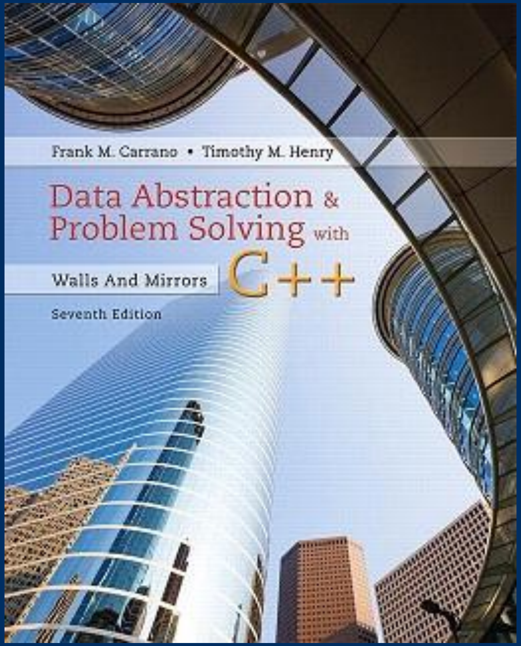
Chapter 4

Link Based Implementations

CS 302 - Data Structures

M. Abdullah Canbaz


Frank M. Carrano • Timothy M. Henry
Data Abstraction & Problem Solving with
Walls And Mirrors C++
Seventh Edition

Modified from authors' slides

Preliminaries

- Another way to organize data items
  - – Place them within objects—usually called nodes
  - – Linked together into a “chain,” one after the other

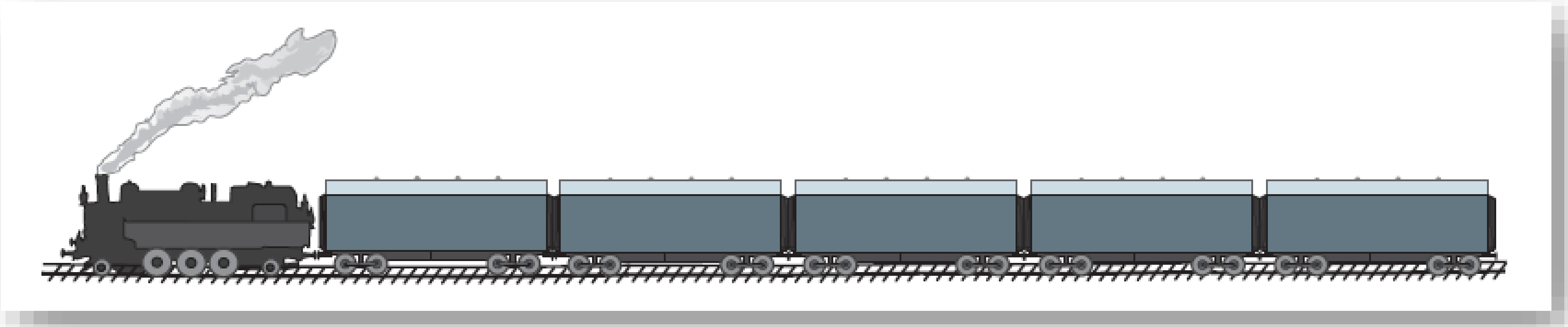
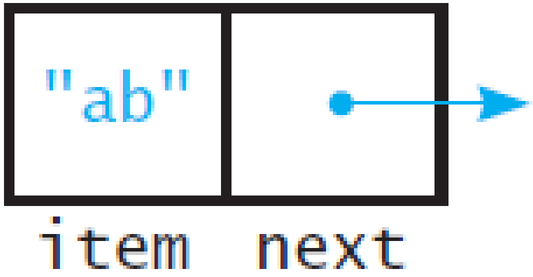


Figure 4-1 A freight train

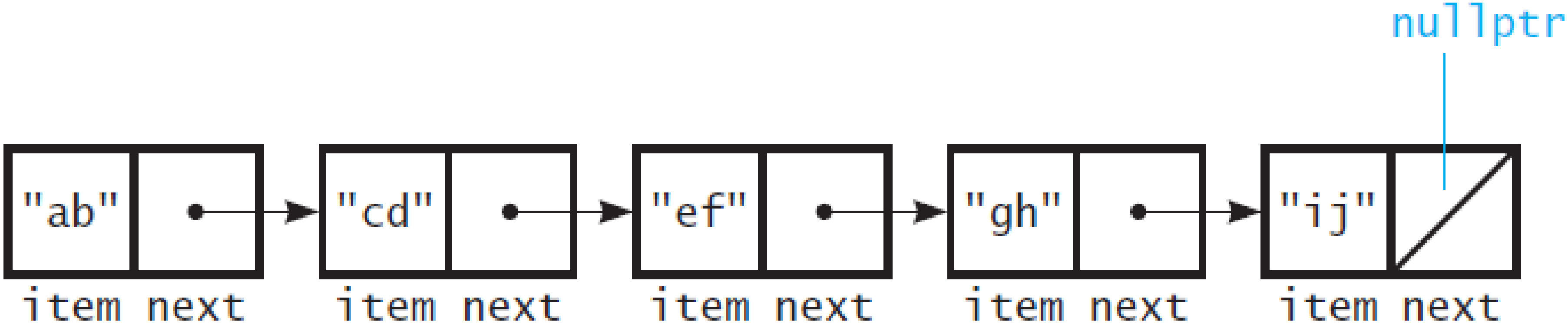
Preliminaries

- Components that can be linked



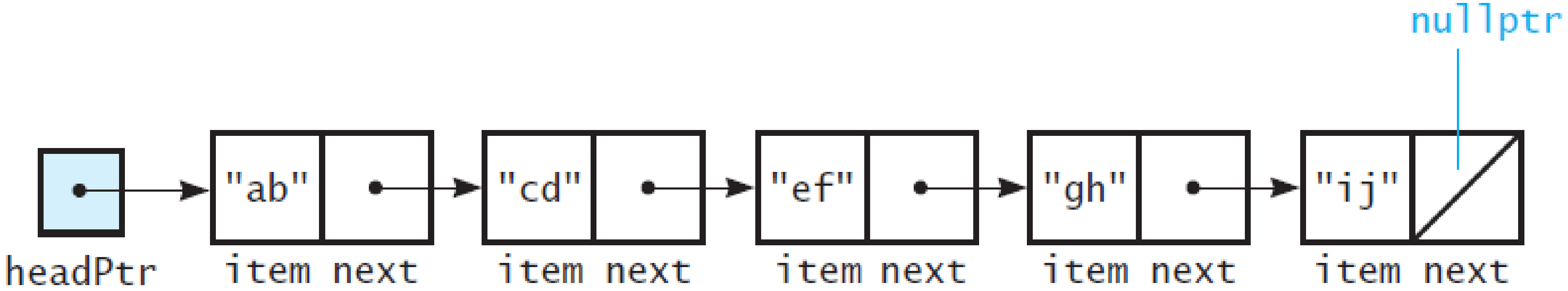
A node

Preliminaries



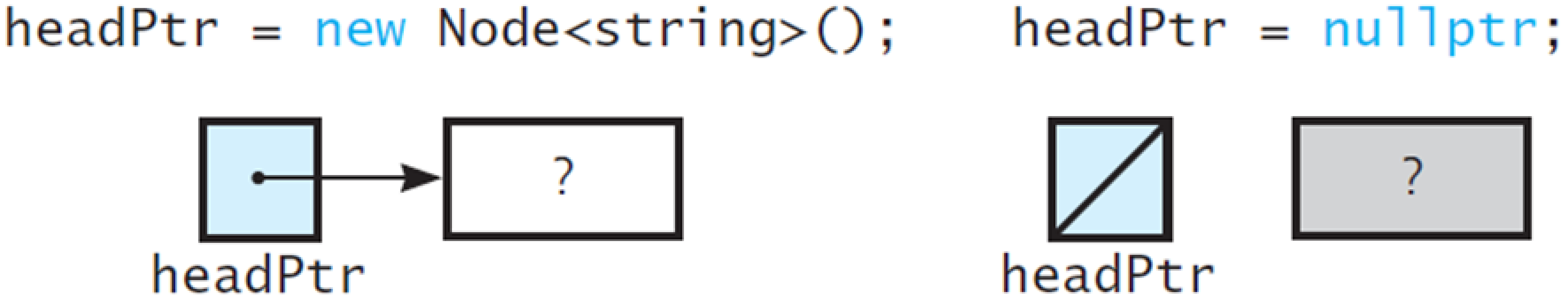
- Several nodes linked together

Preliminaries



- A head pointer to the first of several linked nodes

Preliminaries

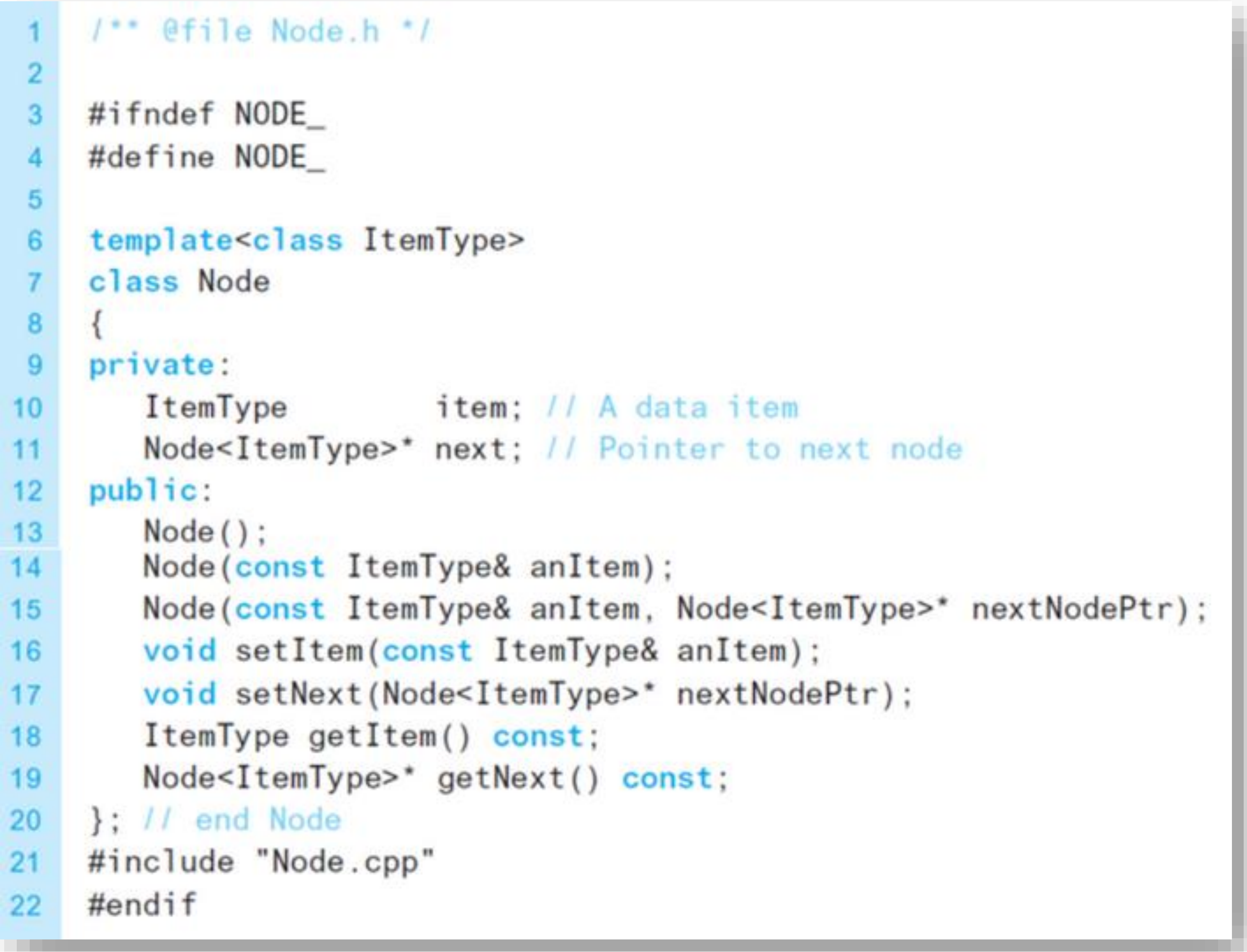


```
headPtr = new Node<string>();
```

```
headPtr = nullptr;
```

