Linked-Based Implementation

Tiziana Ligorio
tligorio@hunter.cuny.edu

Today's Plan

Recap

A quick review of pointers

Linked-Based Implementation

Announcements and Syllabus Check

Almost back on track (w.r.t. tentative schedule)!!!

No TA tutoring on Wednesday and Thursday this week

Google Info Session!!!

Google software engineers will be visiting Hunter College in October to meet with computer science students about how to prepare for a career in tech!

if you're interested in attending, please RSVP here by Monday, Oct. 1

Detailed info:

https://googleinfosession.splashthat.com/

https://docs.google.com/forms/d/e/1FAIpQLSerMq19yz2K9iyE8-GKCnsyG31vM9_zZ3OwV_UiDCTz5KipyA/viewform

RSVP: http://bit.ly/googlehunterinfosession

Pointers Review

Pointer Variables

A typed variable whose value is the address of another variable of same type

```
int x = 5;
int y = 8;
int *p, *q = nullptr;  //declares two int pointers


. . .
p = &x;       // sets p to the address of x
q = &y;       // sets q address of y
```

Make sure you do this if not assigning a value!

We won't do much of this

Run-time Stack

Type	Name	Address	Data
...	...	...	...
int	x	0x12345670	5
int	y	0x12345674	8
int pointer	p	0x12345678	0x12345670
int pointer	q	0x1234567C	0x12345674
...	...	...	...

Dynamic Variables

Created at runtime in the free store or memory heap
using operator **new**

Nameless typed variables accessed through pointers

```
// create a nameless variable of type dataType on the
//application heap and stores its address in p
dataType *p = new dataType;
```

Run-time Stack

Type	Name	Address	Data
...	...	...	...
dataType ptr	p	0x12345678	0x100436f20
...	...	...	...

Free Store (application heap)

Type	Address	Data
...	...	...
dataType	0x100436f20	
...	...	...

Accessing members

```
dataType some_object;
dataType *p = new dataType;
// initialize and do stuff with some_object

. . .

string my_string = some_object.getName();
string another_string = p->getStringData();
```

To access member functions in place of . operator

Deallocating Memory

```
delete p;
p = nullptr;
```

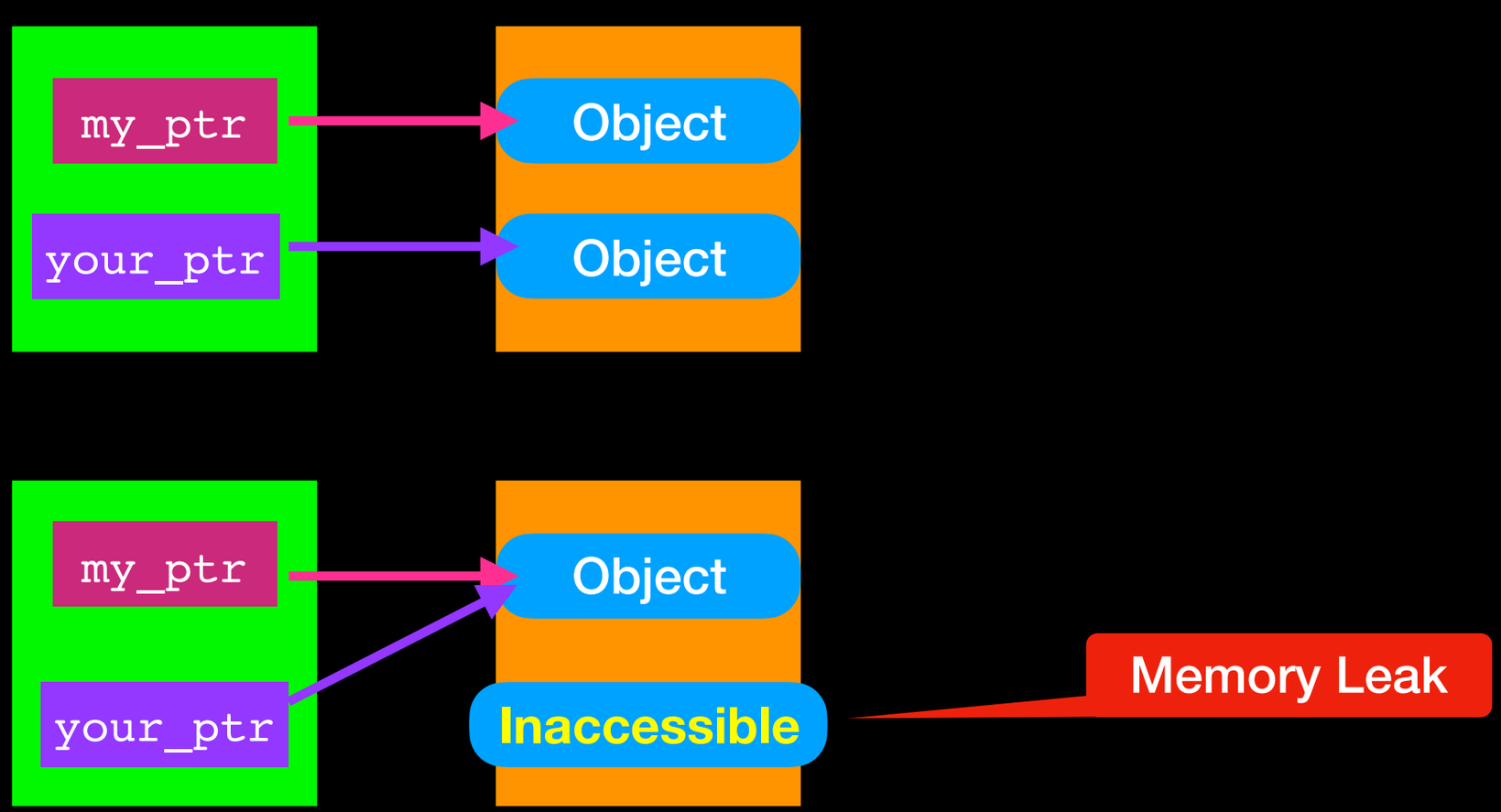
Must do this!!!

Avoid Memory Leaks

Occurs when object is created in free store but program no longer has access to it

```
dataType *my_ptr = new dataType;
dataType *your_ptr = new dataType;
// do stuff with my_ptr and your_ptr
```

