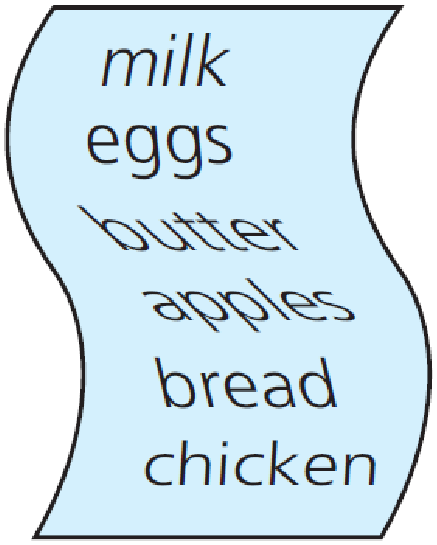
EECE 2560:
Fundamentals of Engineering Algorithms

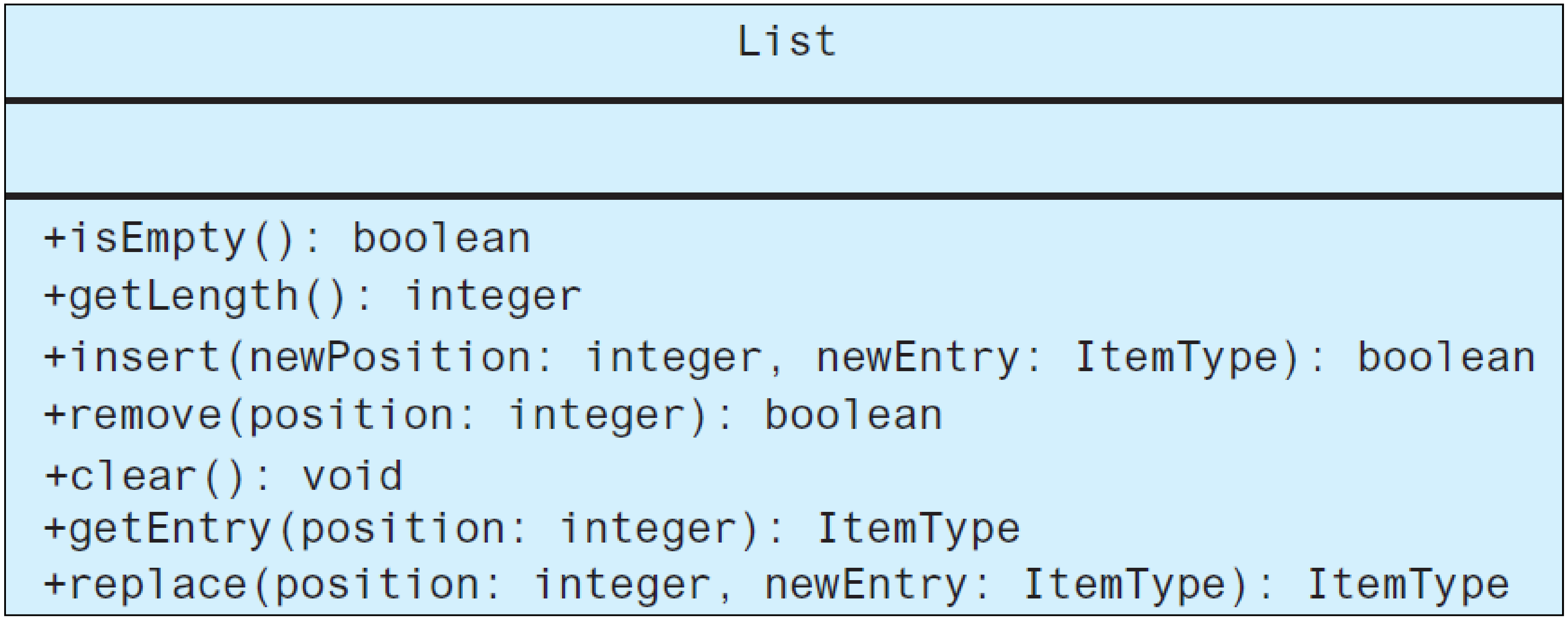
Lists

Specifying the ADT List

- Things you make lists of
  - Chores
  - Addresses
  - Groceries
- Lists contain items of the same type
  - Finite number of objects
  - Not necessarily distinct
  - Ordered by position as determined by user
- Operations
  - Count items
  - Add, remove items
  - Retrieve