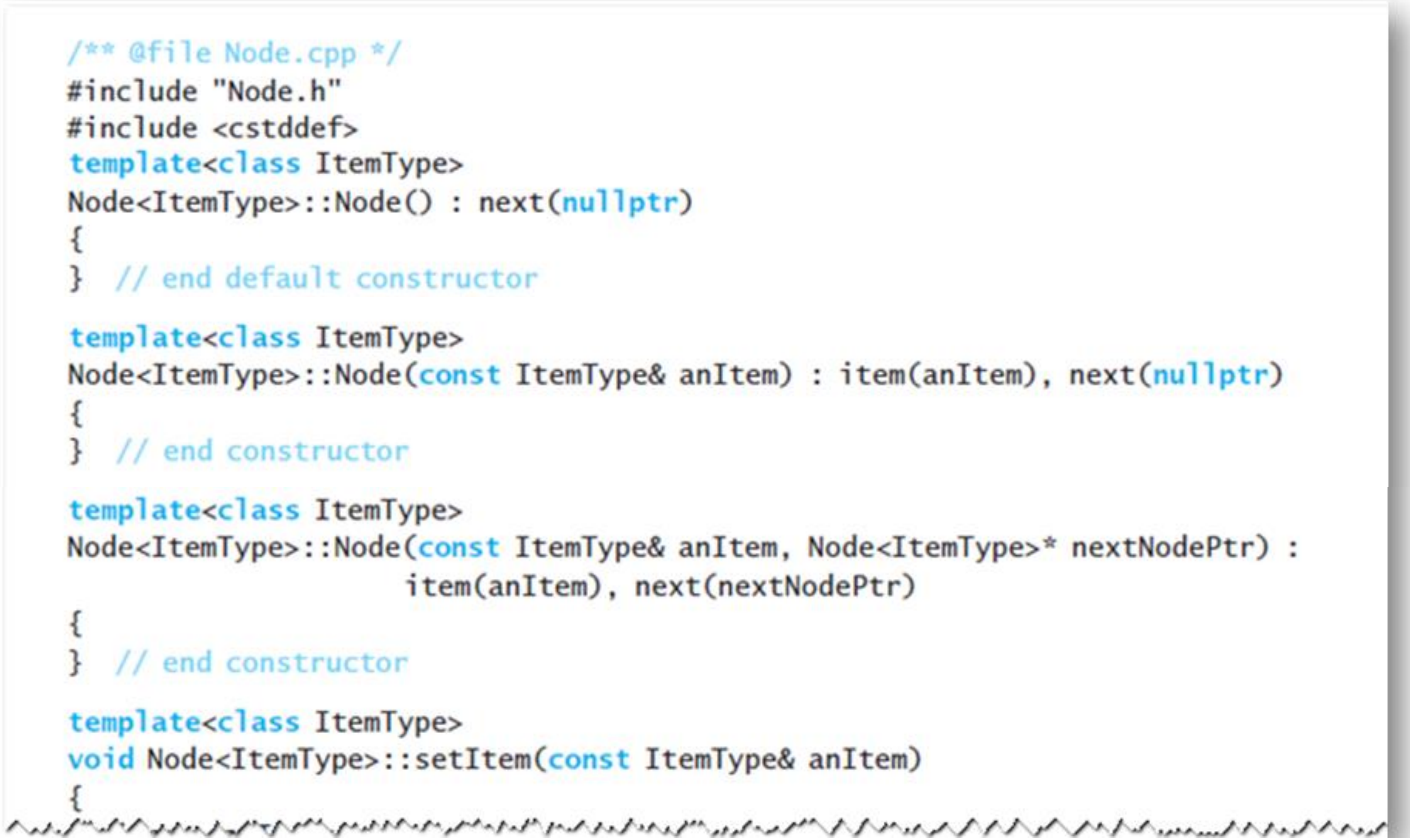
- A lost node

The Class **Node**



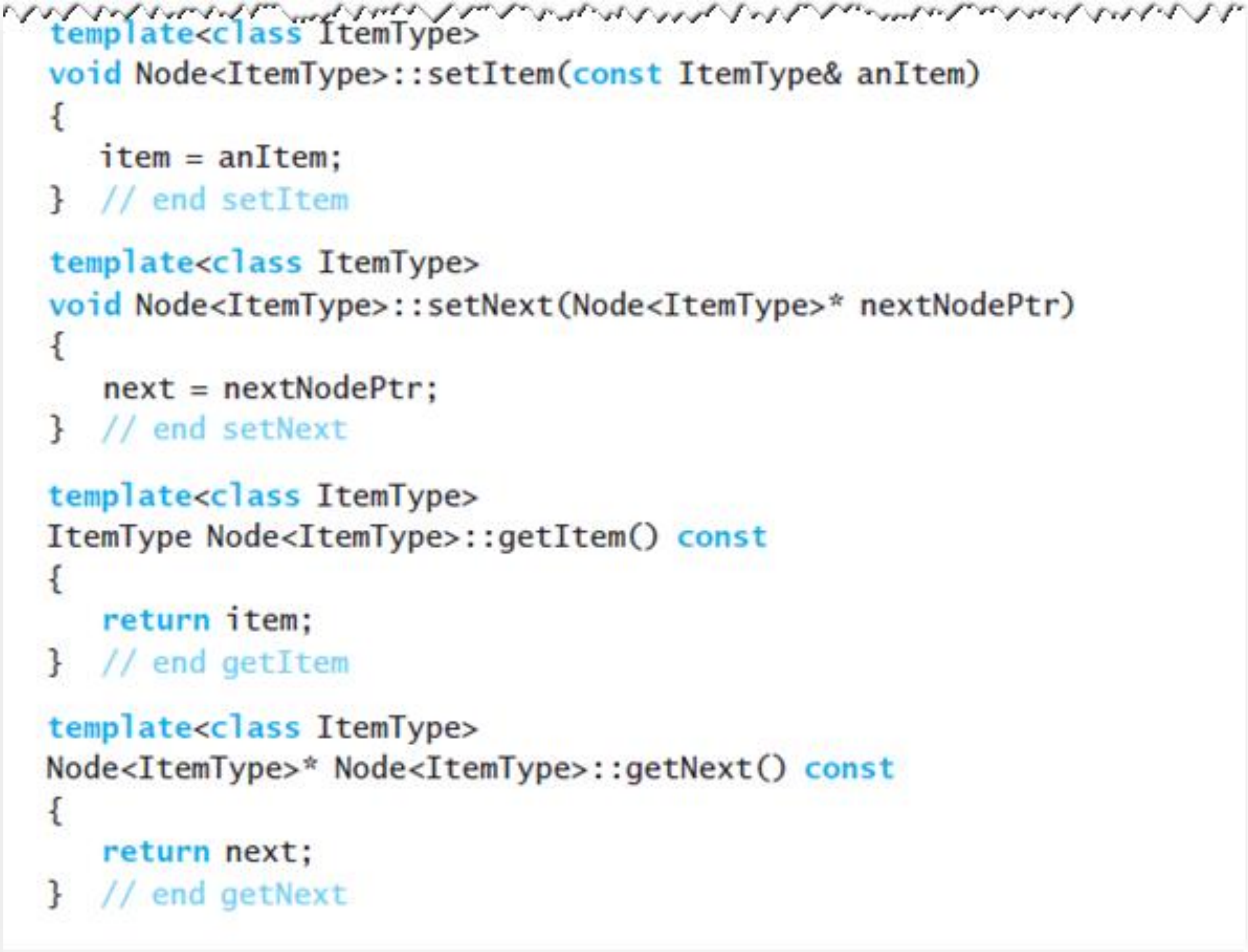
```
/** @file Node.h */

#ifndef NODE_
#define NODE_

template<class ItemType>
class Node
{
private:
    ItemType        item; // A data item
    Node<ItemType>* next; // Pointer to next node
public:
    Node();
    Node(const ItemType& anItem);
    Node(const ItemType& anItem, Node<ItemType>* nextNodePtr);
    void setItem(const ItemType& anItem);
    void setNext(Node<ItemType>* nextNodePtr);
    ItemType getItem() const;
    Node<ItemType>* getNext() const;
}; // end Node
#include "Node.cpp"
#endif
```

The Class **Node**



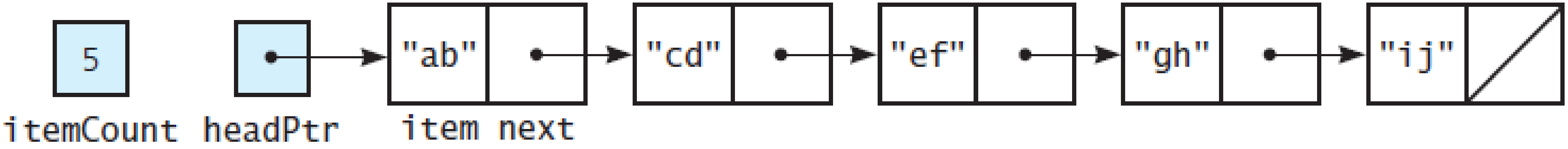
```
/** @file Node.cpp */
#include "Node.h"
#include <cstddef>
template<class ItemType>
Node<ItemType>::Node() : next(nullptr)
{
}  // end default constructor

template<class ItemType>
Node<ItemType>::Node(const ItemType& anItem) : item(anItem), next(nullptr)
{
}  // end constructor

template<class ItemType>
Node<ItemType>::Node(const ItemType& anItem, Node<ItemType>* nextNodePtr) :
                     item(anItem), next(nextNodePtr)
{
}  // end constructor

template<class ItemType>
void Node<ItemType>::setItem(const ItemType& anItem)
{
```

The Class **Node**

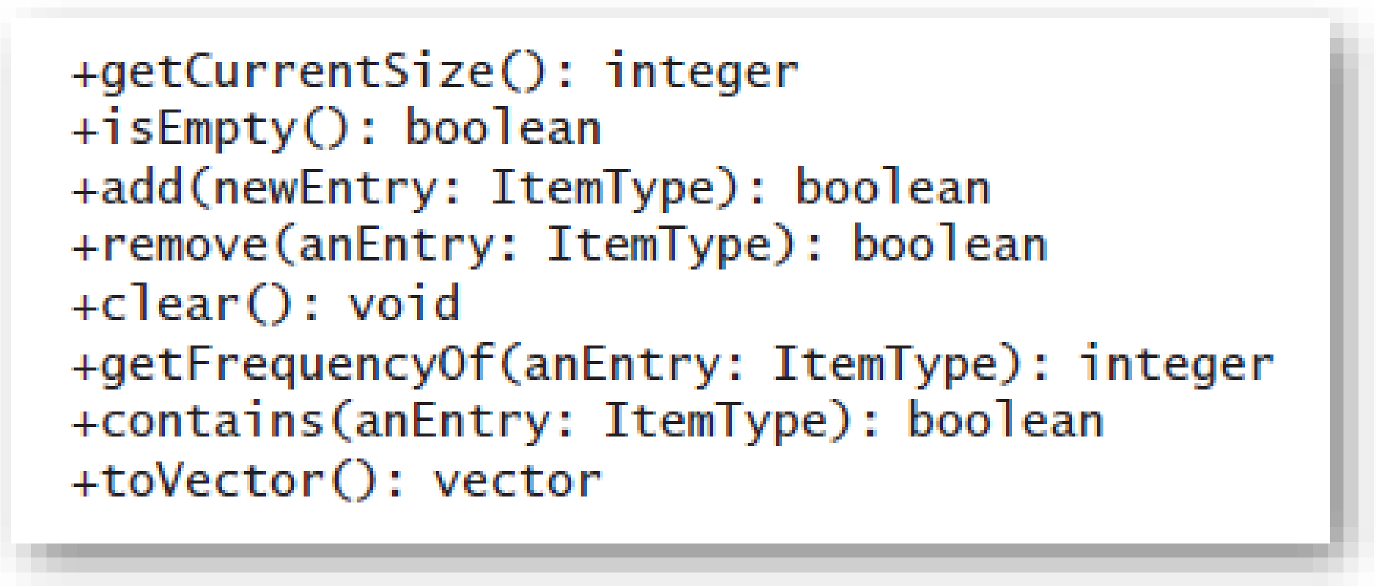


```
template<class ItemType>
void Node<ItemType>::setItem(const ItemType& anItem)
{
   item = anItem;
}  // end setItem

template<class ItemType>
void Node<ItemType>::setNext(Node<ItemType>* nextNodePtr)
{
   next = nextNodePtr;
}  // end setNext

template<class ItemType>
ItemType Node<ItemType>::getItem() const
{
   return item;
}  // end getItem

template<class ItemType>
Node<ItemType>* Node<ItemType>::getNext() const
{
   return next;
}  // end getNext
```

Link-Based Implementation of ADT Bag



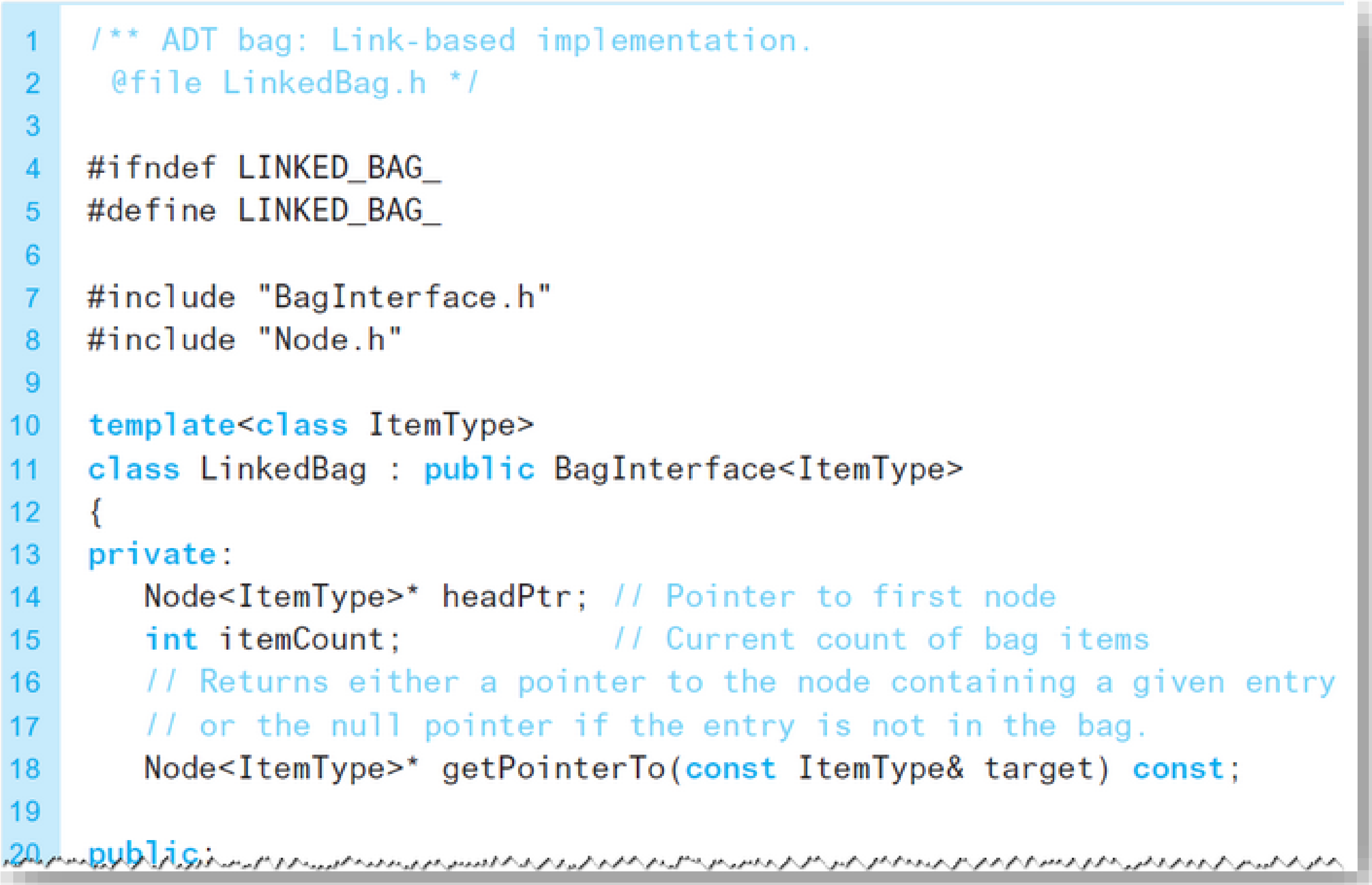
A link-based implementation of the ADT bag



```
+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: ItemType): boolean
+remove(anEntry: ItemType): boolean
+clear(): void
+getFrequencyOf(anEntry: ItemType): integer
+contains(anEntry: ItemType): boolean
+toVector(): vector
```

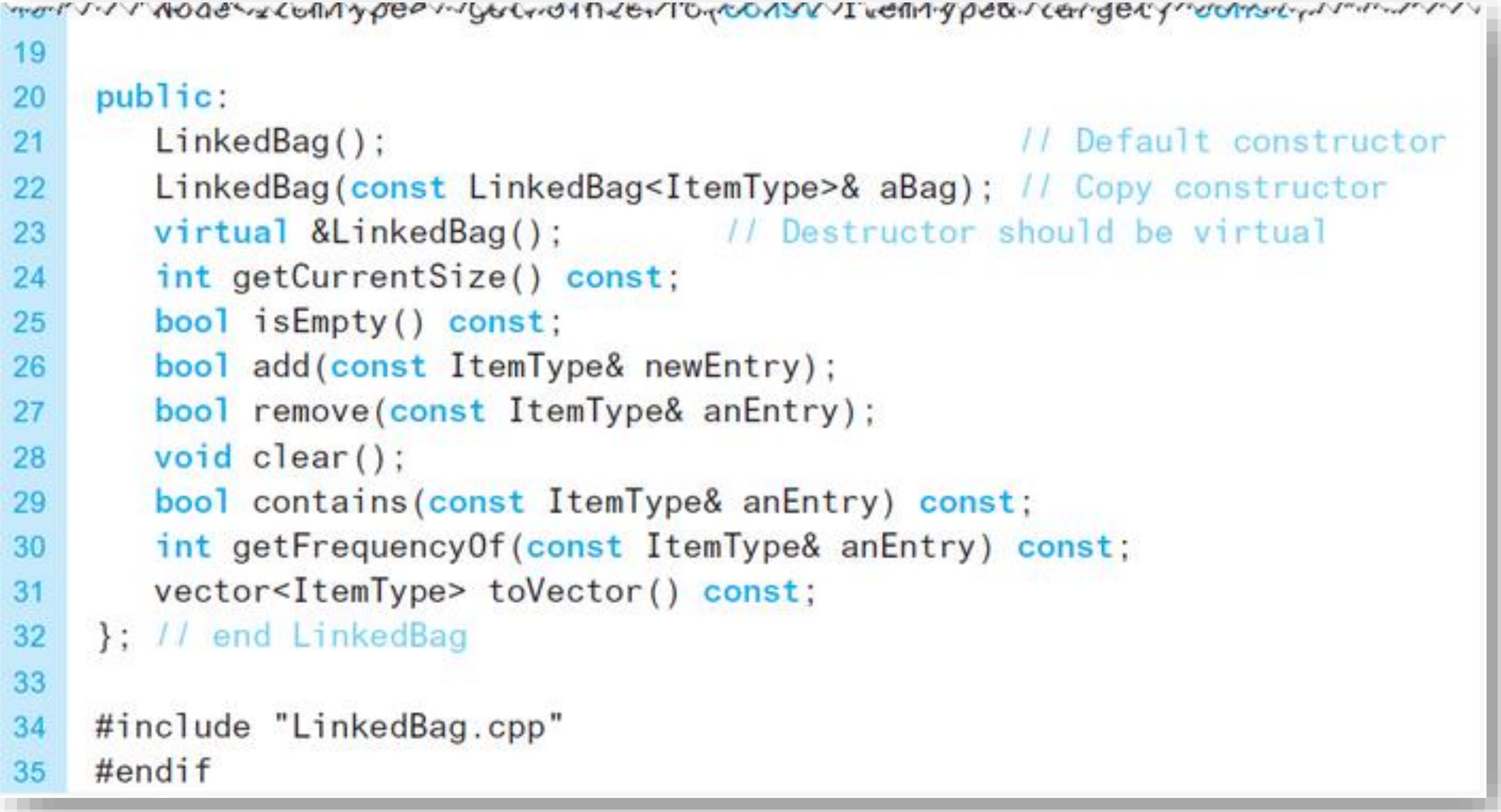
Bag operations, given in UML notation

The header file for the class LinkedBag



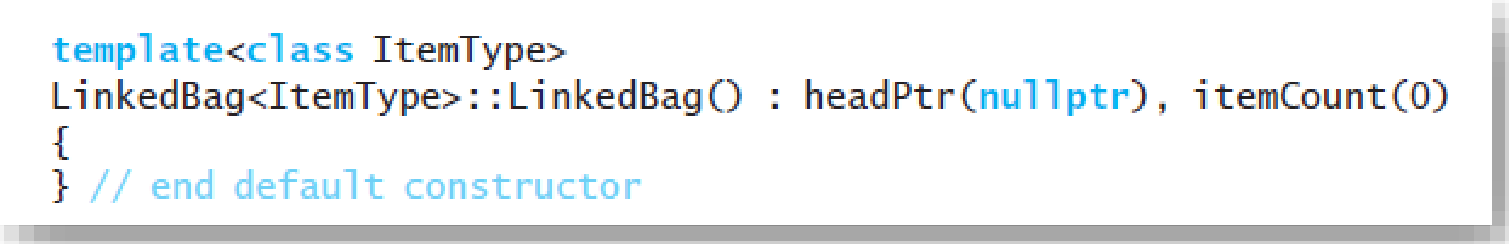
```
/** ADT bag: Link-based implementation.
 @file LinkedBag.h */

#ifndef LINKED_BAG_
#define LINKED_BAG_

#include "BagInterface.h"
#include "Node.h"

template<class ItemType>
class LinkedBag : public BagInterface<ItemType>
{
private:
   Node<ItemType>* headPtr; // Pointer to first node
   int itemCount;           // Current count of bag items
   // Returns either a pointer to the node containing a given entry
   // or the null pointer if the entry is not in the bag.
   Node<ItemType>* getPointerTo(const ItemType& target) const;

public:
```