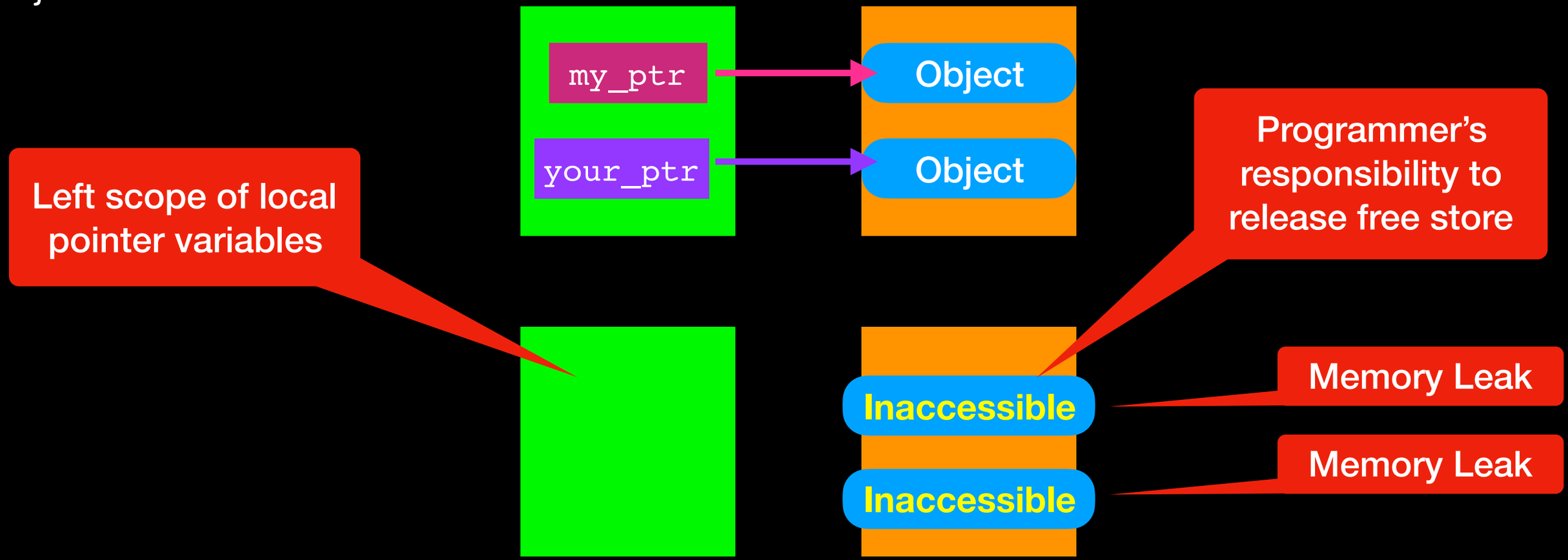
```
your_ptr = my_ptr;
```



Avoid Memory Leaks

Occurs when object is created in free store but program no longer has access to it

```
void leakyFunction(){
dataType *my_ptr = new dataType;
dataType *your_ptr = new dataType;
// do stuff with my_ptr and your_ptr
}
```

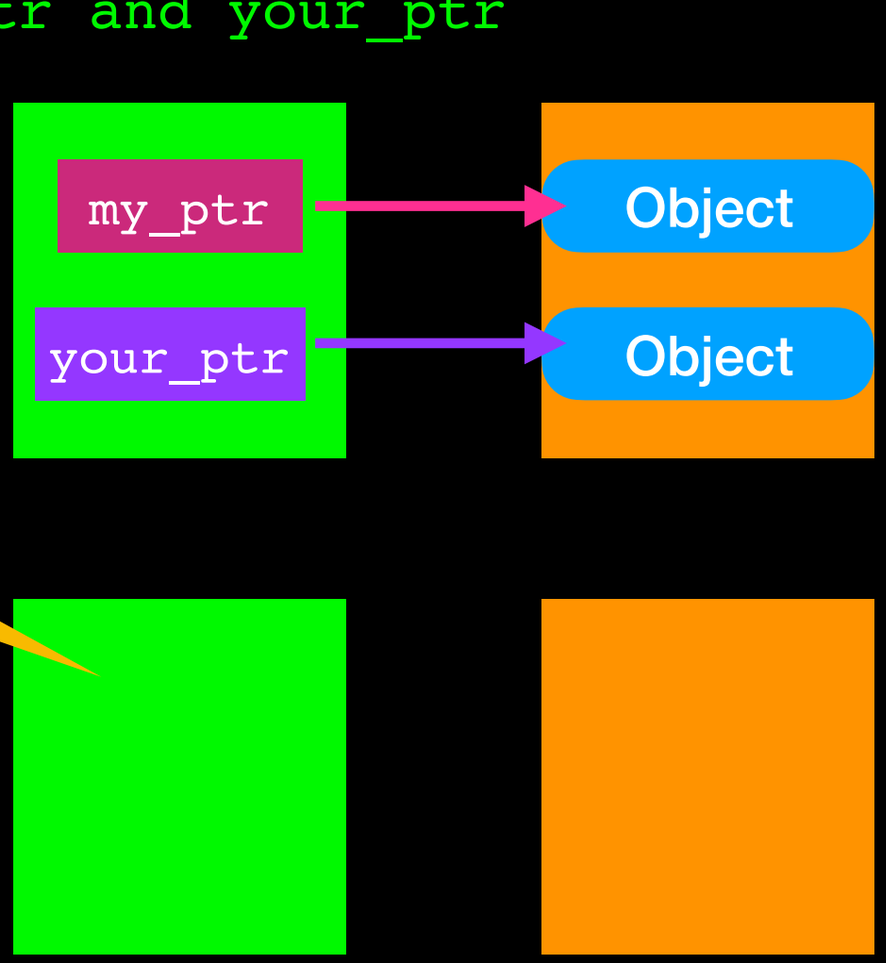


Avoid Memory Leaks

Occurs when object is created in free store but program no longer has access to it

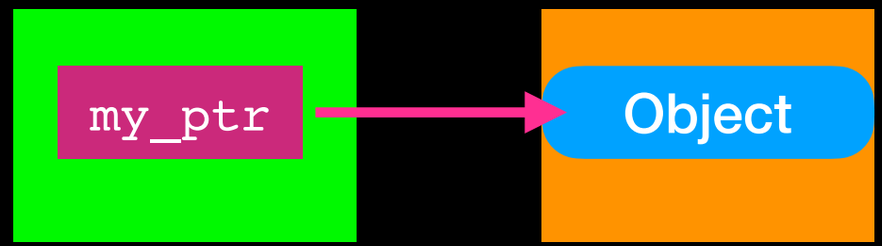
```
void leakyFunction(){
dataType *my_ptr = new dataType;
dataType *your_ptr = new dataType;
// do stuff with my_ptr and your_ptr
delete my_ptr;
delete your_ptr;
}
```

Left scope of local pointer variables

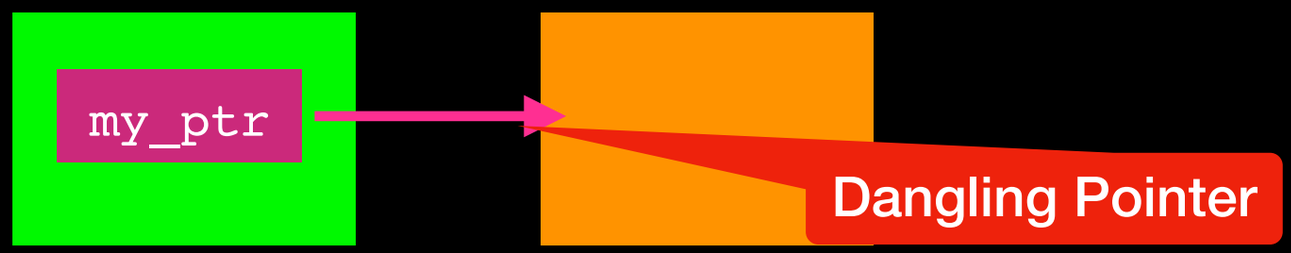


Avoid Dangling Pointers

Pointer variable that no longer references a valid object

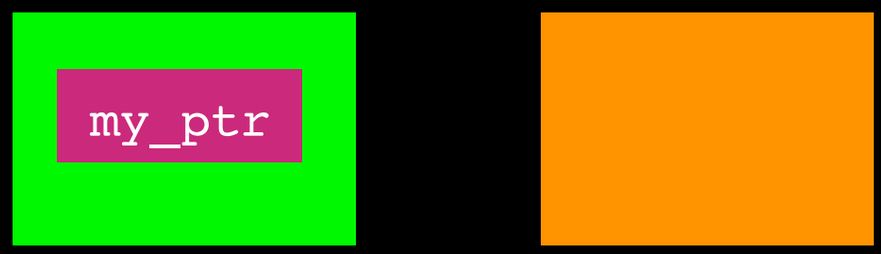



```
delete my_ptr;
```



```
delete my_ptr;
my_ptr = nullptr;
```