List UML Diagram



```
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
```

Using the List Operations (1 of 2)

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 (2 of 2)

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, newItem)

    return success
```

Interface Template for ADT List (1 of 3)

```
template<class ItemType>
class ListInterface {
public:
 /** Sees whether this list is empty.
  @return True if the list is empty; otherwise returns false. */
   virtual bool isEmpty() const = 0;

 /** Gets the current number of entries in this list.
  @return The integer number of entries currently in the list. */
   virtual int getLength() const = 0;
```

Interface Template for ADT List (2 of 3)

```
 /** Inserts an entry into this list at a given position.
  @pre  None.
  @post  If 1 <= position <= getLength() + 1 and the insertion is
   successful, newEntry is at the given position in the list, other
   entries are renumbered accordingly and the returned value is true.
  @param newPosition  The list position at which to insert newEntry.
  @param newEntry  The entry to insert into the list.
  @return  True if insertion is successful, or false if not. */
  virtual bool insert(int newPosition, const ItemType& newEntry) = 0;

/** Removes the entry at a given position from this list.
  @pre  None.
  @post  If 1 <= position <= getLength() and the removal is
   successful, the entry at the given position in the list is
   removed, other items are renumbered accordingly, and the returned
   value is true.
  @param position  The list position of the entry to remove.
  @return  True if removal is successful, or false if not. */
   virtual bool remove(int position) = 0;
```

Interface Template for ADT List (3 of 3)

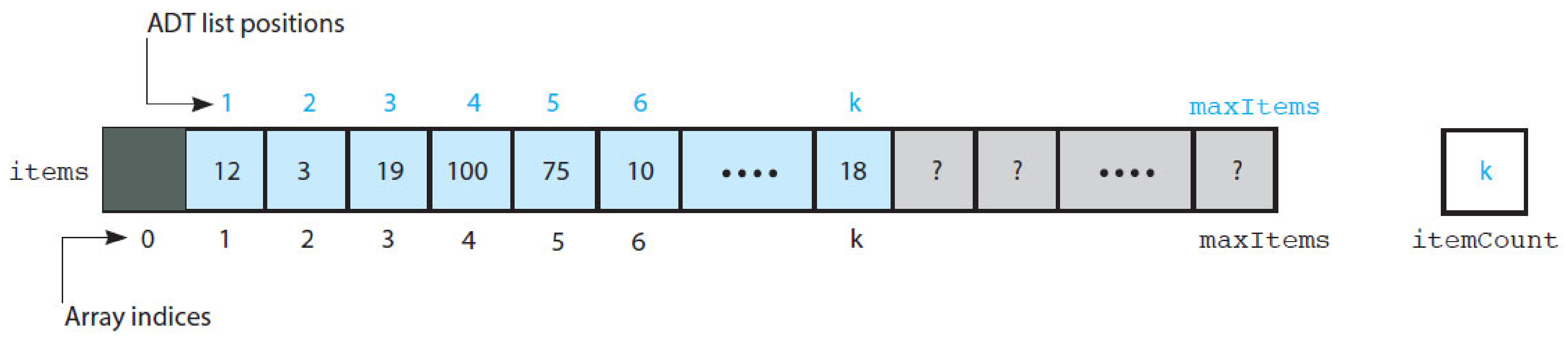
```
   /** Removes all entries from this list.
    @post  List contains no entries and the count of items is 0. */
   virtual void clear() = 0;

  /** Gets the entry at the given position in this list.
    @pre  1 <= position <= getLength().
    @post  The desired entry has been returned.
    @param position  The list position of the desired entry.
    @return  The entry at the given position. */
   virtual ItemType getEntry(int position) const = 0;