The header file for the class LinkedBag



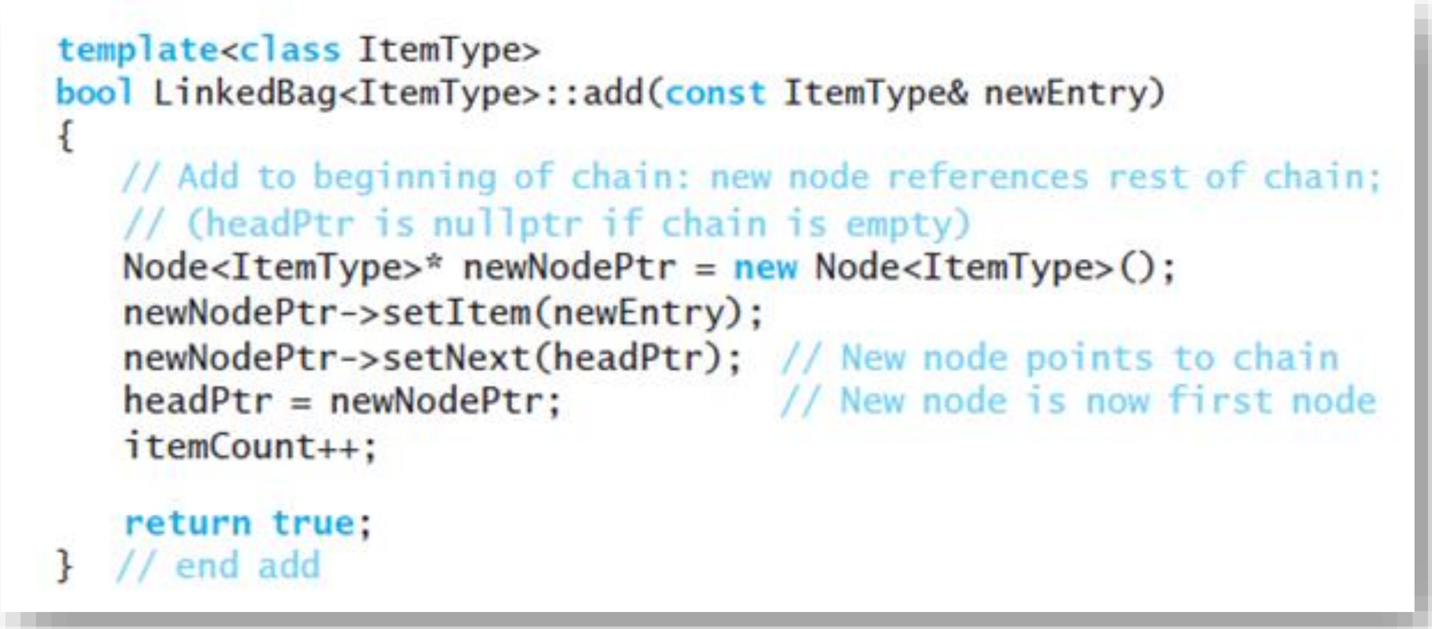
```
   Node<ItemType>* getPointerTo(const ItemType& target) const;

public:
   LinkedBag();                                   // Default constructor
   LinkedBag(const LinkedBag<ItemType>& aBag); // Copy constructor
   virtual &LinkedBag();              // Destructor should be virtual
   int getCurrentSize() const;
   bool isEmpty() const;
   bool add(const ItemType& newEntry);
   bool remove(const ItemType& anEntry);
   void clear();
   bool contains(const ItemType& anEntry) const;
   int getFrequencyOf(const ItemType& anEntry) const;
   vector<ItemType> toVector() const;
}; // end LinkedBag

#include "LinkedBag.cpp"
#endif
```

Defining the Core Methods



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

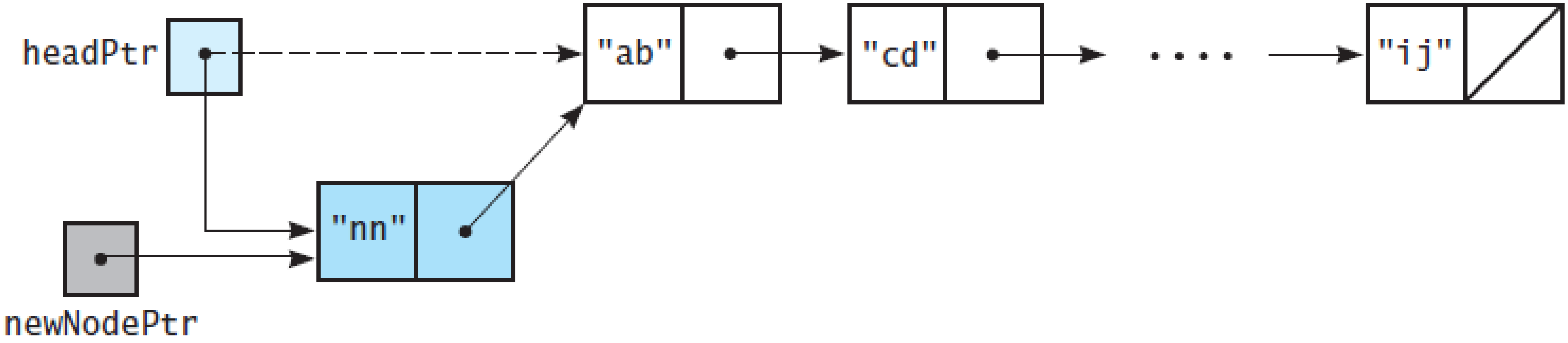
Default Constructor



```
template<class ItemType>
bool LinkedBag<ItemType>::add(const ItemType& newEntry)
{
   // Add to beginning of chain: new node references rest of chain;
   // (headPtr is nullptr if chain is empty)
   Node<ItemType>* newNodePtr = new Node<ItemType>();
   newNodePtr->setItem(newEntry);
   newNodePtr->setNext(headPtr);  // New node points to chain
   headPtr = newNodePtr;          // New node is now first node
   itemCount++;

   return true;
}  // end add
```

Inserting at the beginning of a linked chain

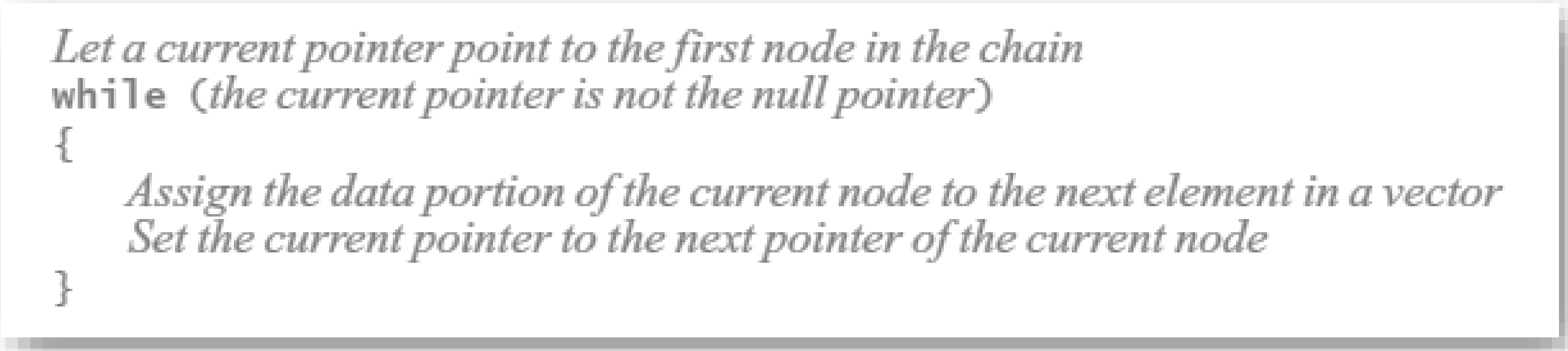
Link-Based Implementation of ADT Bag



Inserting at the beginning of a linked chain

Defining the Core Methods

- Traverse operation visits each node in linked chain
  - Must move from node to node

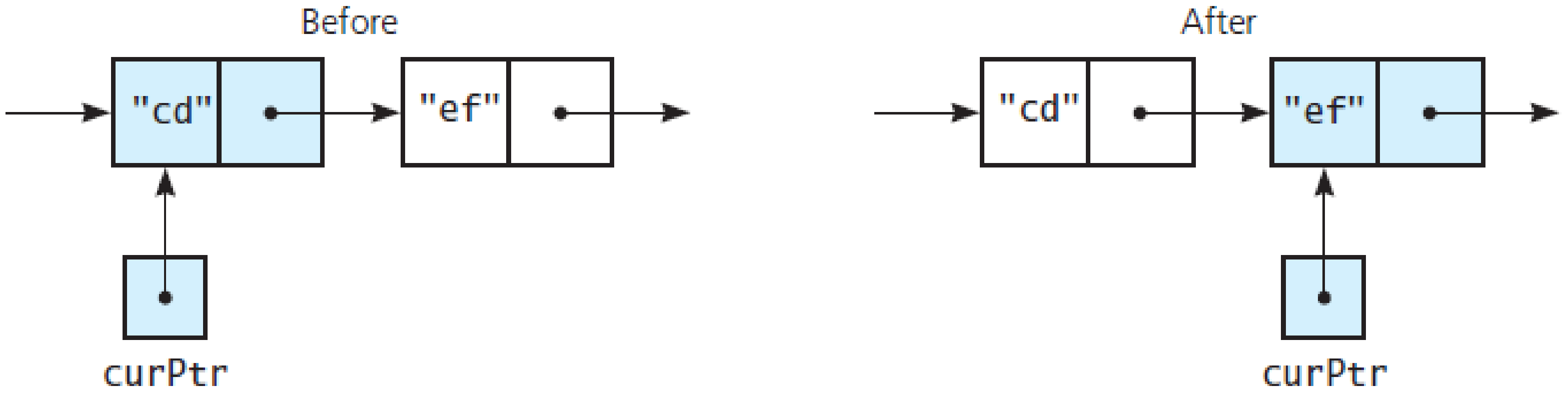


```
Let a current pointer point to the first node in the chain
while (the current pointer is not the null pointer)
{
    Assign the data portion of the current node to the next element in a vector
    Set the current pointer to the next pointer of the current node
}
```

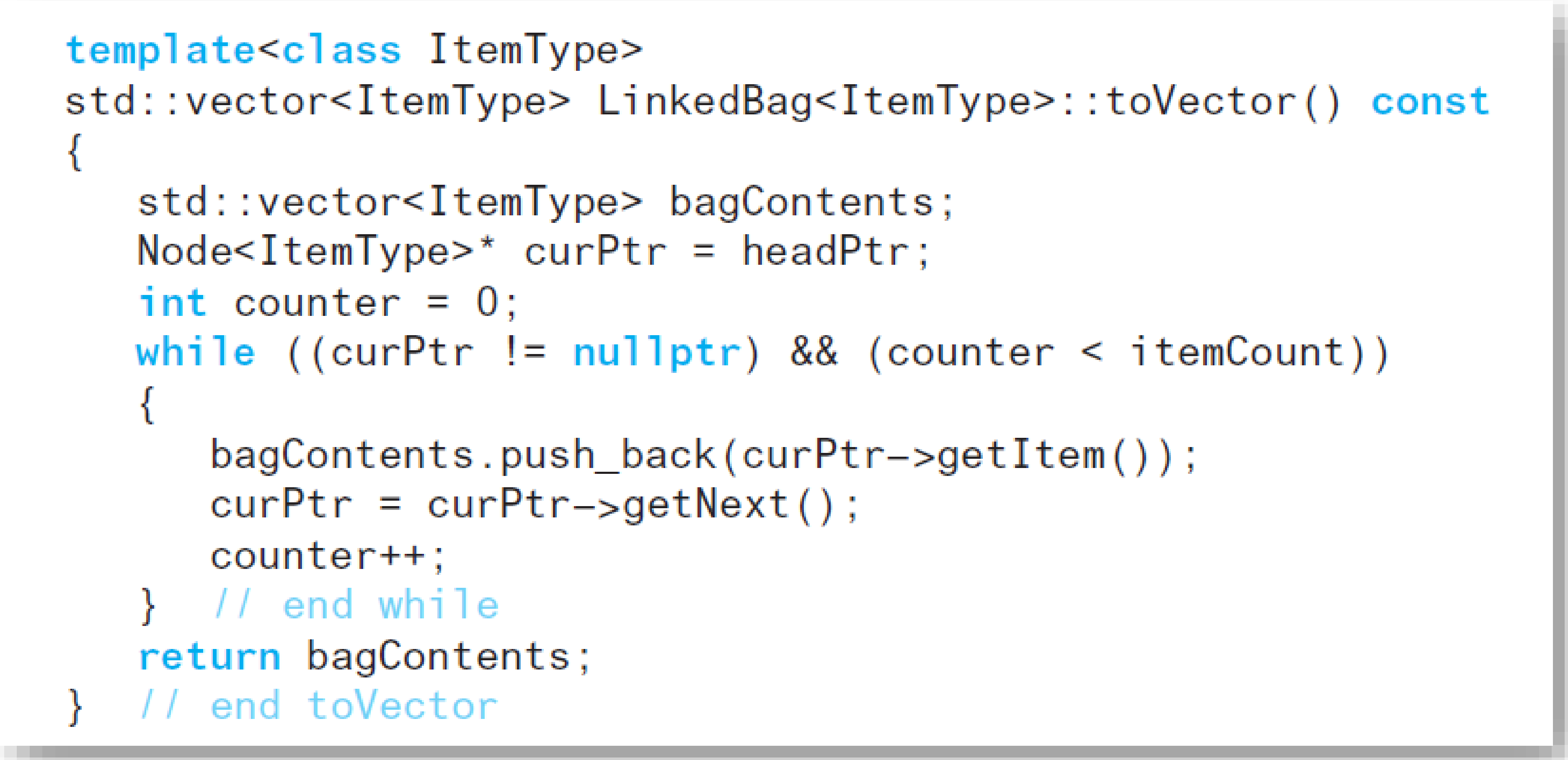
High-level pseudocode for this loop

Link-Based Implementation of ADT Bag

- The effect of the assignment
  `curPtr = curPtr->getNext()`



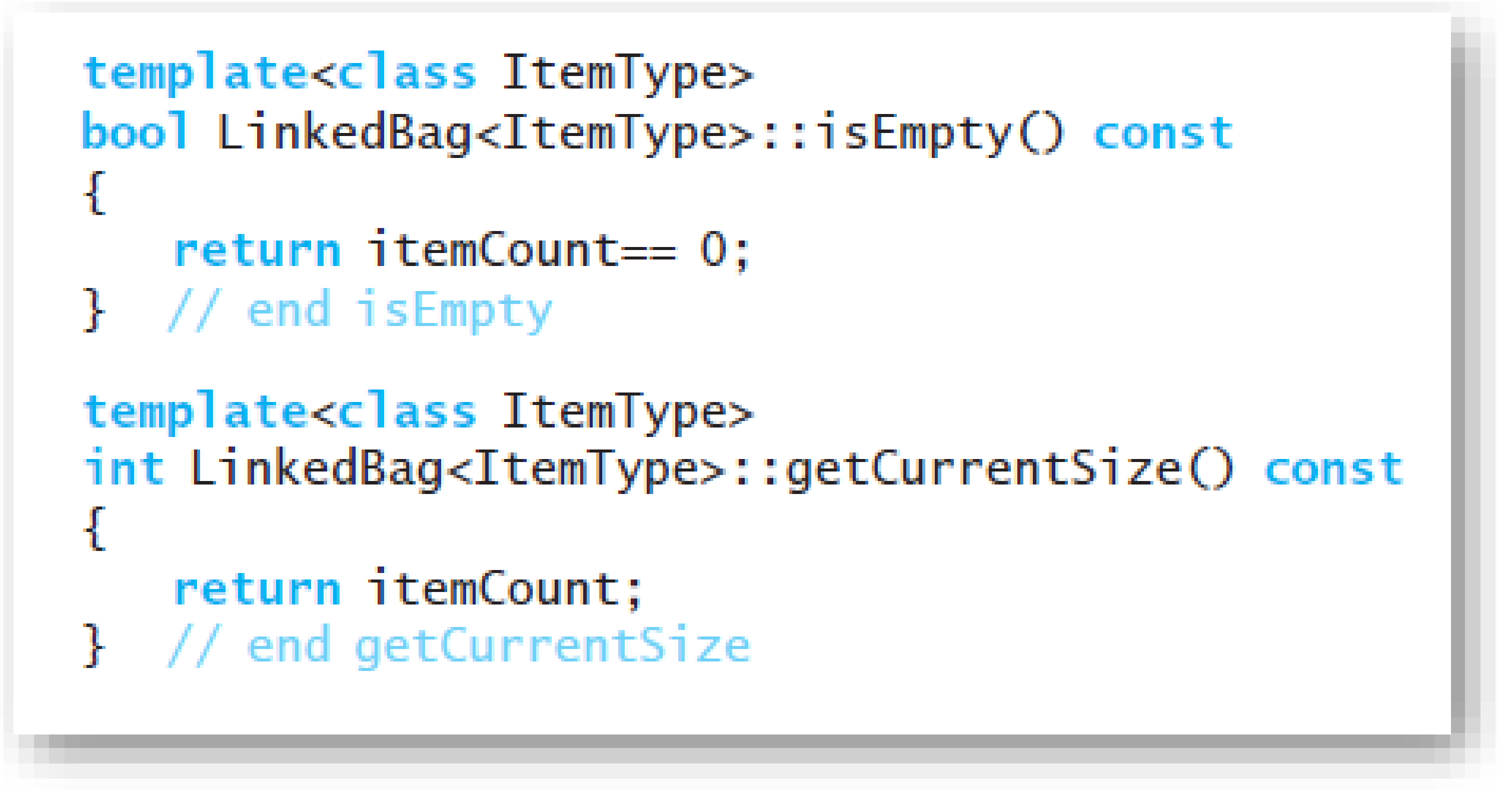
Defining the Core Methods



```
template<class ItemType>
std::vector<ItemType> LinkedBag<ItemType>::toVector() const
{
   std::vector<ItemType> bagContents;
   Node<ItemType>* curPtr = headPtr;
   int counter = 0;
   while ((curPtr != nullptr) && (counter < itemCount))
   {
      bagContents.push_back(curPtr->getItem());
      curPtr = curPtr->getNext();
      counter++;
   }  // end while
   return bagContents;
}  // end toVector
```