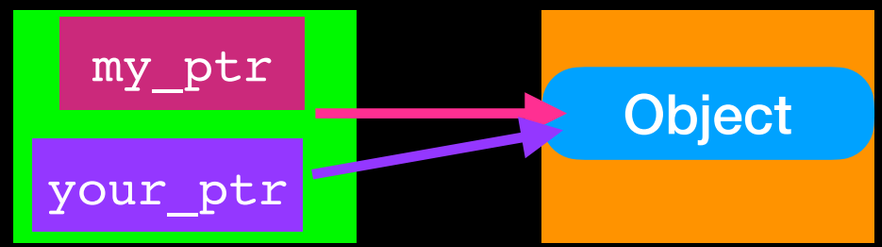
Must do this!!!

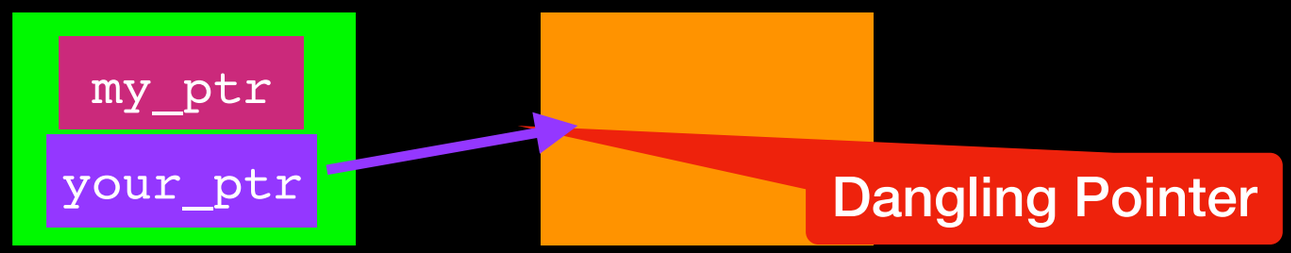


Avoid Dangling Pointers

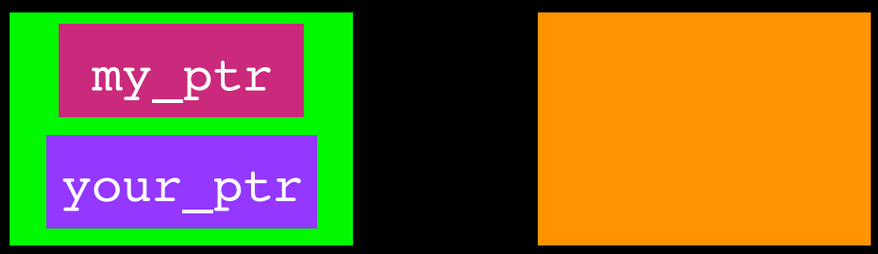
Pointer variable that no longer references a valid object

```
delete my_ptr;
my_ptr = nullptr;
```





```
delete my_ptr;
my_ptr = nullptr;
your_ptr = nullptr;
```



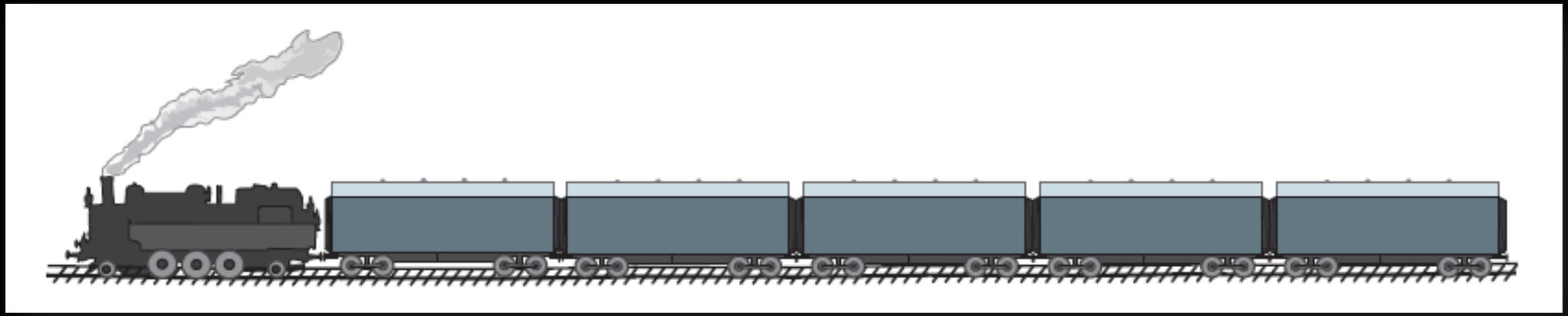
Must set all pointers to `nullptr`!!!