   /** Replaces the entry at the given position in this list.
    @pre  1 <= position <= getLength().
    @post  The entry at the given position is newEntry.
    @param position  The list position of the entry to replace.
    @param newEntry  The replacement entry. */
   virtual void replace(int position, const ItemType& newEntry) = 0;
}; // end ListInterface
```

Array-Based Implementation

- 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



ArrayList Header File

```
template<class ItemType>
class ArrayList : public ListInterface<ItemType>
{
private:
  static const int DEFAULT_CAPACITY = 5; // Small capacity to test for a full list
  ItemType items[DEFAULT_CAPACITY+1]; // Array of list items (not using element [0]
   int itemCount;          // Current count of list items
   int maxItems;          // Maximum capacity of the list

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

ArrayList Implementation (1 of 8)

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
```

ArrayList Implementation (2 of 8)

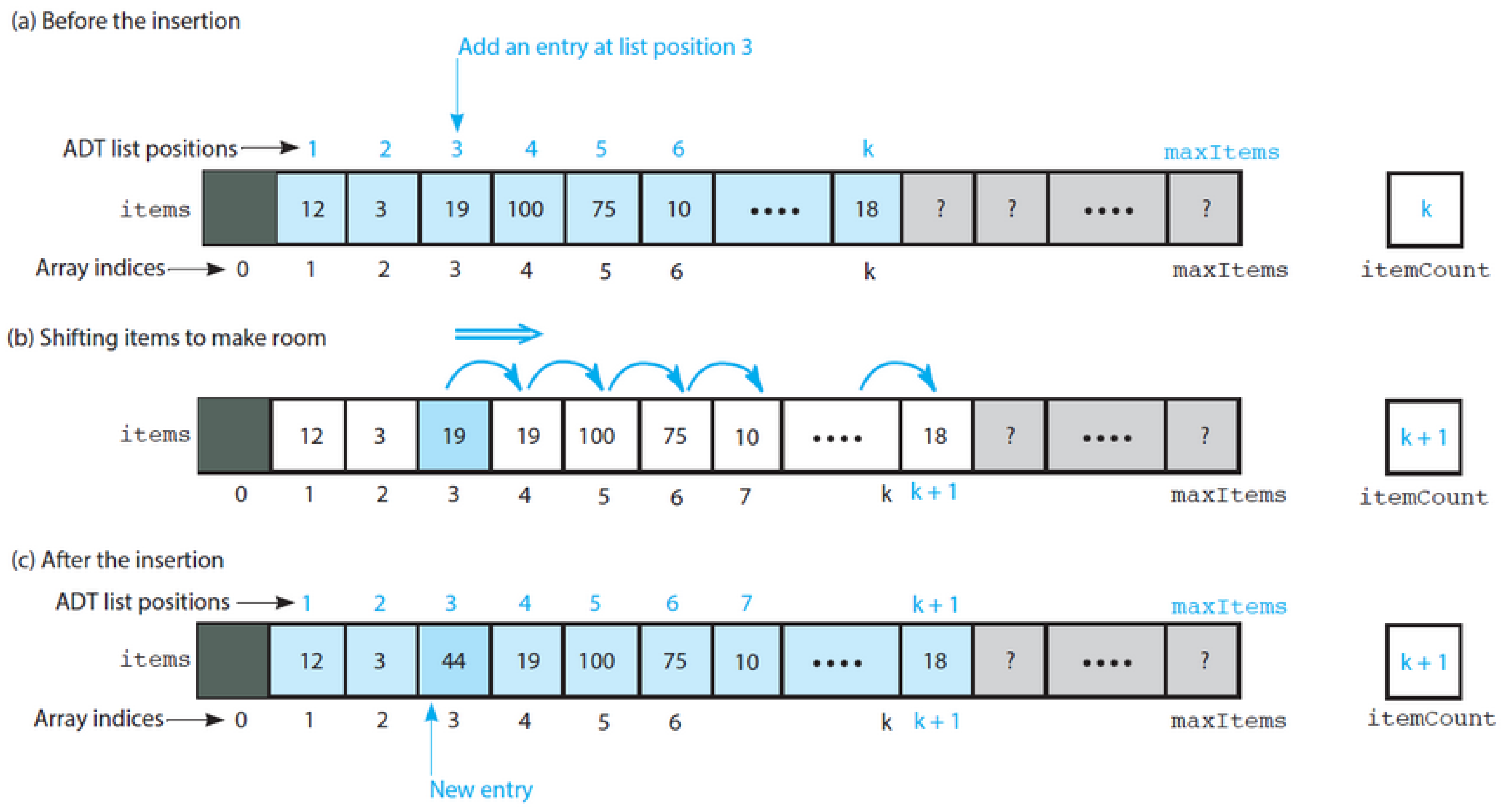
Method **insert**