- Definition of toVector

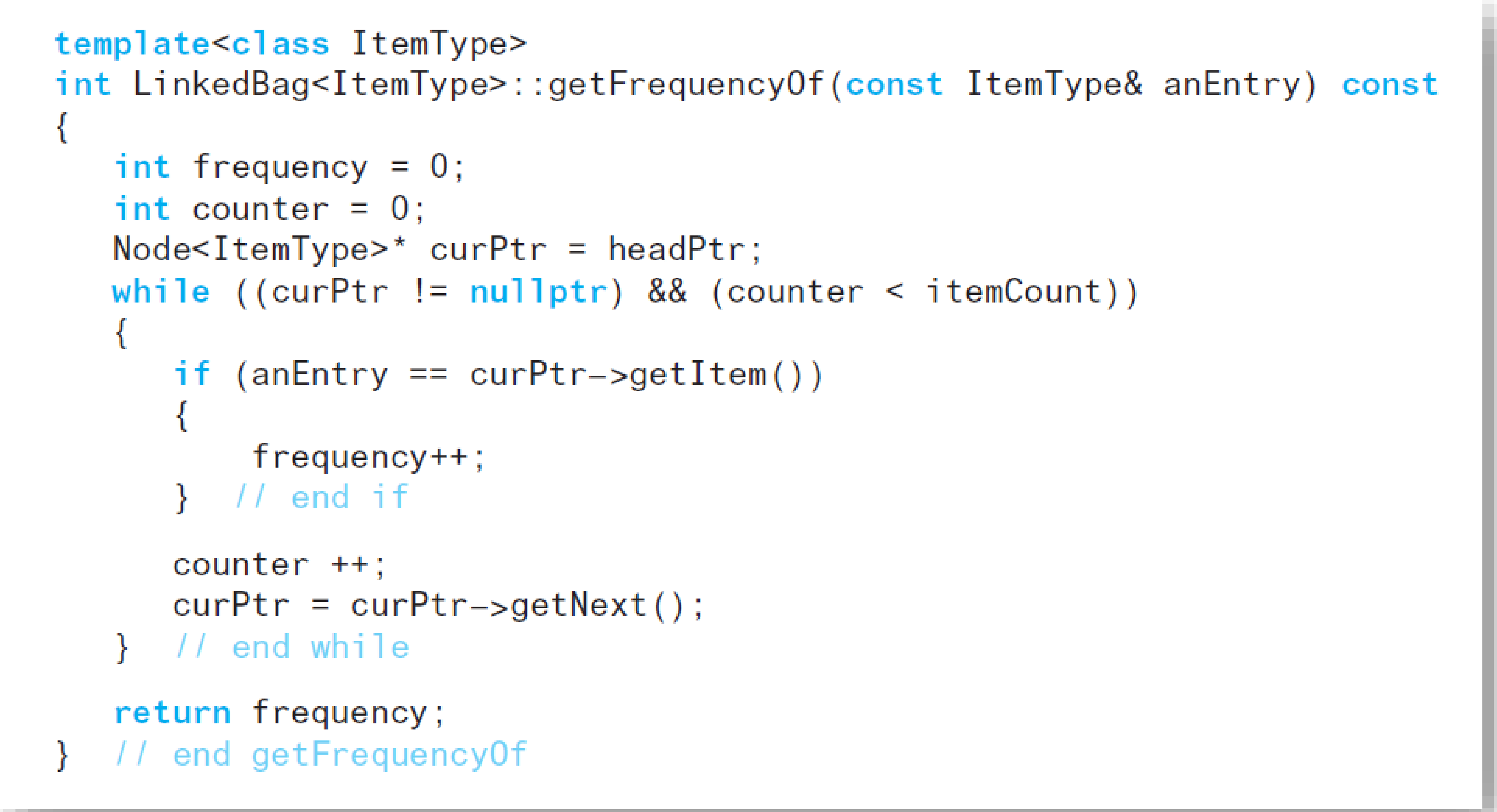
Defining the Core Methods



```
template<class ItemType>
bool LinkedBag<ItemType>::isEmpty() const
{
    return itemCount== 0;
}  // end isEmpty

template<class ItemType>
int LinkedBag<ItemType>::getCurrentSize() const
{
    return itemCount;
}  // end getCurrentSize
```

- Methods isEmpty and getCurrentSize

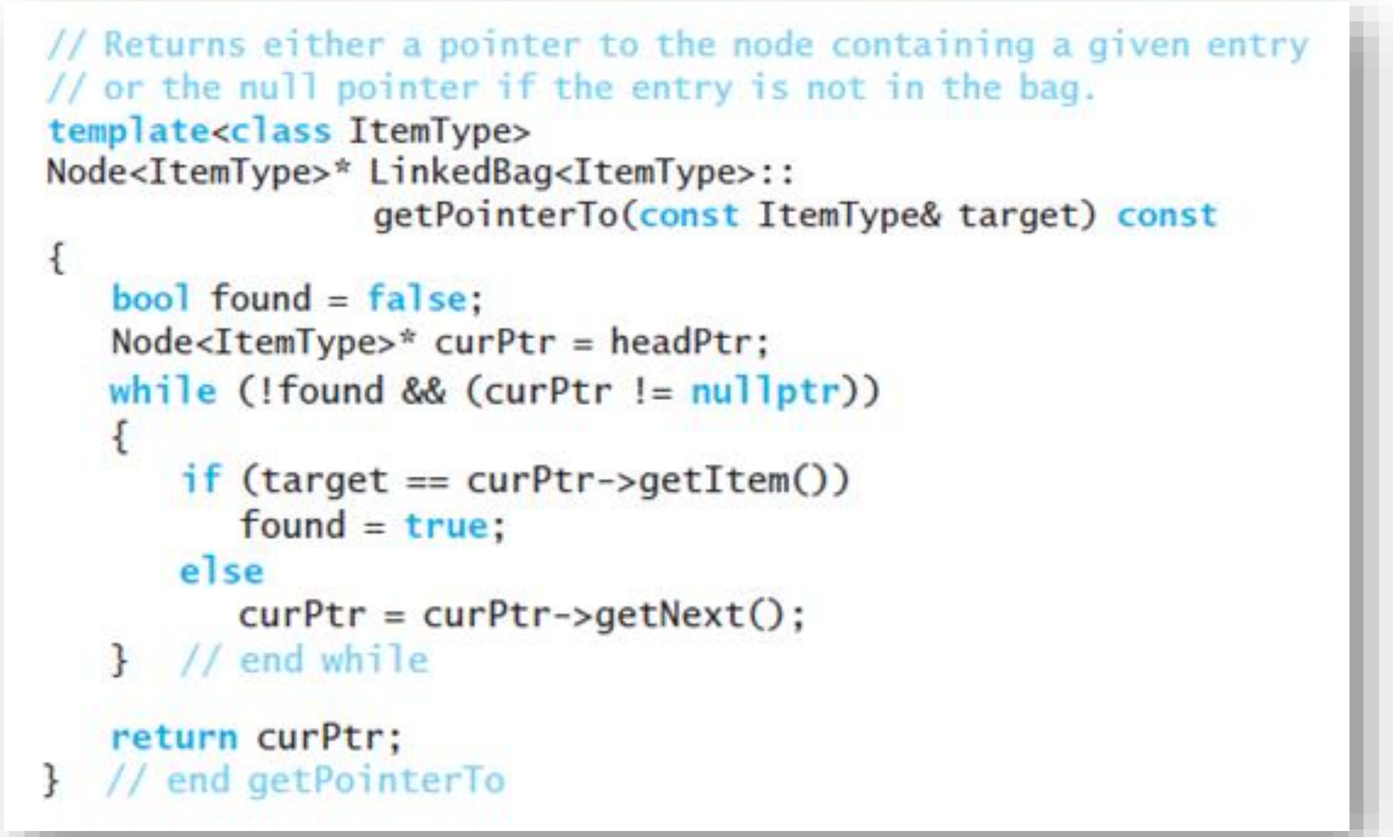
Implementing More Methods



```
template<class ItemType>
int LinkedBag<ItemType>::getFrequencyOf(const ItemType& anEntry) const
{
   int frequency = 0;
   int counter = 0;
   Node<ItemType>* curPtr = headPtr;
   while ((curPtr != nullptr) && (counter < itemCount))
   {
      if (anEntry == curPtr->getItem())
      {
         frequency++;
      } // end if

      counter ++;
      curPtr = curPtr->getNext();
   } // end while

   return frequency;
} // end getFrequencyOf
```

- Method getFrequencyOf

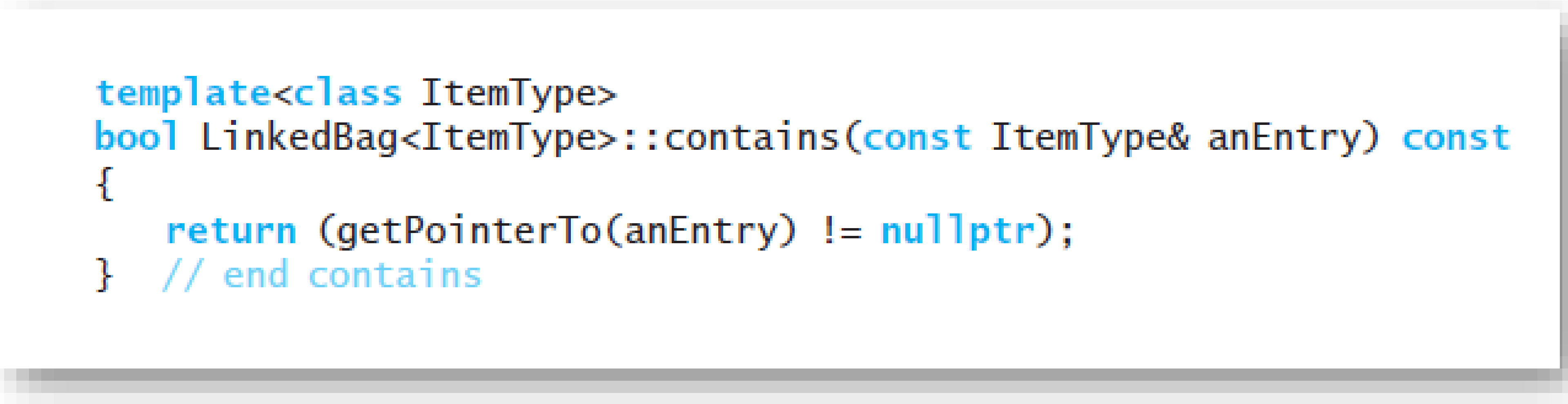
Implementing More Methods



```
// Returns either a pointer to the node containing a given entry
// or the null pointer if the entry is not in the bag.
template<class ItemType>
Node<ItemType>* LinkedBag<ItemType>::
                getPointerTo(const ItemType& target) const
{
   bool found = false;
   Node<ItemType>* curPtr = headPtr;
   while (!found && (curPtr != nullptr))
   {
      if (target == curPtr->getItem())
         found = true;
      else
         curPtr = curPtr->getNext();
   }  // end while

   return curPtr;
}  // end getPointerTo
```

- Search for a specific entry.
- To avoid duplicate code, we perform this search in a private method

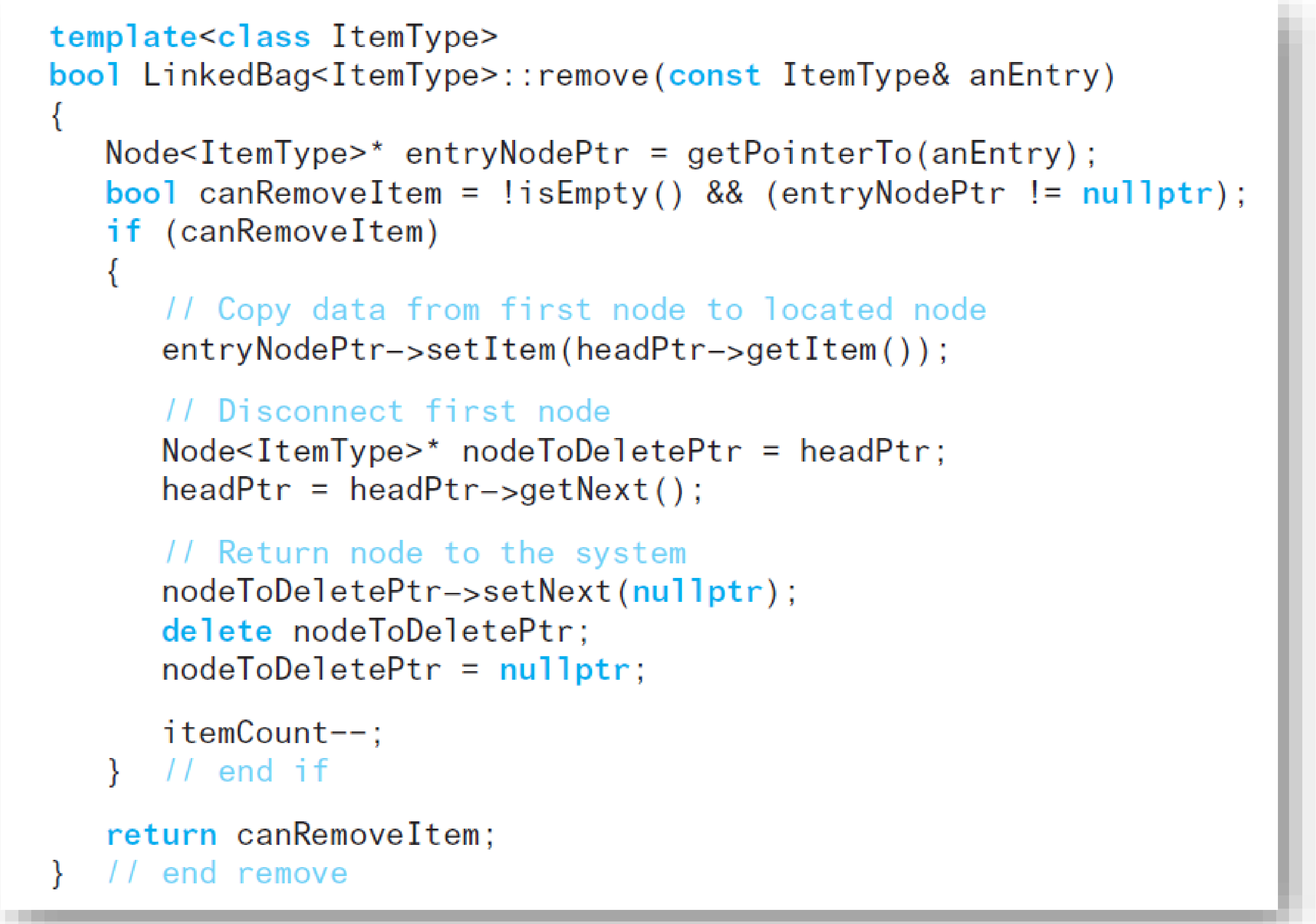
Implementing More Methods



```
template<class ItemType>
bool LinkedBag<ItemType>::contains(const ItemType& anEntry) const
{
   return (getPointerTo(anEntry) != nullptr);
}  // end contains
```