Link-Based Implementation

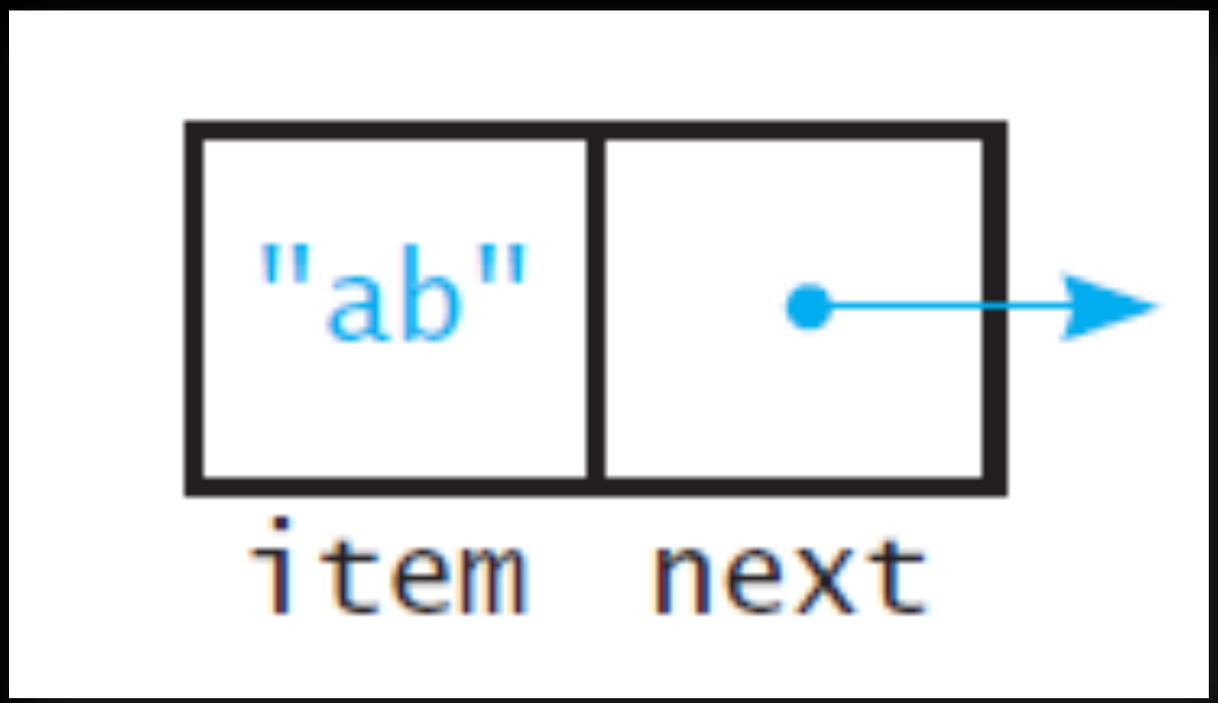
Data Organization

Place data within a Node object

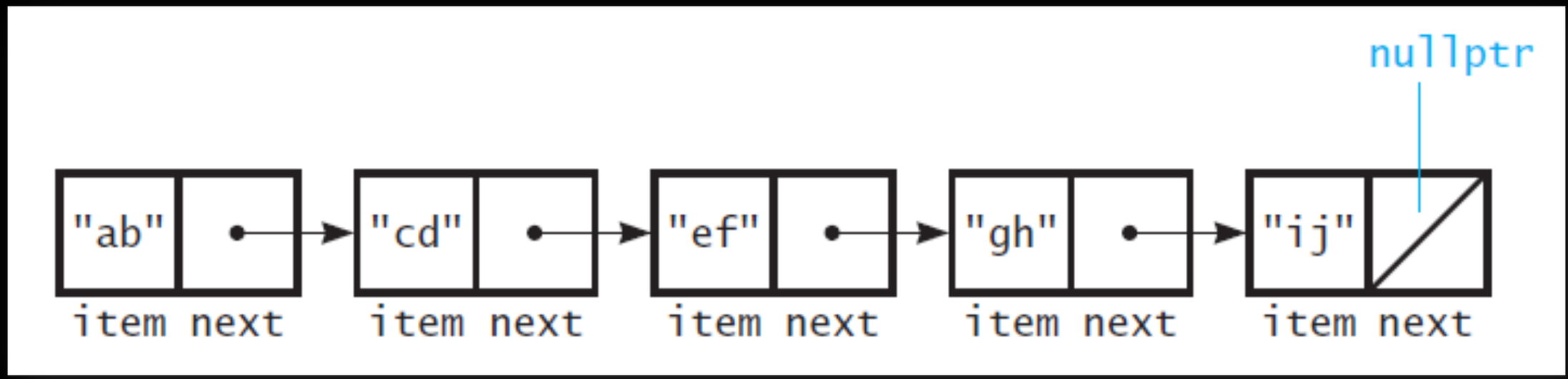
Link nodes into a chain



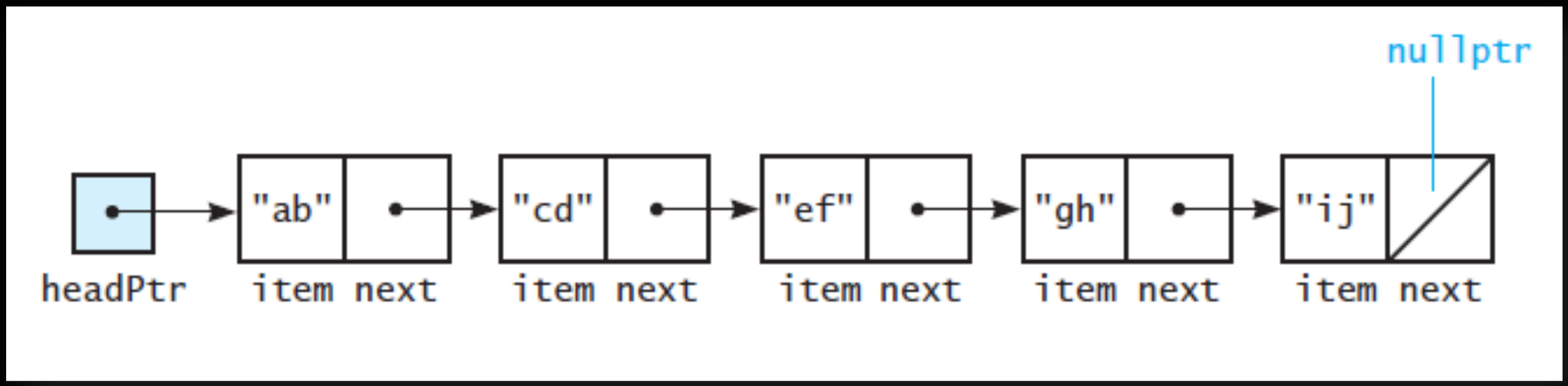
Node



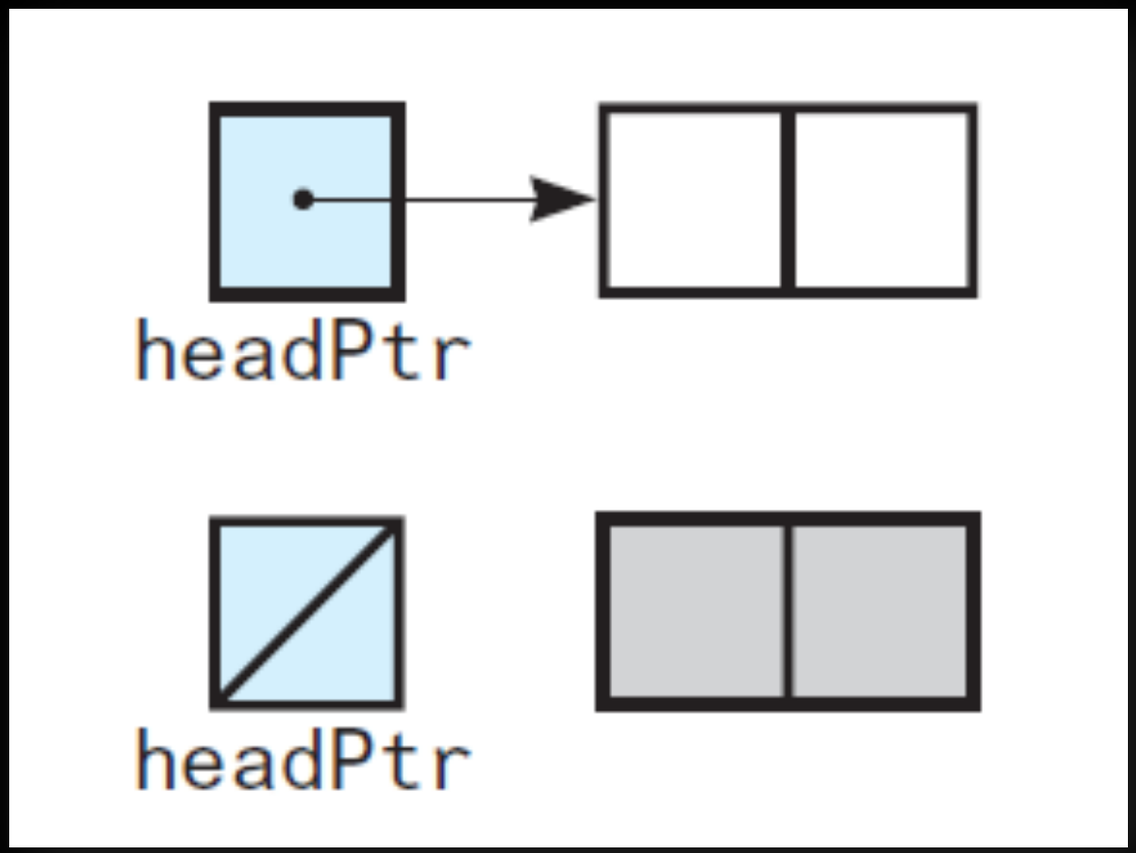
Chain


nullptr
"ab"
"cd"
"ef"
"gh"
"ij"
item next
item next
item next
item next
item next