```
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 at positions >= newPosition toward
   // the end of the array (no shift if newPosition == itemCount + 1)
   for (int entryPosition = itemCount;
            entryPosition >= newPosition; entryPosition--)
     items[entryPosition+1] = items[entryPosition]; // copy the entry right
   // Insert new entry
   items[newPosition] = newEntry;
   itemCount++;  // Increase count of entries
 }  // end if

 return ableToInsert;
}  // end insert
```

ArrayList Implementation (3 of 8)

Shifting items for insertion:



(a) Before the insertion
Add an entry at list position 3
ADT list positions → 1 2 3 4 5 6 k maxItems
items
12 3 19 100 75 10 •••• 18 ? ? •••• ?
Array indices → 0 1 2 3 4 5 6 k maxItems
k
itemCount
(b) Shifting items to make room
items
12 3 19 19 100 75 10 •••• 18 ? •••• ?
0 1 2 3 4 5 6 7 k k + 1 maxItems
k + 1
itemCount
(c) After the insertion
ADT list positions → 1 2 3 4 5 6 7 k + 1 maxItems
items
12 3 44 19 100 75 10 •••• 18 ? •••• ?
Array indices → 0 1 2 3 4 5 6 k k + 1 maxItems
k + 1
itemCount
New entry

ArrayList Implementation (4 of 8)

Method **remove**

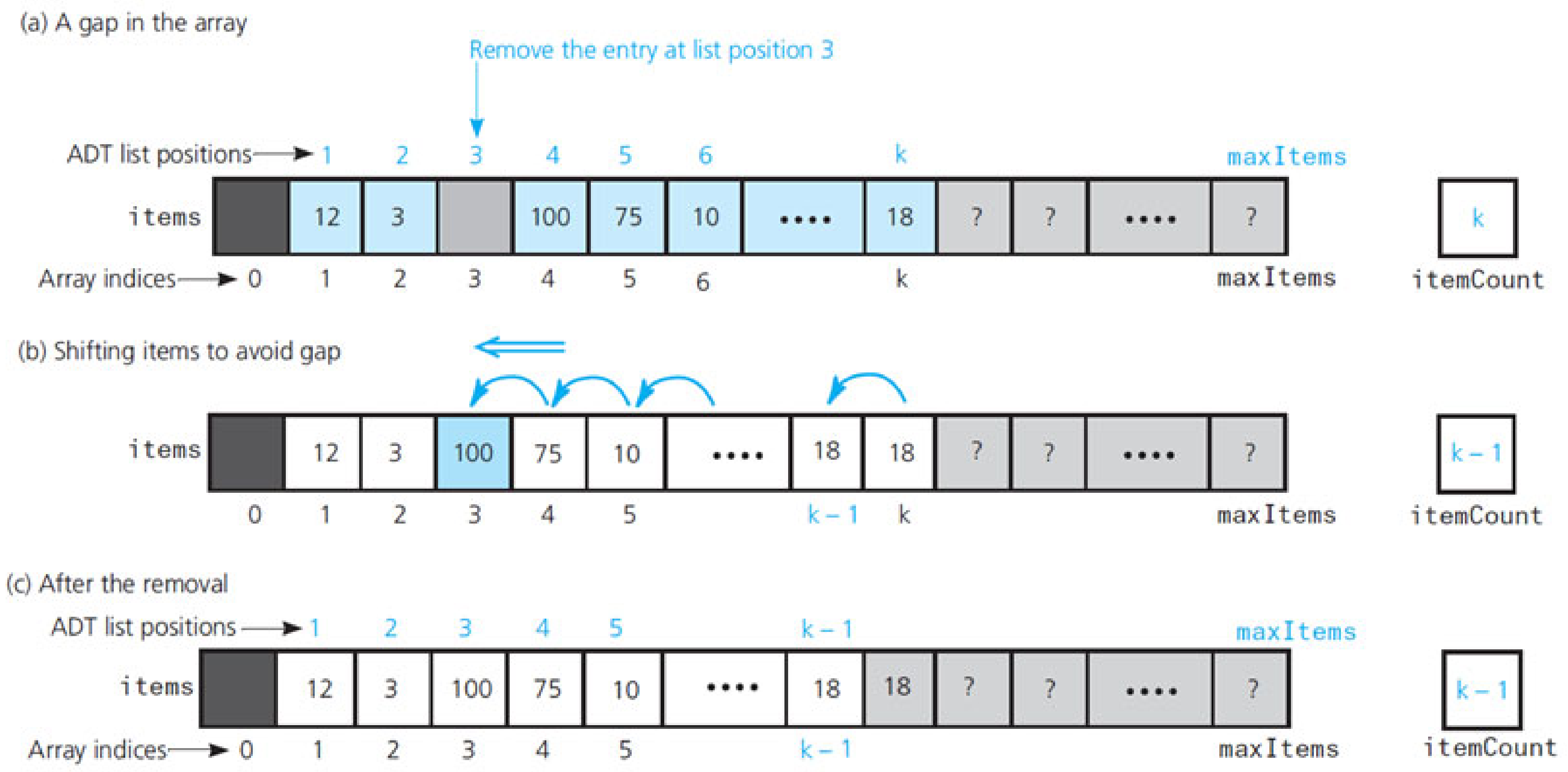
```
template<class ItemType>
bool ArrayList<ItemType>::remove(int position)
{
 bool ableToRemove = (position >= 1) && (position <= itemCount);
   if (ableToRemove){
    // Remove entry by shifting all entries after the one at position toward the beginning
    // of the array/ (no shift if position ==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