- Note: definition of the method contains calls getPointerTo

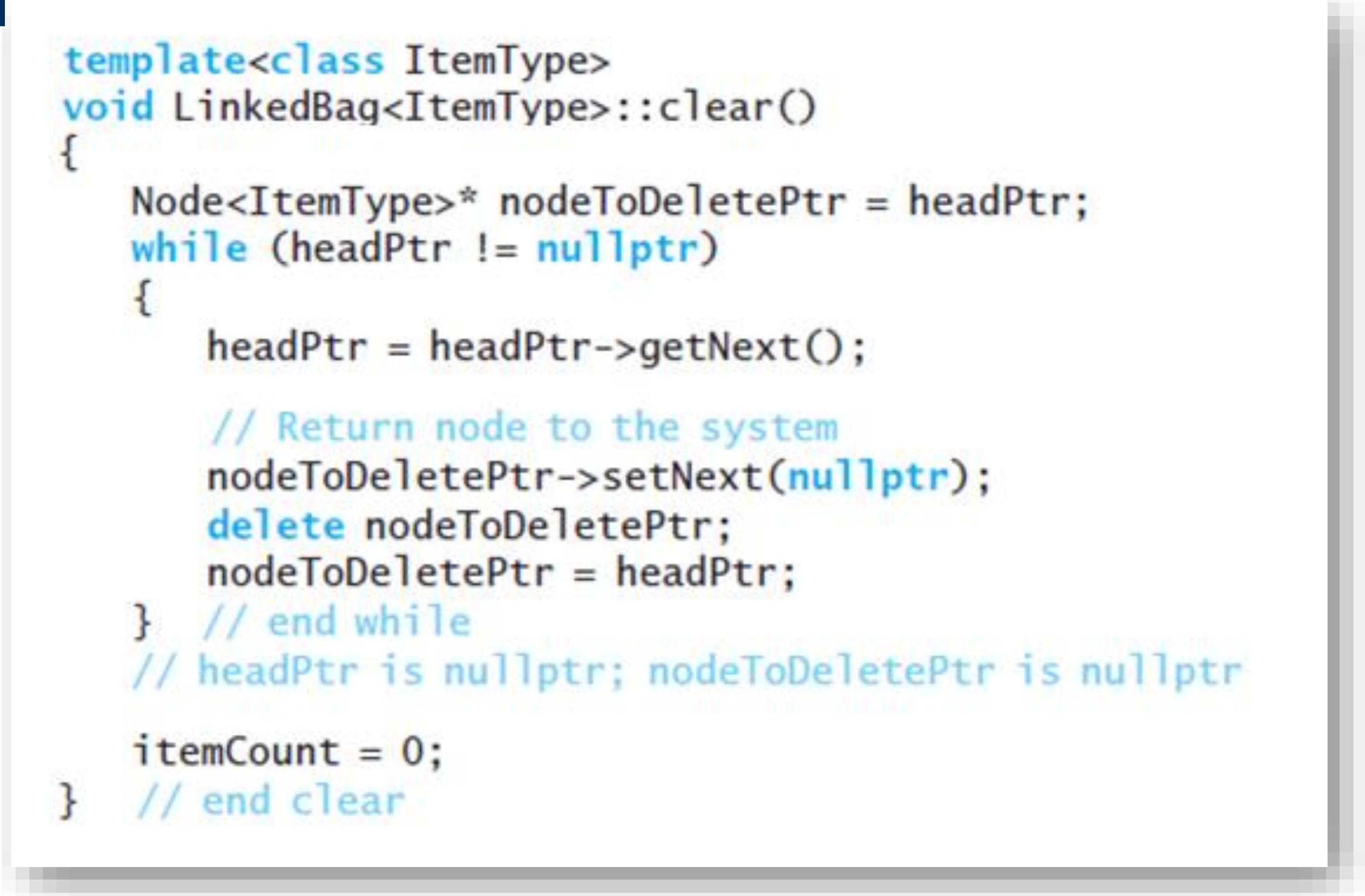
Implementing More Methods



```
template<class ItemType>
bool LinkedBag<ItemType>::remove(const ItemType& anEntry)
{
   Node<ItemType>* entryNodePtr = getPointerTo(anEntry);
   bool canRemoveItem = !isEmpty() && (entryNodePtr != nullptr);
   if (canRemoveItem)
   {
      // Copy data from first node to located node
      entryNodePtr->setItem(headPtr->getItem());

      // Disconnect first node
      Node<ItemType>* nodeToDeletePtr = headPtr;
      headPtr = headPtr->getNext();

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

      itemCount--;
   }  // end if

   return canRemoveItem;
}  // end remove
```

- Method remove also calls getPointerTo

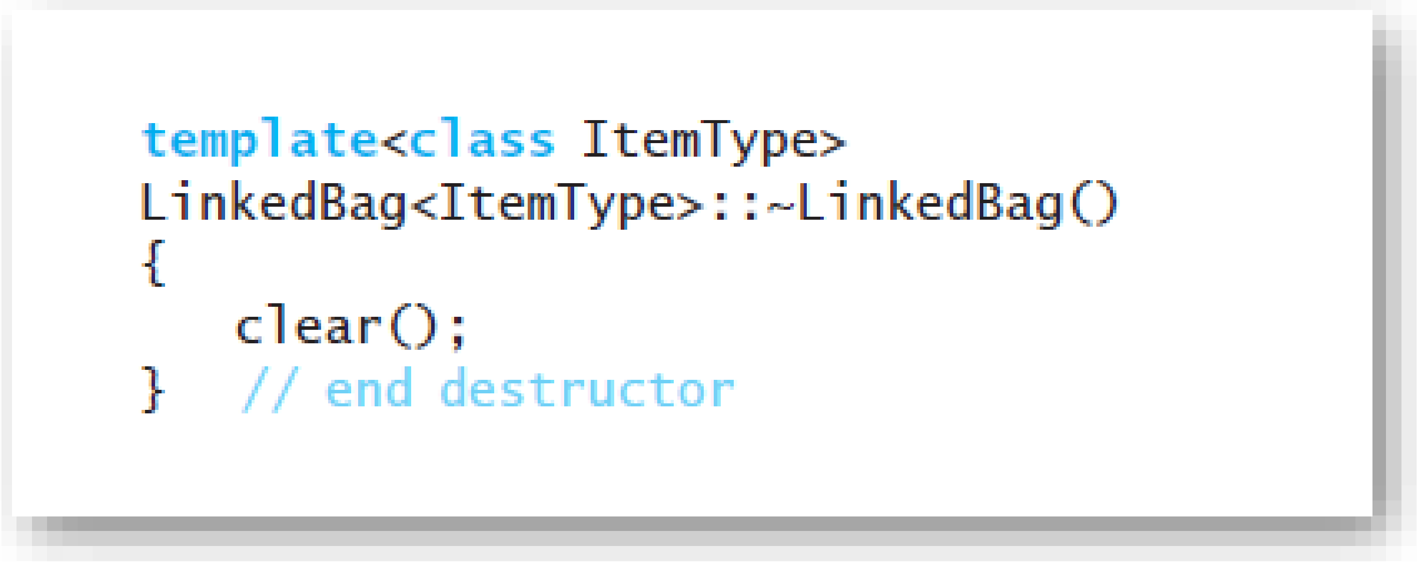
Implementing More Methods



```
template<class ItemType>
void LinkedBag<ItemType>::clear()
{
    Node<ItemType>* nodeToDeletePtr = headPtr;
    while (headPtr != nullptr)
    {
        headPtr = headPtr->getNext();

        // Return node to the system
        nodeToDeletePtr->setNext(nullptr);
        delete nodeToDeletePtr;
        nodeToDeletePtr = headPtr;
    }  // end while
    // headPtr is nullptr; nodeToDeletePtr is nullptr

    itemCount = 0;
}  // end clear
```

- Method clear deallocates all nodes in the chain

Implementing More Methods

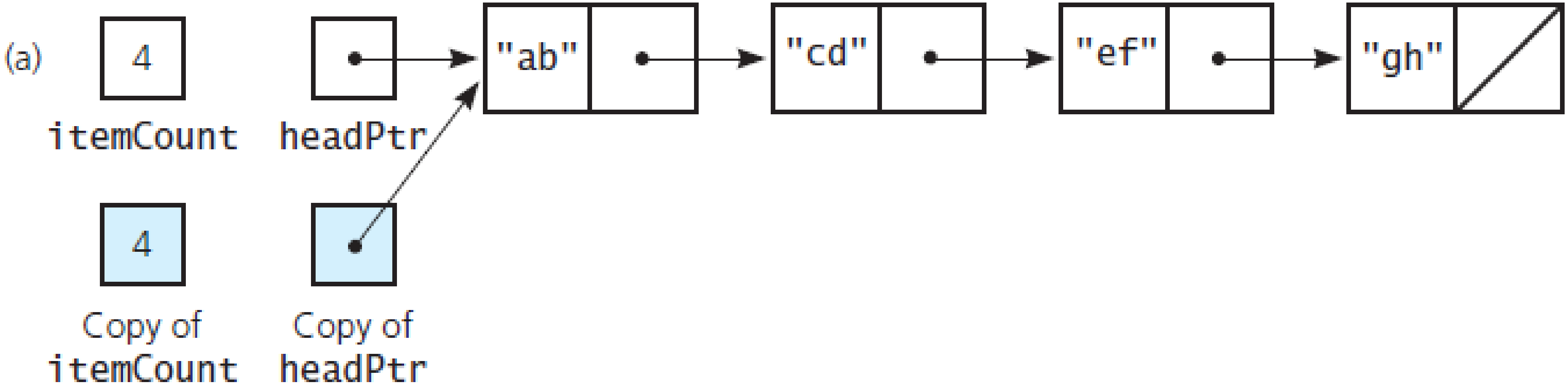


```
template<class ItemType>
LinkedBag<ItemType>::~LinkedBag()
{
   clear();
}  // end destructor
```

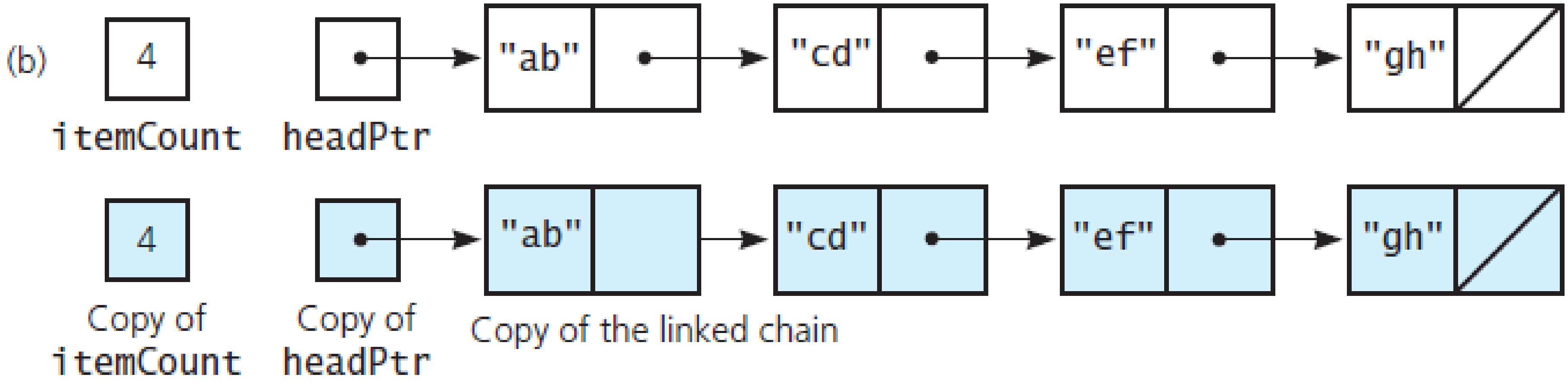
- Destructor calls clear, destroys instance of a class

Link-Based Implementation of ADT Bag

- (a) A linked chain and its shallow copy;

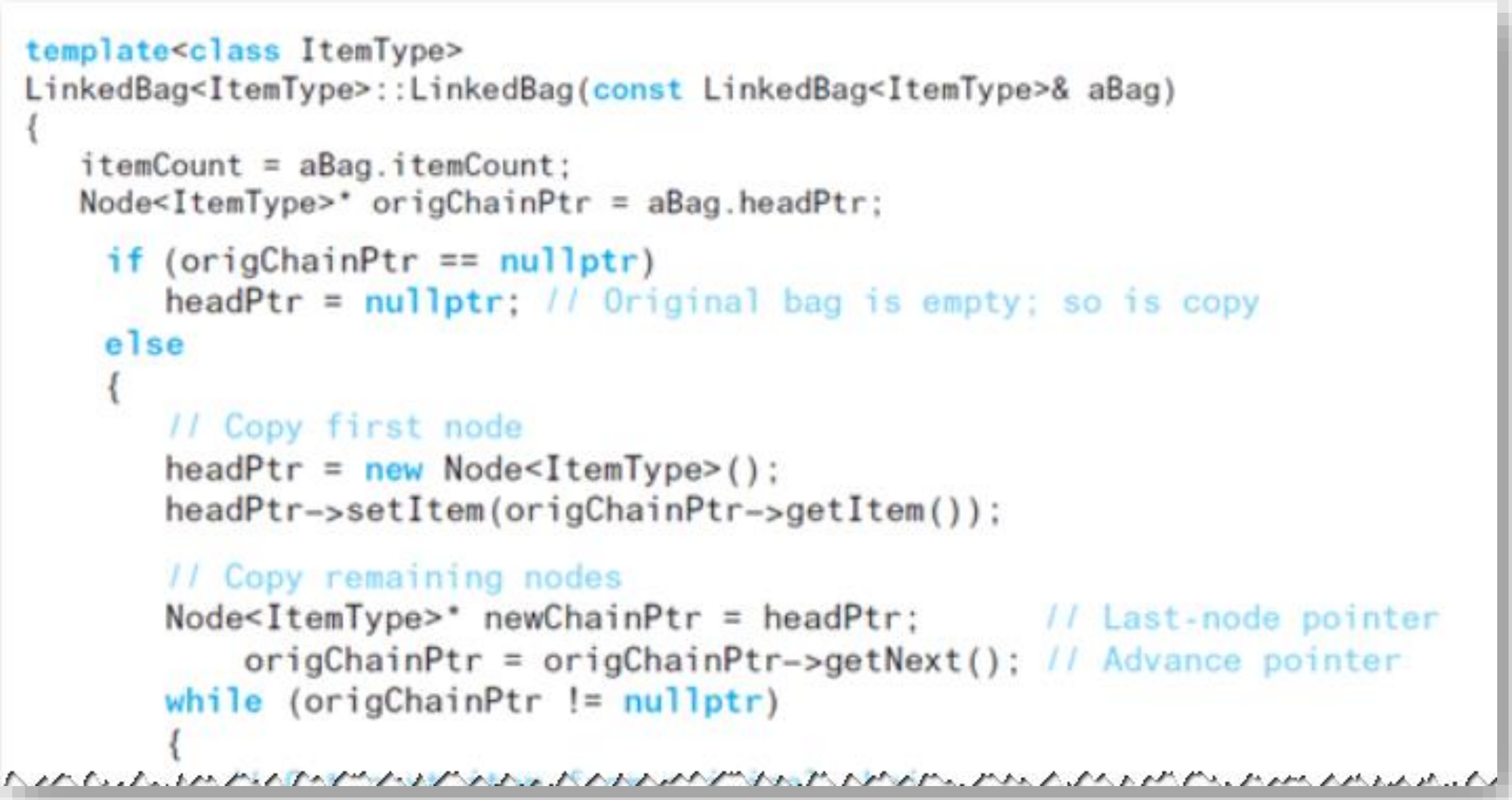


Testing Multiple ADT Implementations



(b) a linked chain and its deep copy

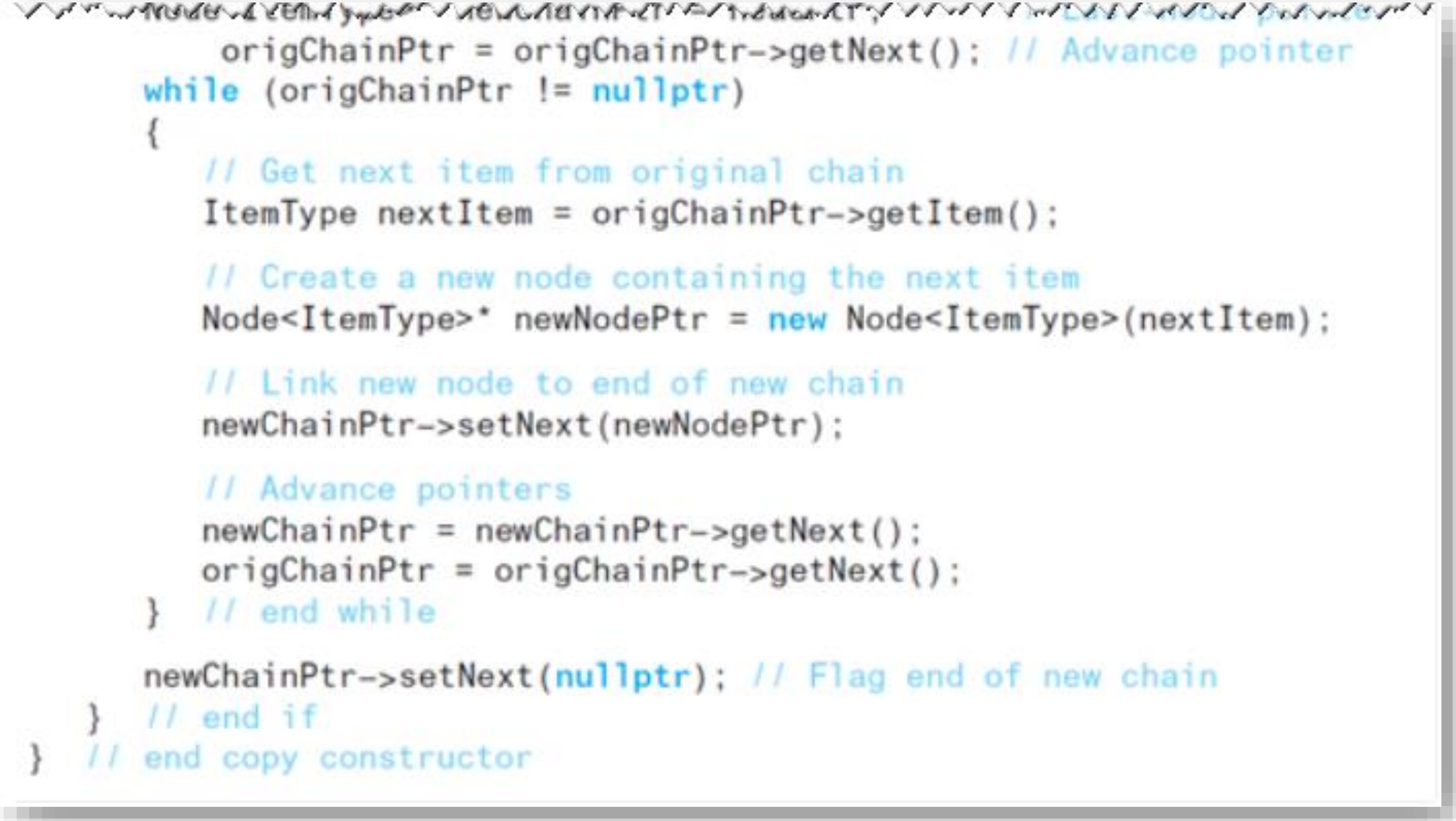
Implementing More Methods



```
template<class ItemType>
LinkedBag<ItemType>::LinkedBag(const LinkedBag<ItemType>& aBag)
{
   itemCount = aBag.itemCount;
   Node<ItemType>* origChainPtr = aBag.headPtr;

    if (origChainPtr == nullptr)
        headPtr = nullptr; // Original bag is empty; so is copy
    else
    {
        // Copy first node
        headPtr = new Node<ItemType>();
        headPtr->setItem(origChainPtr->getItem());

        // Copy remaining nodes
        Node<ItemType>* newChainPtr = headPtr;      // Last-node pointer
            origChainPtr = origChainPtr->getNext(); // Advance pointer
        while (origChainPtr != nullptr)
        {
```

- Copy constructor to accomplish deep copy.