Entering the Chain

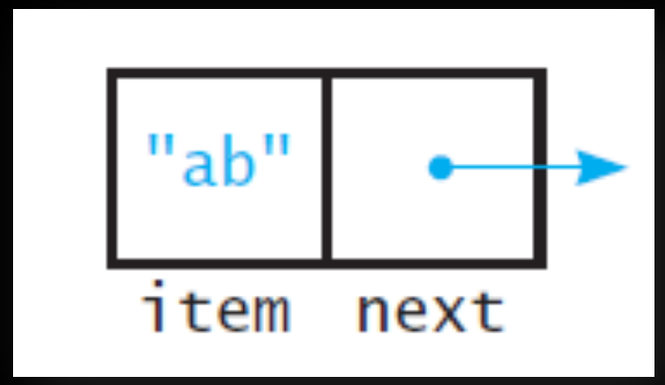


The Empty Chain



The Class Node

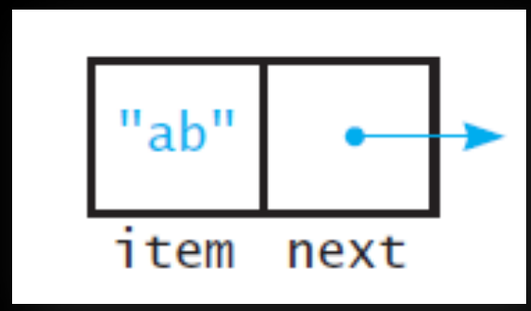
```
#ifndef NODE_H_
#define NODE_H_

template<class ItemType>
class 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;

private:
    ItemType item;          // A data item
    Node<ItemType>* next;  // Pointer to next node
}; // end Node



#include "Node.cpp"
#endif // NODE_H_
```



Node Implementation

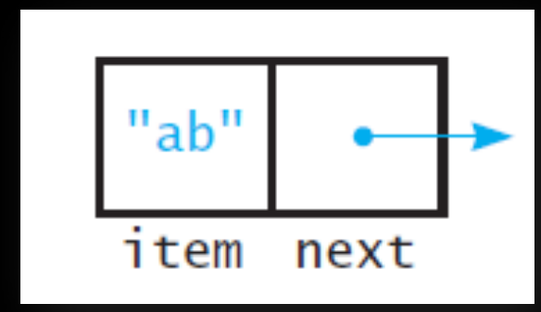
The Constructors

```
#include "Node.h"


template<class ItemType>
Node<ItemType>::Node() : next_(nullptr)
{

} // end default constructor


template<class ItemType>
Node<ItemType>::Node(const ItemType& an_item) : item_(an_item), next_(nullptr)
{

} // end constructor


template<class ItemType>
Node<ItemType>::Node(const ItemType& an_item, Node<ItemType>* next_node_ptr) :
                item_(an_item), next_(next_node_ptr)
{

} // end constructor
```



Node Implementation

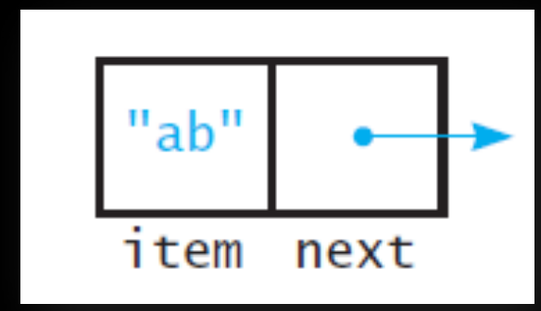
The "*setData*" members




```
#include "Node.h"


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

   item_ = an_item;

} // end setItem



template<class ItemType>
void Node<ItemType>::setNext(Node<ItemType>* next_node_ptr)
{

   next_ = next_node_ptr;

} // end setNext
```

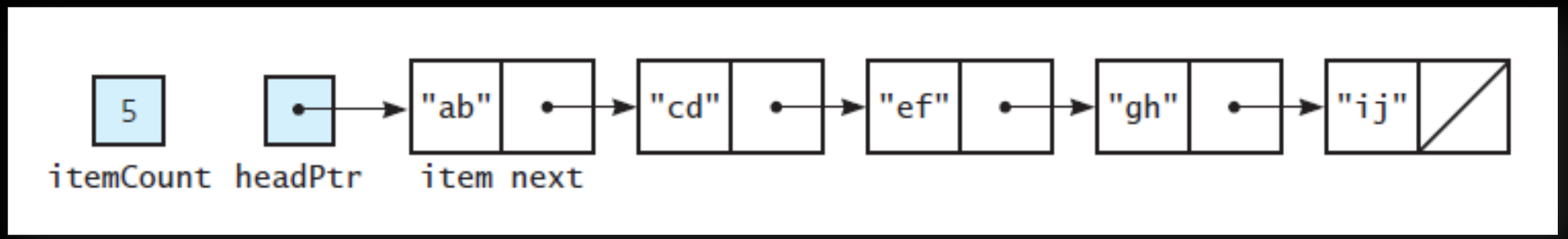
Node Implementation

The "get*Data*" members




```
#include "Node.h"


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
```

A Linked Bag ADT




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

The Class LinkedBag

```
#ifndef LINKED_BAG_H_
#define LINKED_BAG_H_

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

template<class ItemType>
class LinkedBag : public BagInterface<ItemType>
{
public:
   LinkedBag();
   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;
   std::vector<ItemType> toVector() const;

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;
}; // end LinkedBag

#include “LinkedBag.cpp"
#endif //LINKED_BAG_H_
```

More than one public methods will need to know the a pointer to a target so we separate it out into a private helper function (similar to ArrayBag but here we get pointers rather than indices)

