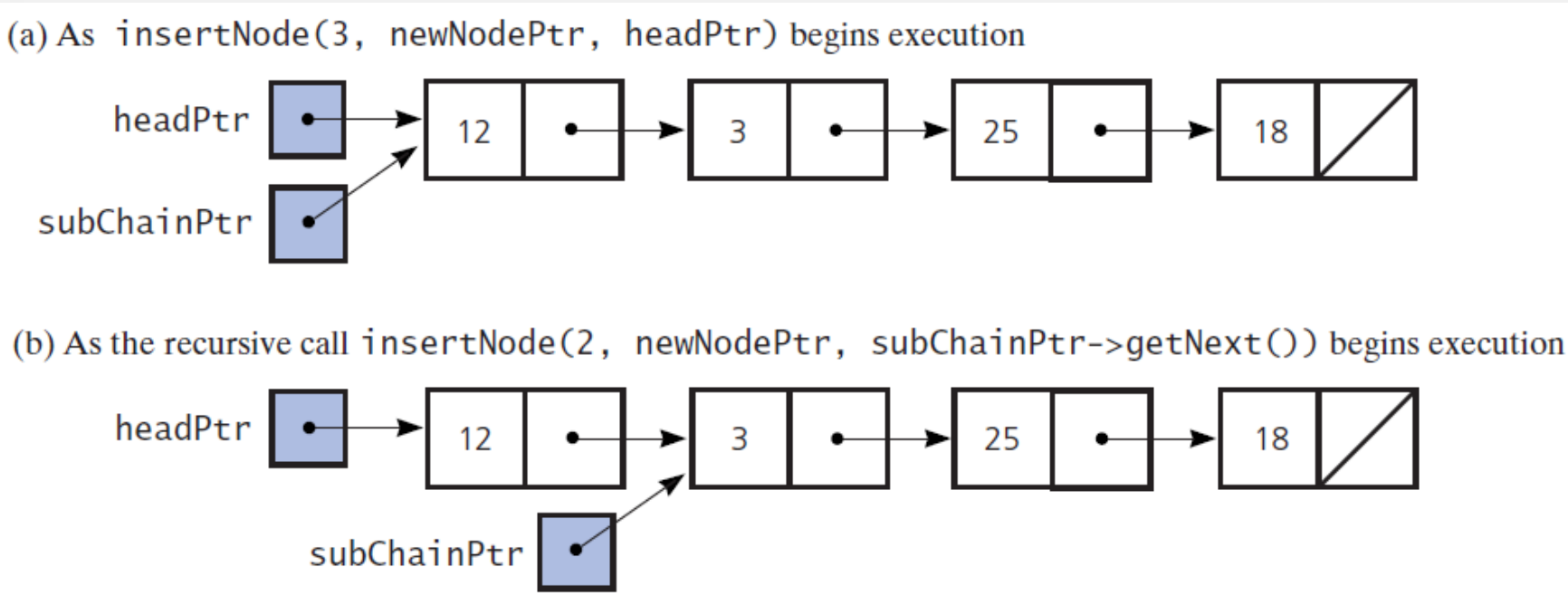


(c) As `insertNode(1, newNodePtr, headPtr)` begins execution



- Recursively adding a node at the beginning of a chain

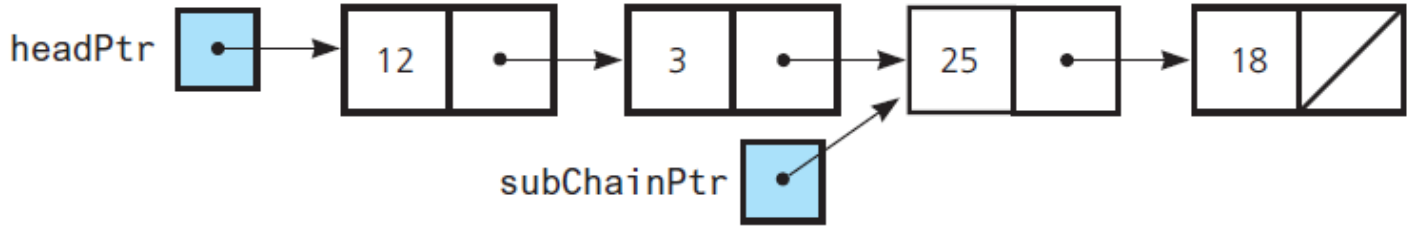
Using Recursion in LinkedList Methods



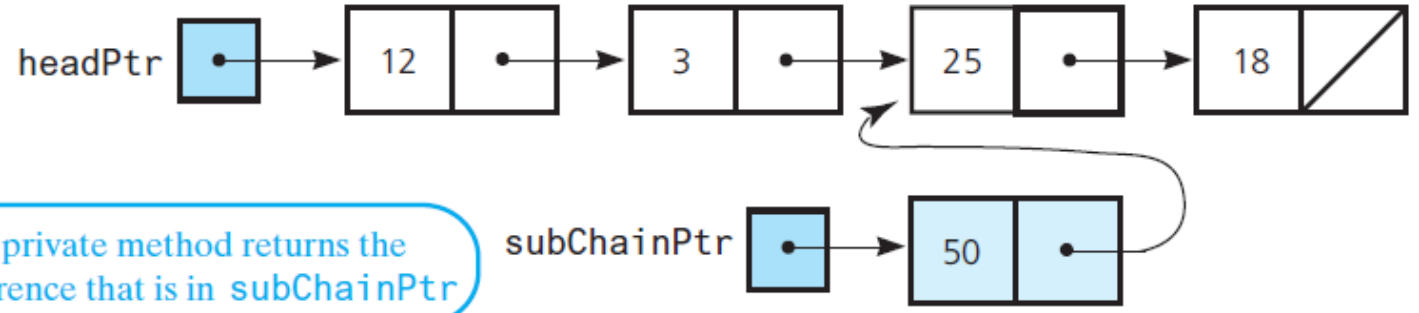
- Recursively adding a node between existing nodes in a chain

Using Recursion in LinkedList Methods