Implementing More Methods



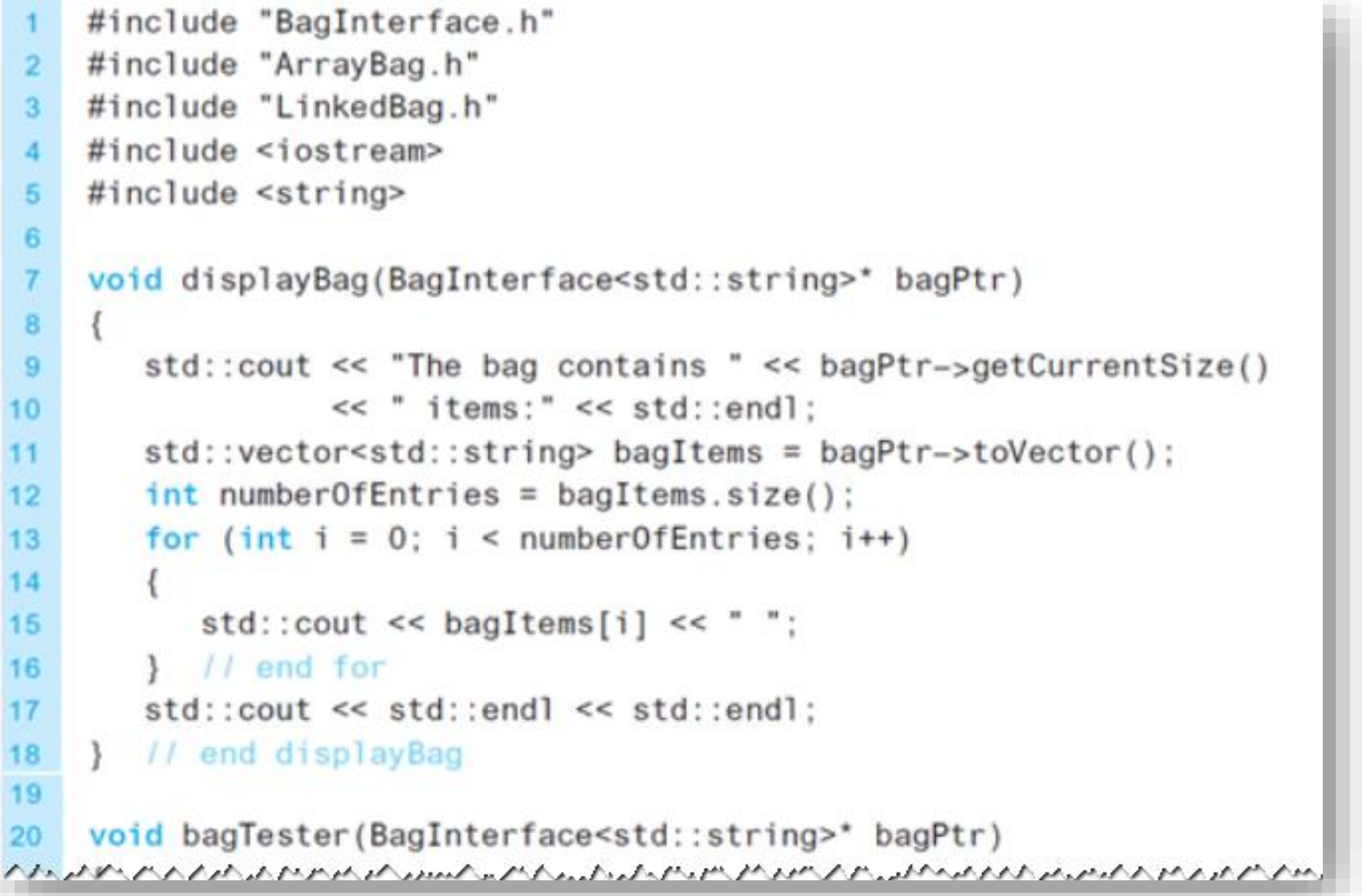
```
          origChainPtr = origChainPtr->getNext(); // Advance pointer
      while (origChainPtr != nullptr)
      {
         // Get next item from original chain
         ItemType nextItem = origChainPtr->getItem();

         // Create a new node containing the next item
         Node<ItemType>* newNodePtr = new Node<ItemType>(nextItem);

         // Link new node to end of new chain
         newChainPtr->setNext(newNodePtr);

         // Advance pointers
         newChainPtr = newChainPtr->getNext();
         origChainPtr = origChainPtr->getNext();
      }  // end while

      newChainPtr->setNext(nullptr); // Flag end of new chain
   }  // end if
}  // end copy constructor
```

- Copy constructor to accomplish deep copy.

Testing Multiple ADT implementations

- Used ADT bag methods when we tested our implementation
  - test program of Listing 3-2
- Can use the same code—with a few changes
  - Change each occurrence of ArrayBag to LinkedBag and recompile the program

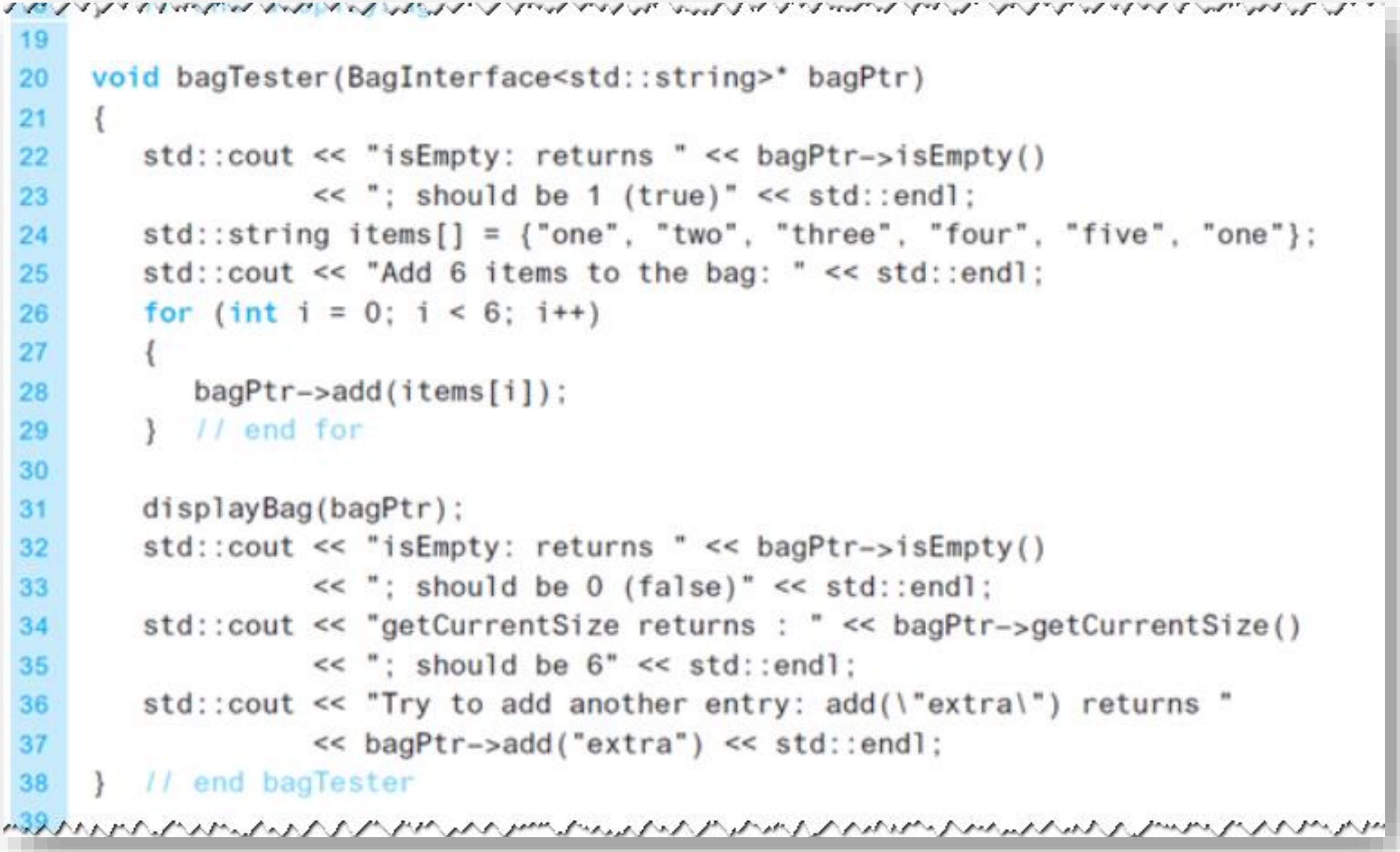
Testing Multiple ADT Implementations



```cpp
#include "BagInterface.h"
#include "ArrayBag.h"
#include "LinkedBag.h"
#include <iostream>
#include <string>

void displayBag(BagInterface<std::string>* bagPtr)
{
   std::cout << "The bag contains " << bagPtr->getCurrentSize()
             << " items:" << std::endl;
   std::vector<std::string> bagItems = bagPtr->toVector();
   int numberOfEntries = bagItems.size();
   for (int i = 0; i < numberOfEntries; i++)
   {
      std::cout << bagItems[i] << " ";
   }  // end for
   std::cout << std::endl << std::endl;
}  // end displayBag

void bagTester(BagInterface<std::string>* bagPtr)
```

- A program that tests the core methods of classes that are derived from the abstract class BagInterface

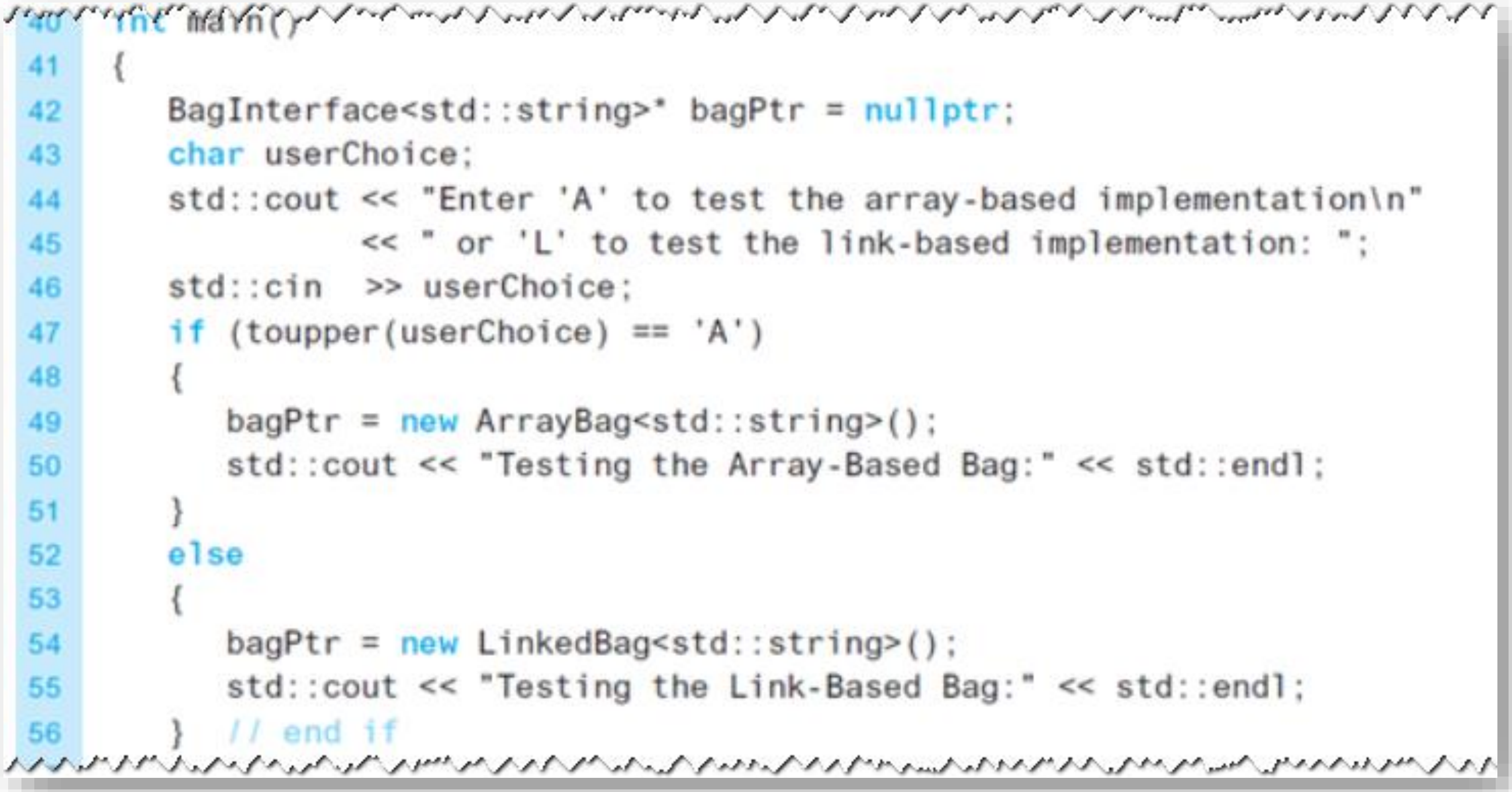
Testing Multiple ADT Implementations



```cpp
void bagTester(BagInterface<std::string>* bagPtr)
{
    std::cout << "isEmpty: returns " << bagPtr->isEmpty()
              << "; should be 1 (true)" << std::endl;
    std::string items[] = {"one", "two", "three", "four", "five", "one"};
    std::cout << "Add 6 items to the bag: " << std::endl;
    for (int i = 0; i < 6; i++)
    {
        bagPtr->add(items[i]);
    }  // end for

    displayBag(bagPtr);
    std::cout << "isEmpty: returns " << bagPtr->isEmpty()
              << "; should be 0 (false)" << std::endl;
    std::cout << "getCurrentSize returns : " << bagPtr->getCurrentSize()
              << "; should be 6" << std::endl;
    std::cout << "Try to add another entry: add(\"extra\") returns "
              << bagPtr->add("extra") << std::endl;
}  // end bagTester
```

- A program that tests the core methods of classes that are derived from the abstract class BagInterface