LinkedBag Implementation

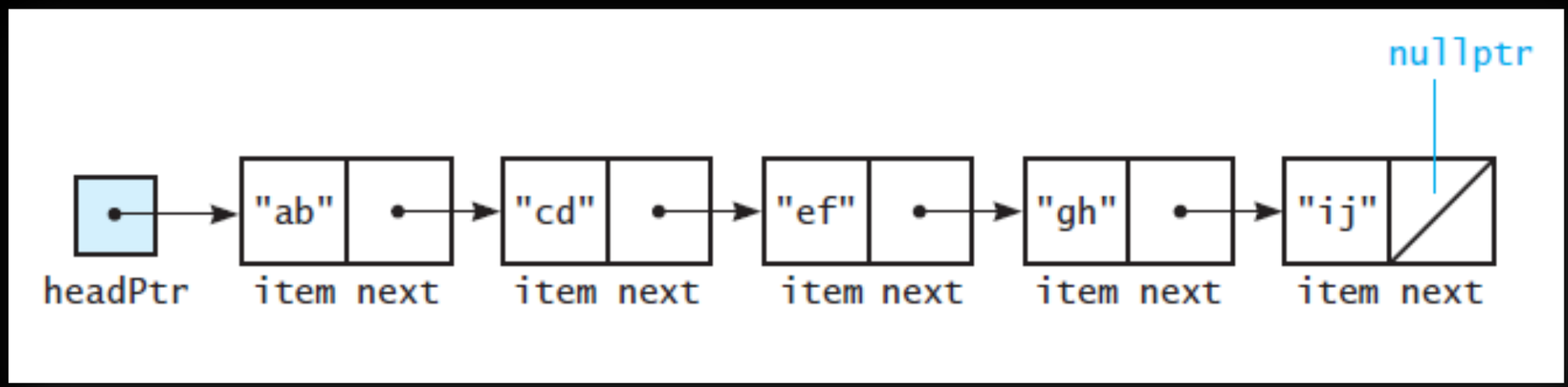
```
#include "LinkedBag.h"



template<class ItemType>
LinkedBag<ItemType>::LinkedBag() : headPtr(nullptr),
itemCount(0)
{

}  // end default constructor
```

The default constructor

Private data member initialization

In-Class Task

Write a sequence of steps (pseudocode) to add to the front of the chain

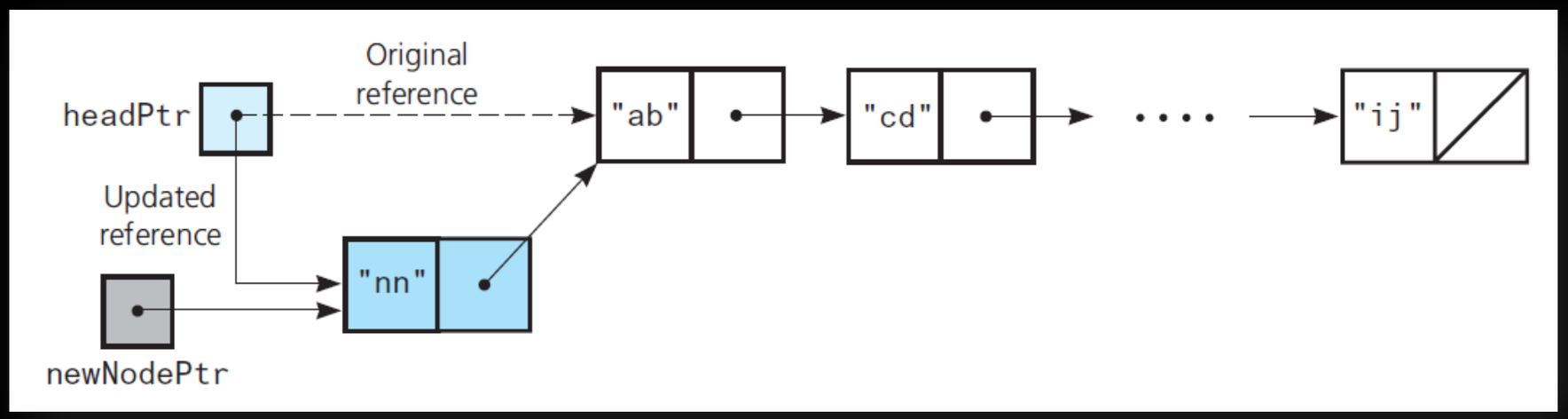


LinkedBag Implementation

```
#include "LinkedBag.h"


template<class ItemType>
bool LinkedBag<ItemType>::add(const ItemType& newEntry)
{
   // Add to beginning of chain: new node references rest of chain;
   // (headPtr is null 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
```

The `add` method
Add at beginning of chain is easy because we have headPtr

Dynamic memory allocation
Adding nodes to the heap!

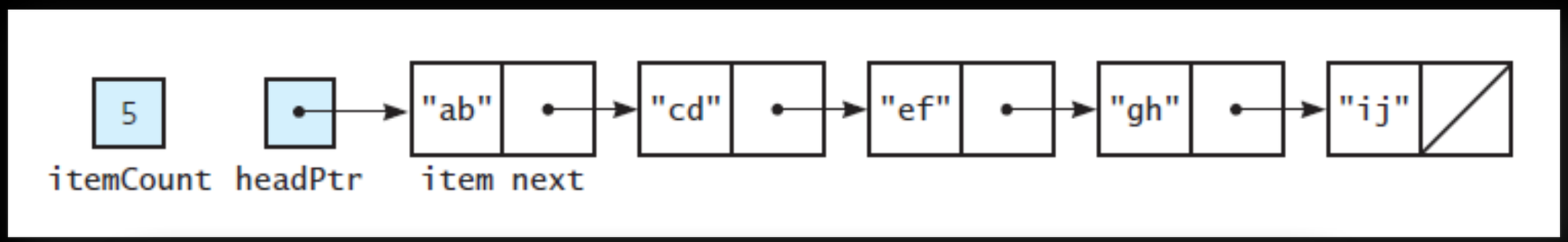


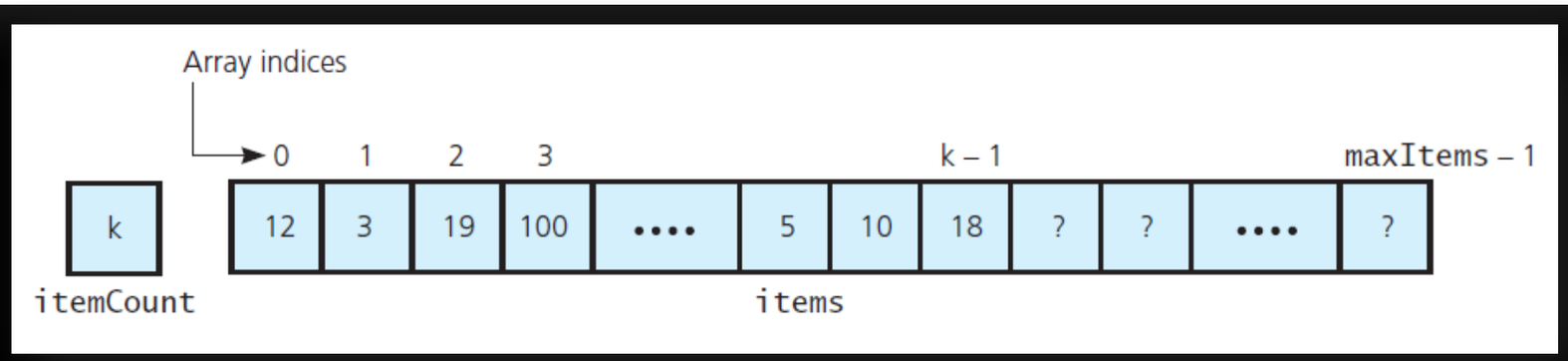
Efficiency

Create a new node and assign two pointers

What about adding to end of chain?