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

   return ableToRemove;
}  // end remove
```

ArrayList Implementation (5 of 8)

Shifting items to remove an entry:



ArrayList Implementation (6 of 8)

Method **clear**

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

ArrayList Implementation (7 of 8)

Method **getEntry**

```
template<class ItemType>
ItemType ArrayList<ItemType>::getEntry(int position) const
{
   // Enforce precondition
   bool ableToGet = (position >= 1) && (position <= itemCount);

  // If not able to get, the program is terminated and display an error message
  assert(ableToGet);

  return items[position];
}  // end getEntry
```

ArrayList Implementation (8 of 8)

Method **replace**

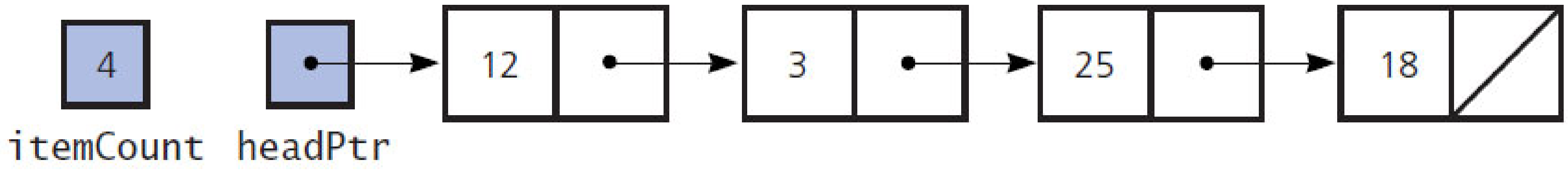
```
template<class ItemType>
void ArrayList<ItemType>::replace(int position,
                                  const ItemType& newEntry)
{
   // Enforce precondition
   bool ableToSet = (position >= 1) && (position <= itemCount);

   assert(ableToSet);

   items[position] = newEntry;
}  // end replace
```

Link-Based Implementation

- 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



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