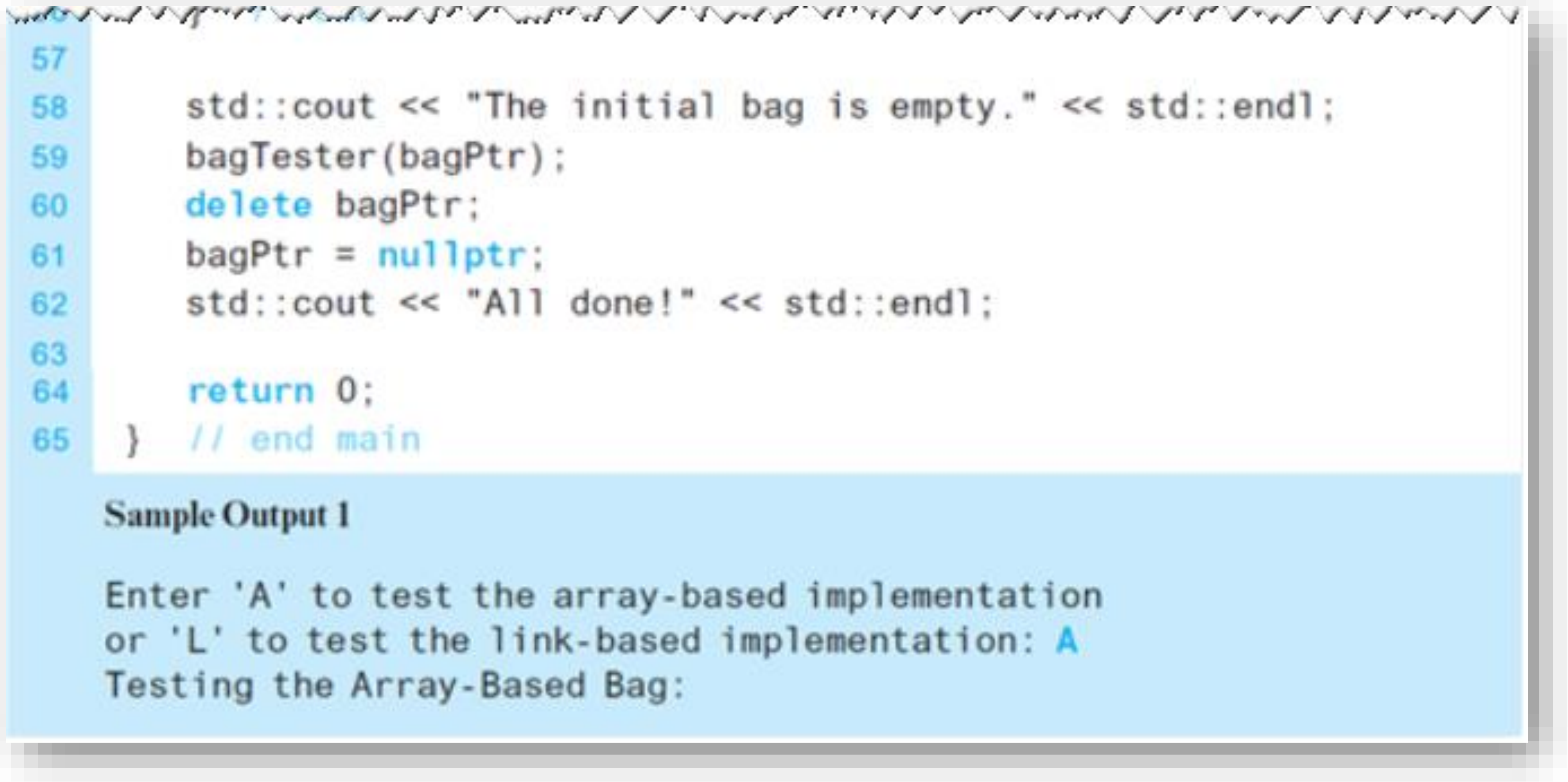
Testing Multiple ADT Implementations



```
int main()
{
   BagInterface<std::string>* bagPtr = nullptr;
   char userChoice;
   std::cout << "Enter 'A' to test the array-based implementation\n"
             << " or 'L' to test the link-based implementation: ";
   std::cin  >> userChoice;
   if (toupper(userChoice) == 'A')
   {
      bagPtr = new ArrayBag<std::string>();
      std::cout << "Testing the Array-Based Bag:" << std::endl;
   }
   else
   {
      bagPtr = new LinkedBag<std::string>();
      std::cout << "Testing the Link-Based Bag:" << std::endl;
   }  // end if
```

A program that tests the core methods of classes that are derived from the abstract class BagInterface

Testing Multiple ADT Implementations



```
   std::cout << "The initial bag is empty." << std::endl;
   bagTester(bagPtr);
   delete bagPtr;
   bagPtr = nullptr;
   std::cout << "All done!" << std::endl;

   return 0;
}  // end main
```

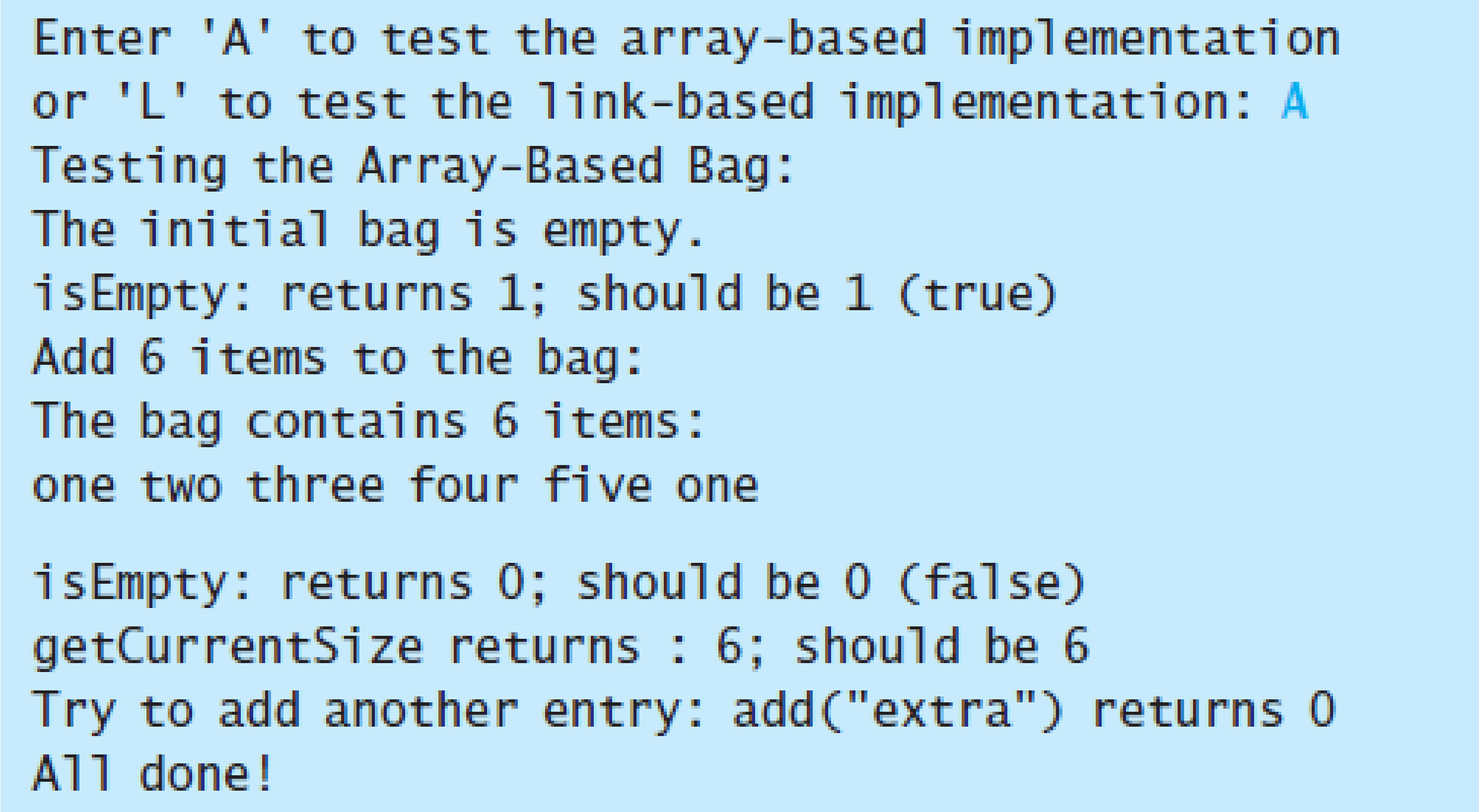
Sample Output 1

```
Enter 'A' to test the array-based implementation
or 'L' to test the link-based implementation: A
Testing the Array-Based Bag:
```

A program that tests the core methods of classes that are derived from the abstract class BagInterface

Testing Multiple ADT Implementations

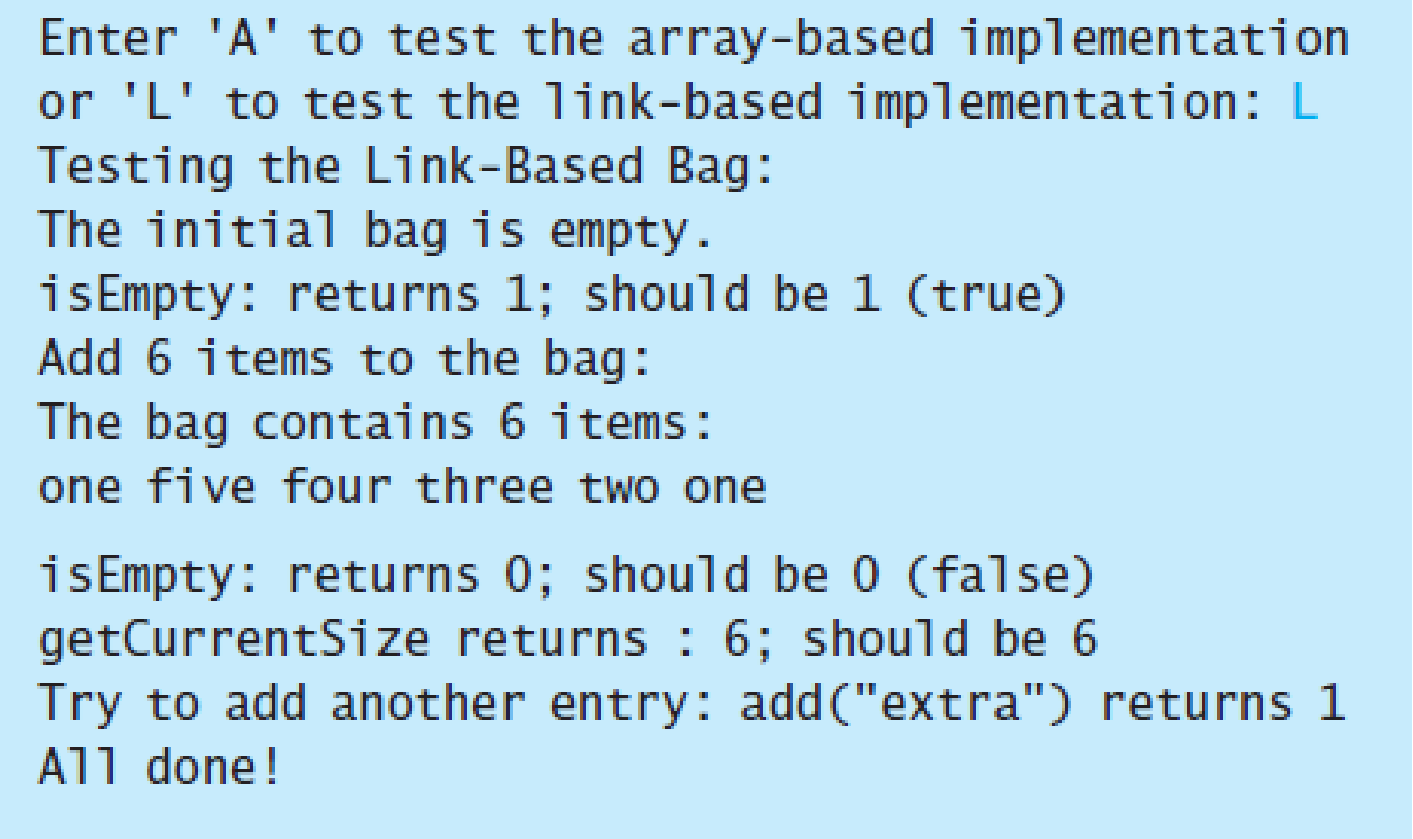
Sample output 1 of test program



```
Enter 'A' to test the array-based implementation
or 'L' to test the link-based implementation: A
Testing the Array-Based Bag:
The initial bag is empty.
isEmpty: returns 1; should be 1 (true)
Add 6 items to the bag:
The bag contains 6 items:
one two three four five one

isEmpty: returns 0; should be 0 (false)
getCurrentSize returns : 6; should be 6
Try to add another entry: add("extra") returns 0
All done!
```

Testing Multiple ADT Implementations

Sample output 2 of test program



```
Enter 'A' to test the array-based implementation
or 'L' to test the link-based implementation: L
Testing the Link-Based Bag:
The initial bag is empty.
isEmpty: returns 1; should be 1 (true)
Add 6 items to the bag:
The bag contains 6 items:
one five four three two one

isEmpty: returns 0; should be 0 (false)
getCurrentSize returns : 6; should be 6
Try to add another entry: add("extra") returns 1
All done!
```

Comparing Array-Based and Link-Based Implementations

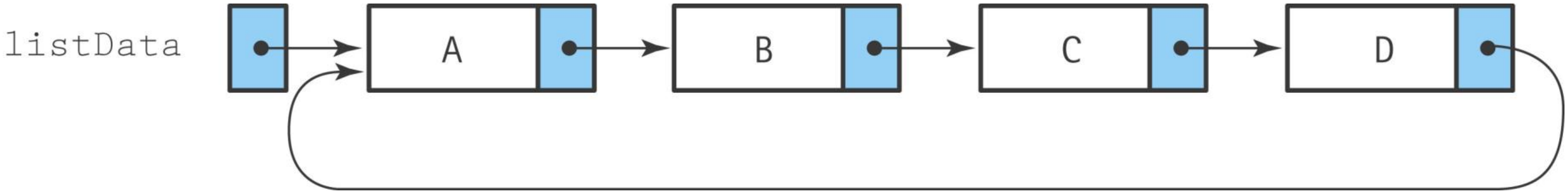
- Arrays easy to use, but have fixed size
  - Not always easy to predict number of items in ADT
  - Array could waste space
  - Can be dynamically resized
- Increasing size of dynamically allocated array can waste storage and time
- Can access array items directly with equal access time
  - Dynamic arrays are better when direct access of items is frequent

Comparing Array-Based and Link-Based Implementations

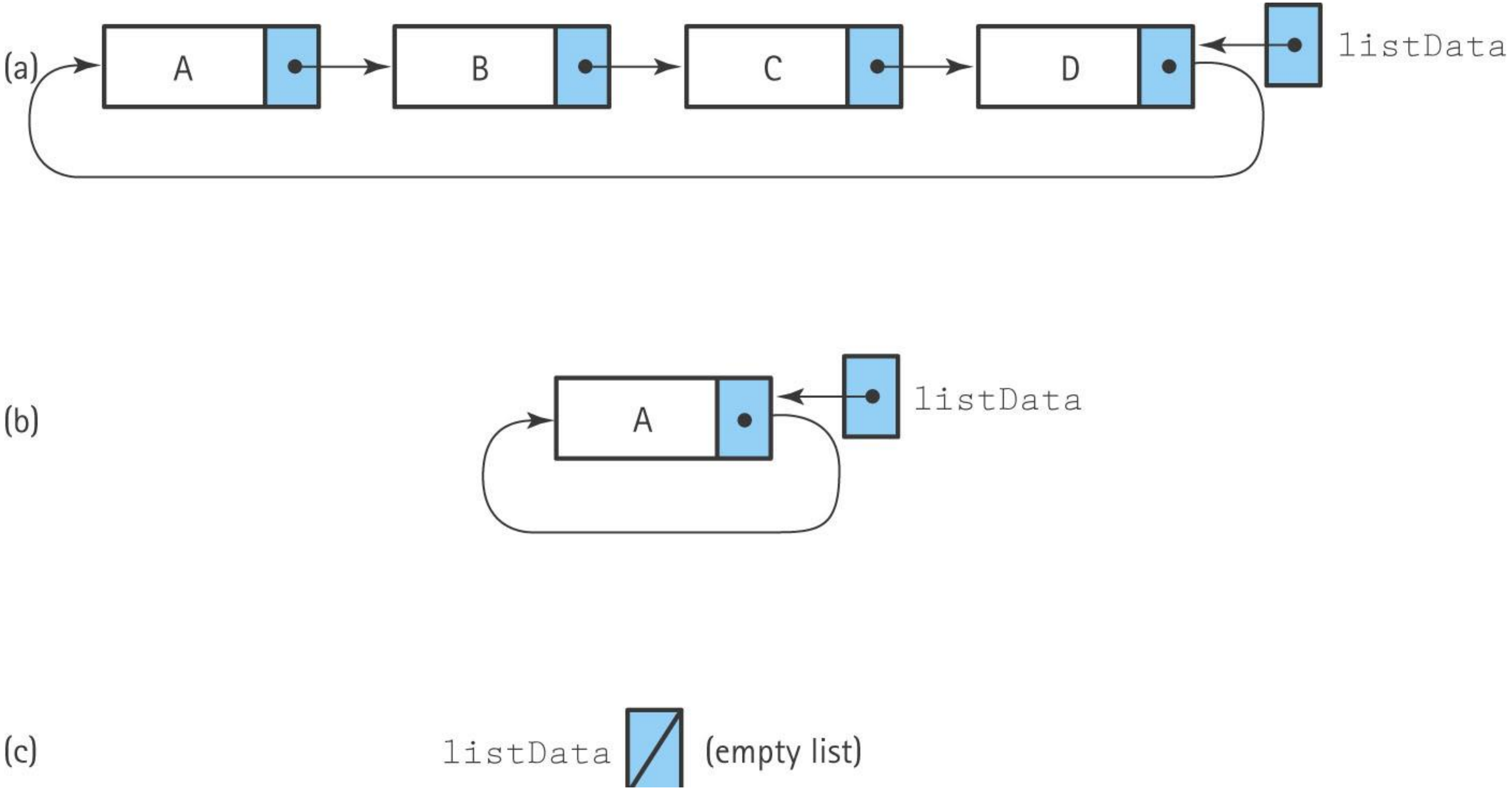
- Linked chains do not have fixed size
  - In a chain of linked nodes, an item points explicitly to the next item
  - Link-based implementation requires more memory
- Array items accessed directly, equal access time
  - Must traverse linked chain for i^{th} item
    - access time varies

What is a Circular Linked List?