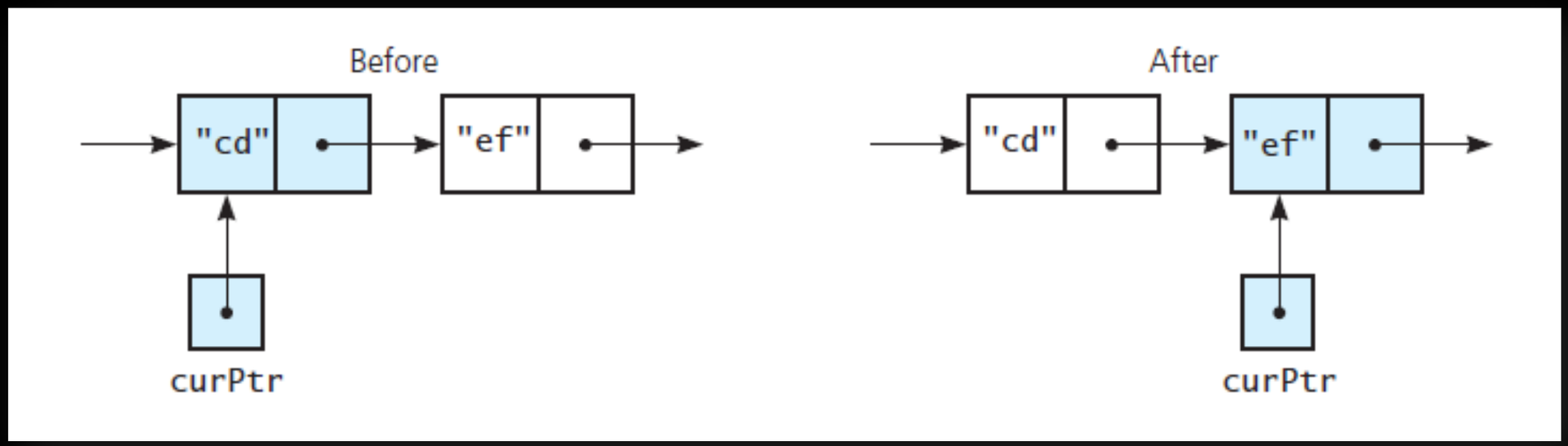
What about adding to front of array?





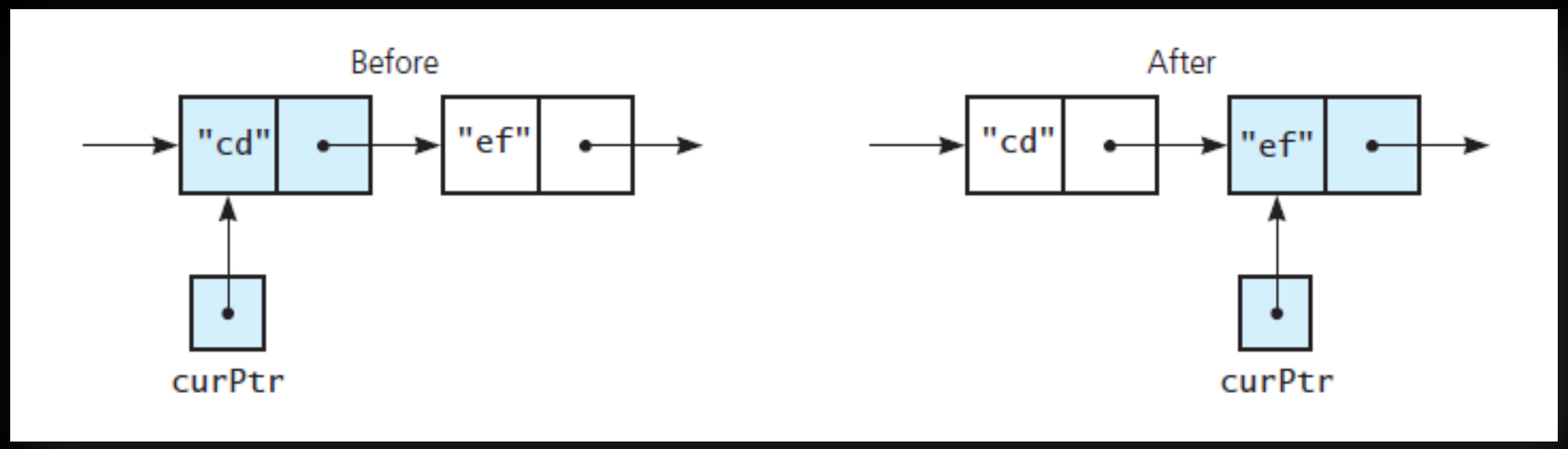
In-Class Task

Write ***Pseudocode*** to traverse the chain from first node to last



Traversing the chain

```
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
}
```

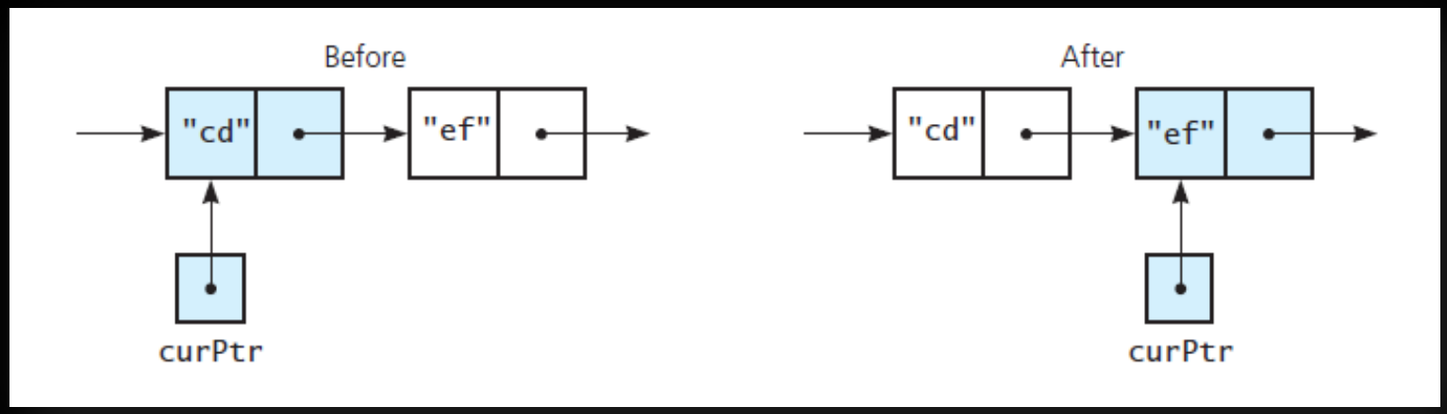


LinkedBag Implementation

The `toVector` method

```
#include "LinkedBag.h"


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
```