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

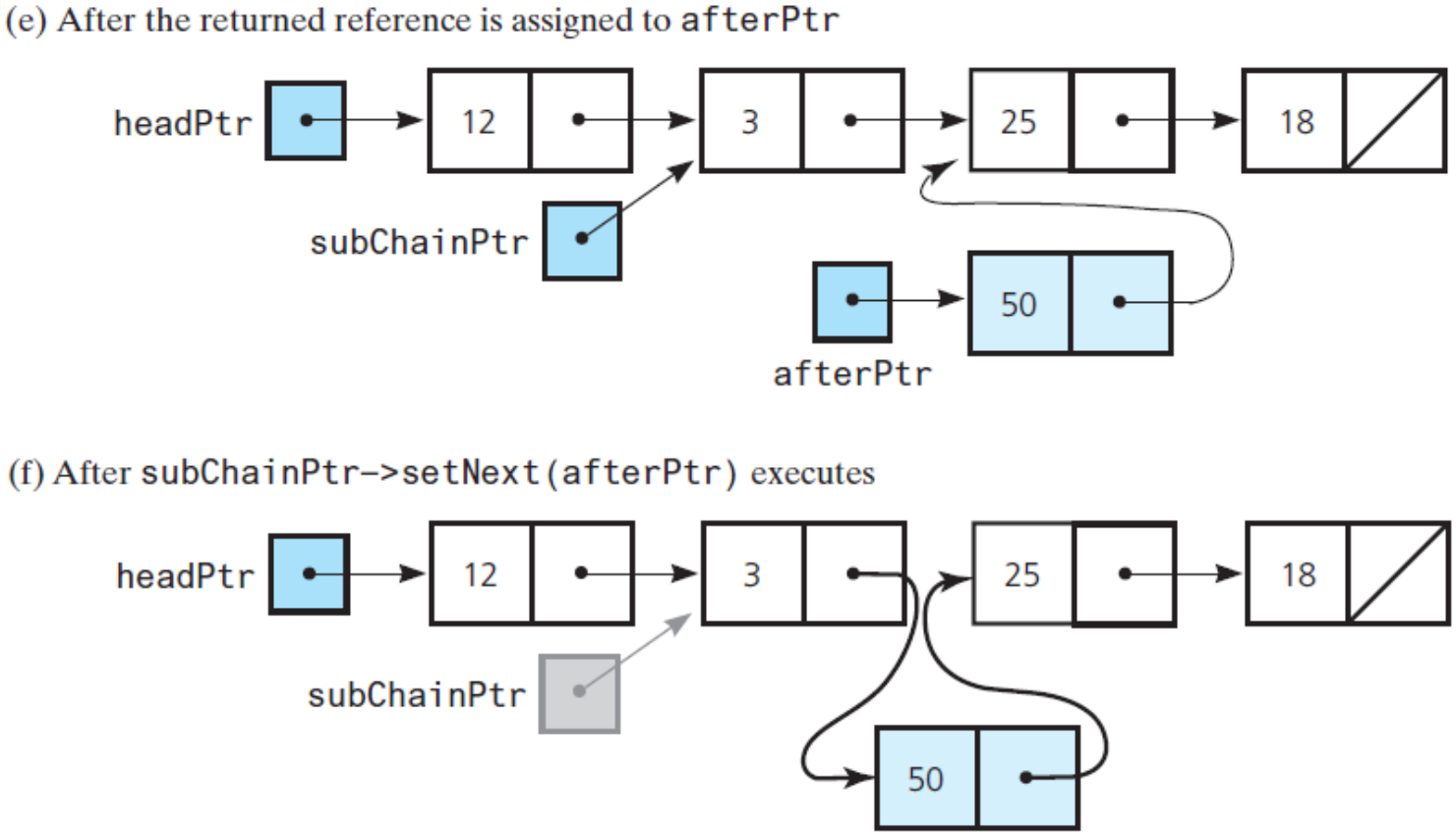


(d) After a new node is linked to the beginning of the subchain (the base case)



- Recursively adding a node between existing nodes in a chain

Using Recursion in LinkedList Methods



- Recursively adding a node between existing nodes in a chain

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