- A circular linked list is a list in which every node has a successor; the “last” element is succeeded by the “first” element.

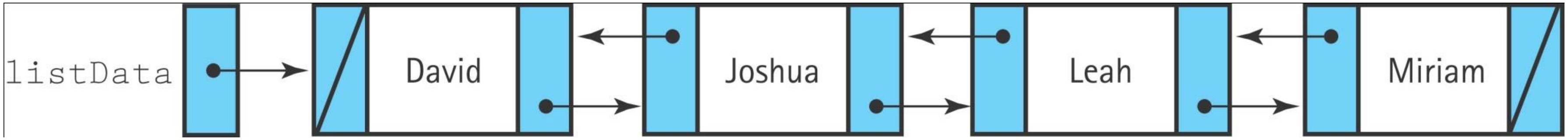


External Pointer to the Last Node



What is a Doubly Linked List?

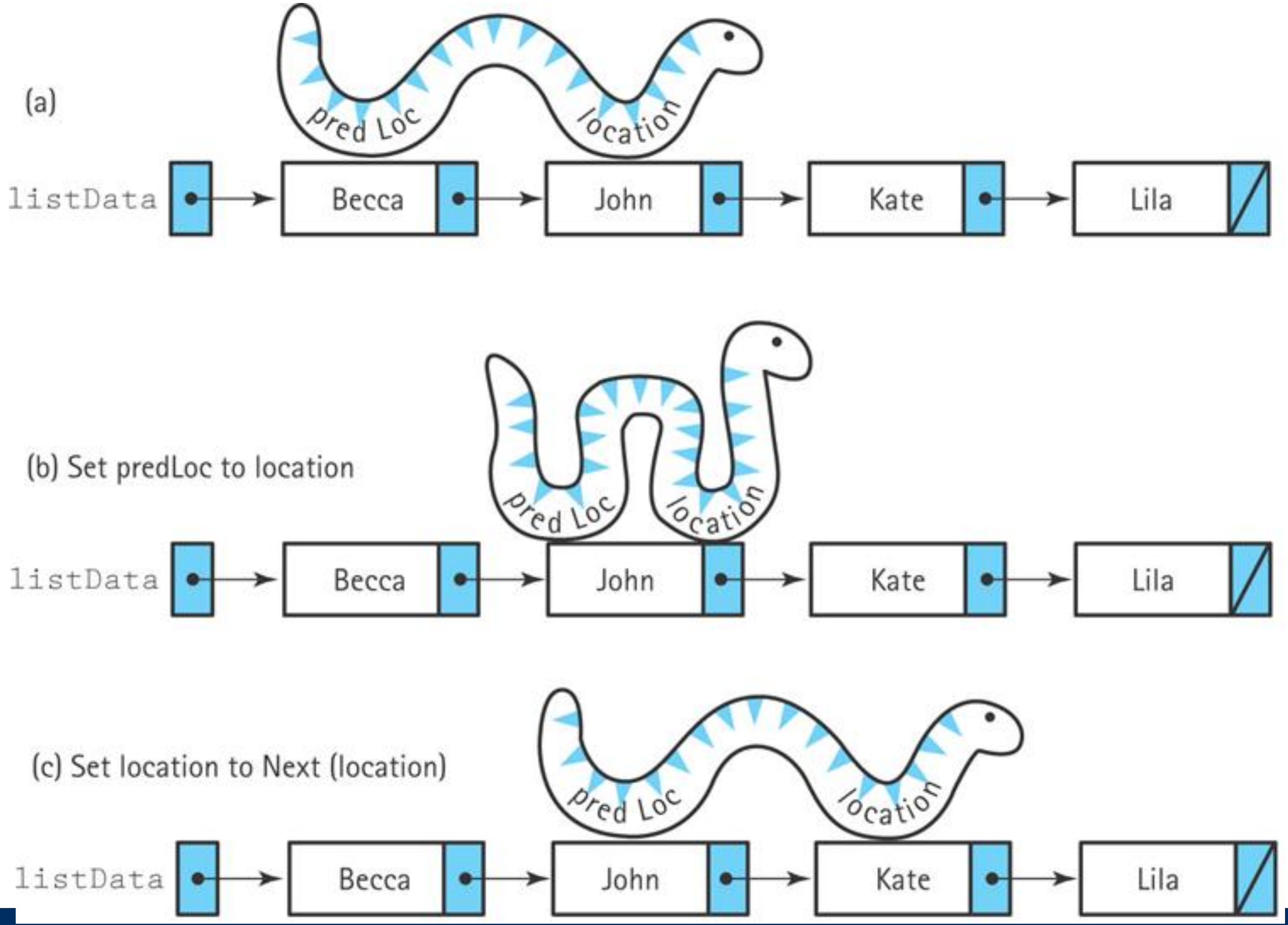
- A doubly linked list is a list in which each node is linked to both its successor and its predecessor.



Insert Item algorithm for

- Find proper position for the new element in the sorted list using two pointers predLoc and location, where predLoc trails behind location.
- Obtain a node for insertion and place item in it.
- Insert the node by adjusting pointers.
- Increment length.

The Inchworm Effect

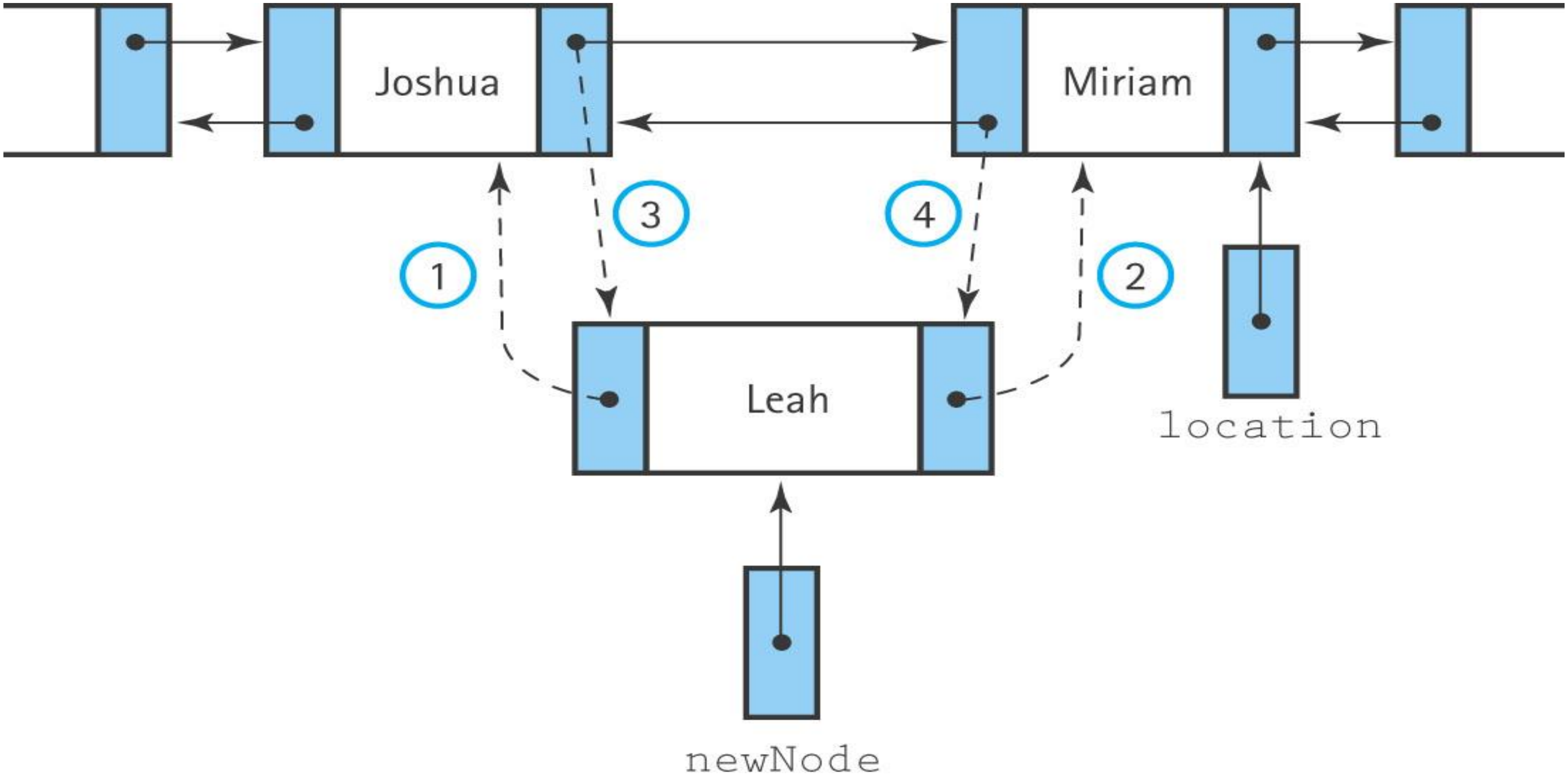


(a)

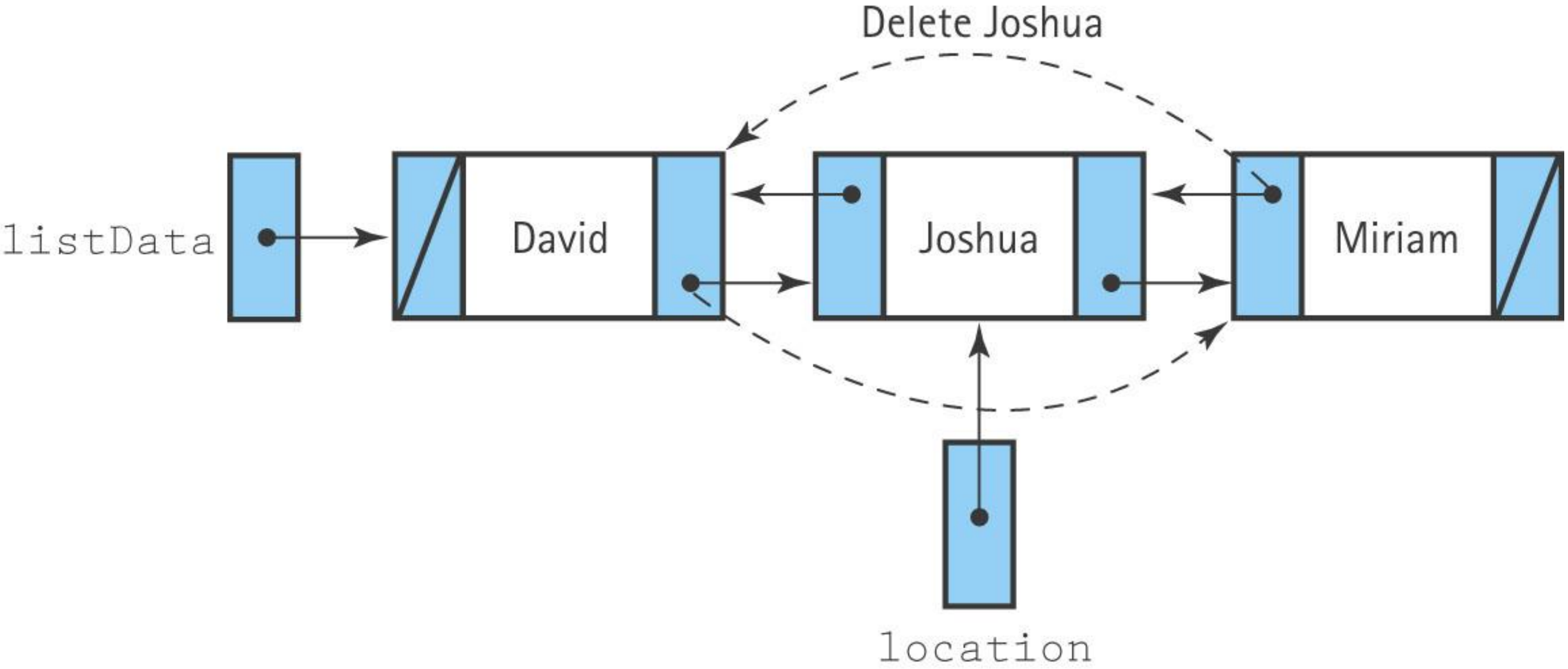
(b) Set predLoc to location

(c) Set location to Next (location)

Linking the New Node into the List



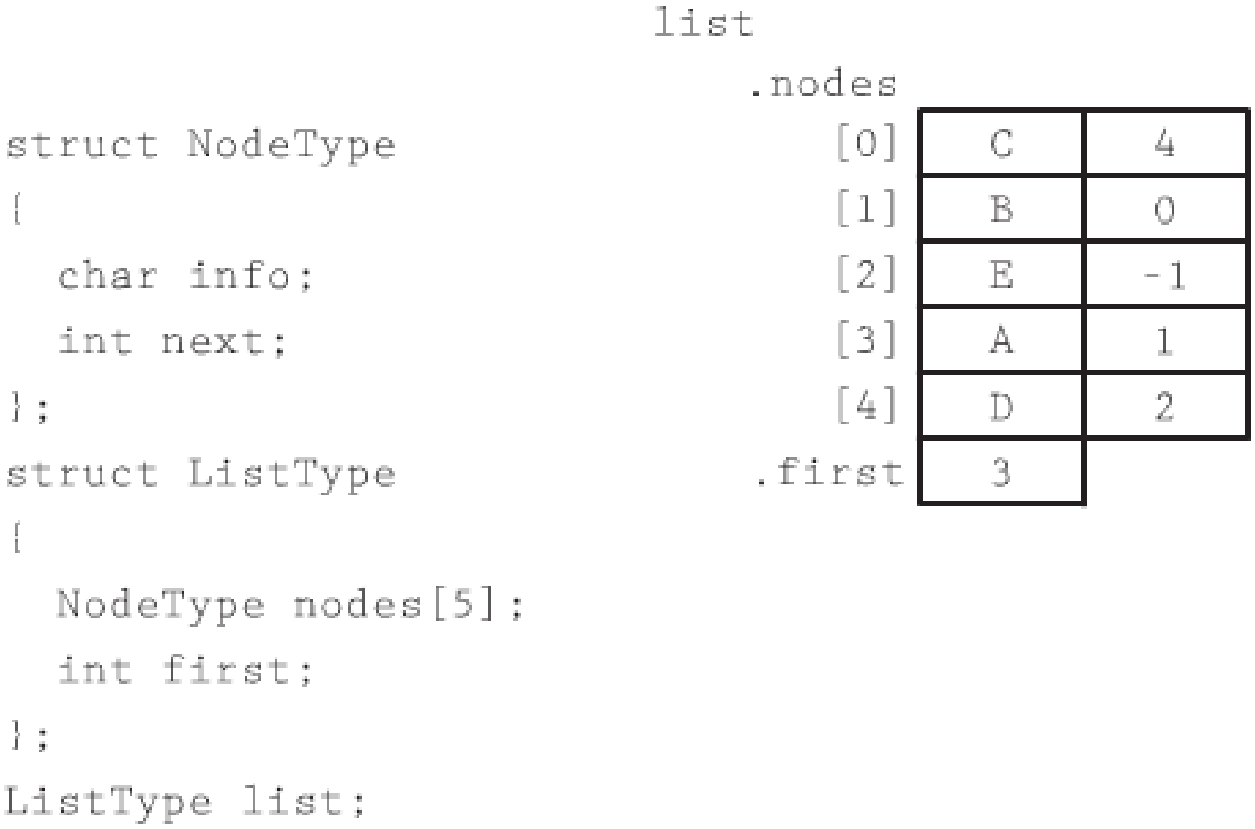
Deleting from a Doubly Linked List



What are the advantages of a circular doubly linked list?

A linked list in static storage ?

A Linked List as an Array of Records



```
struct NodeType
{
   char info;
   int next;
};
struct ListType
{
   NodeType nodes[5];
   int first;
};
ListType list;
```

list

.nodes

[0]	C	4
[1]	B	0
[2]	E	-1
[3]	A	1
[4]	D	2
.first	3	

Out of the Box

Letter Combinations of a Phone Number
https://leetcode.com/problems/letter-combinations-of-a-phone-number/description/

Roman to Integer
https://leetcode.com/problems/roman-to-integer/description/