Traversing:
- Visit each node
- Copy it



LinkedBag Implementation

Similarly `getFrequencyOf` will:

traverse the chain and

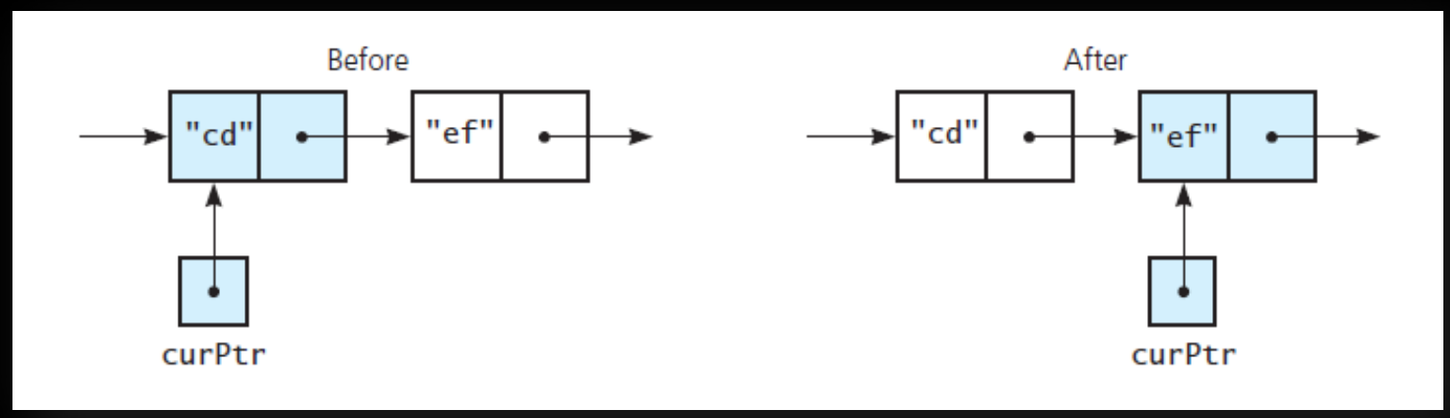
count frequency of (count each) `anEntry`

LinkedBag Implementation

The `getPointerTo` method

```
#include "LinkedBag.h"


template<class ItemType>
Node<ItemType>* LinkedBag<ItemType>::getPointerTo(const ItemType& anEntry) const
{
   bool found = false;
   Node<ItemType>* curPtr = headPtr;


   while (!found && (curPtr != nullptr))
   {
      if (anEntry == curPtr->getItem())
         found = true;
      else
         curPtr = curPtr->getNext();
   } // end while


   return curPtr;
} // end getPointerTo
```

Traversing:
- Visit each node
- if found what looking for
- return



Efficiency

No fixed number of steps

Depends on location of `anEntry`

- 1 "*check*" if it is found at first node (best case)
- n "*checks*" if it is found at last node (worst case)
- approximately n/2 on average?

Purposely vague
Some fixed amount of work

LinkedBag Implementation

```
#include "LinkedBag.h"


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());
      // Delete 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
```

The `remove` method

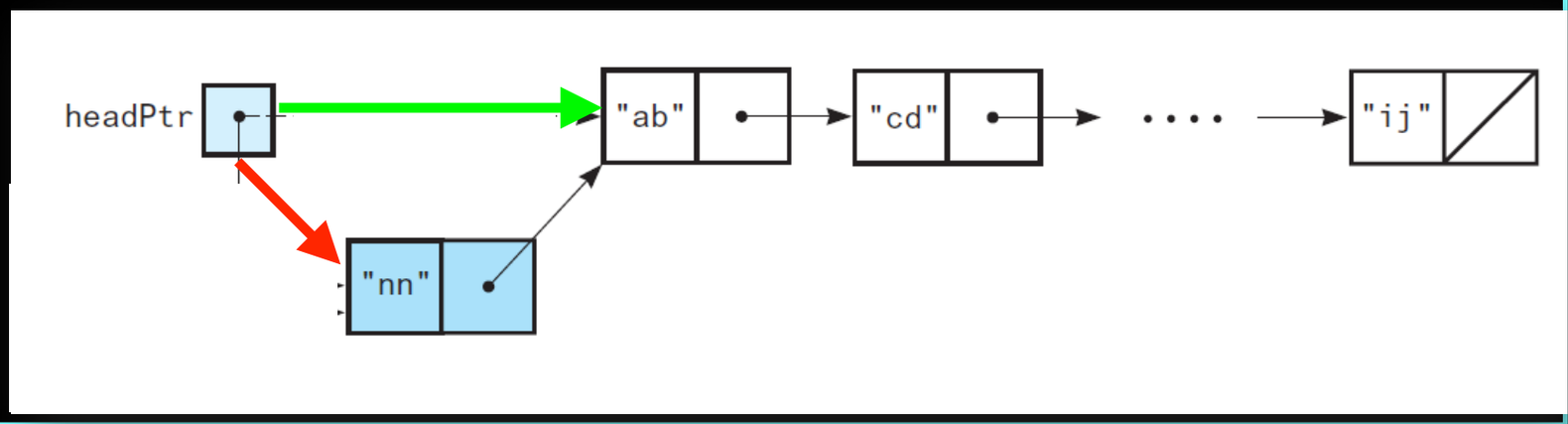
Find `anEntry`

Deleting first node is easy

Copy data from first node to node to delete

Delete first node

Must do this!!! Avoid memory leaks!!!



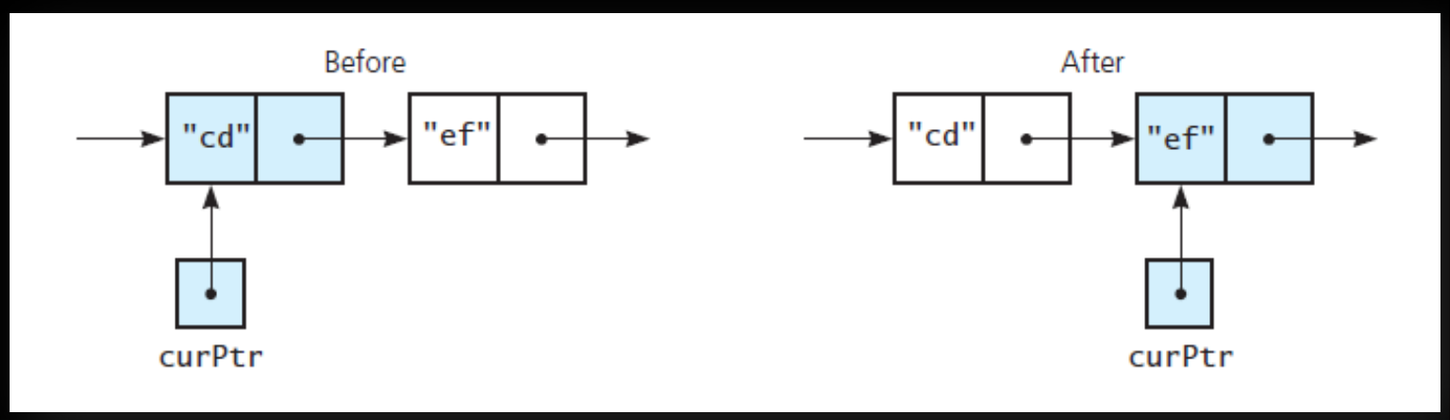
LinkedBag Implementation

```
#include "LinkedBag.h"


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
```

The `clear` method

Once again we are traversing:
- Visit each node
- Delete it

Must do this!!! Avoid memory Leak!!!



Dynamic Memory Considerations

Each new node added to the chain is allocated dynamically and stored on the heap

Programmer must ensure this memory is deallocated when object is destroyed!

Avoid memory leaks!!!!

LinkedBag Implementation

```
#include "LinkedBag.h"


template<class ItemType>
LinkedBag<ItemType>::~LinkedBag()
{

    clear();

}   // end destructor
```

The `destructor`

Ensure heap space is returned to the system

Must do this!!! Avoid memory leaks!!!

Copy Constructor

1. Initialize one object from another of the same type

```
MyClass one;
MyClass two = one;
```

More explicitly

```
MyClass one;
MyClass two(one); // Identical to above.
```

Creates a **new object** as a copy of another one

Compiler will provide one but may not appropriate for complex objects

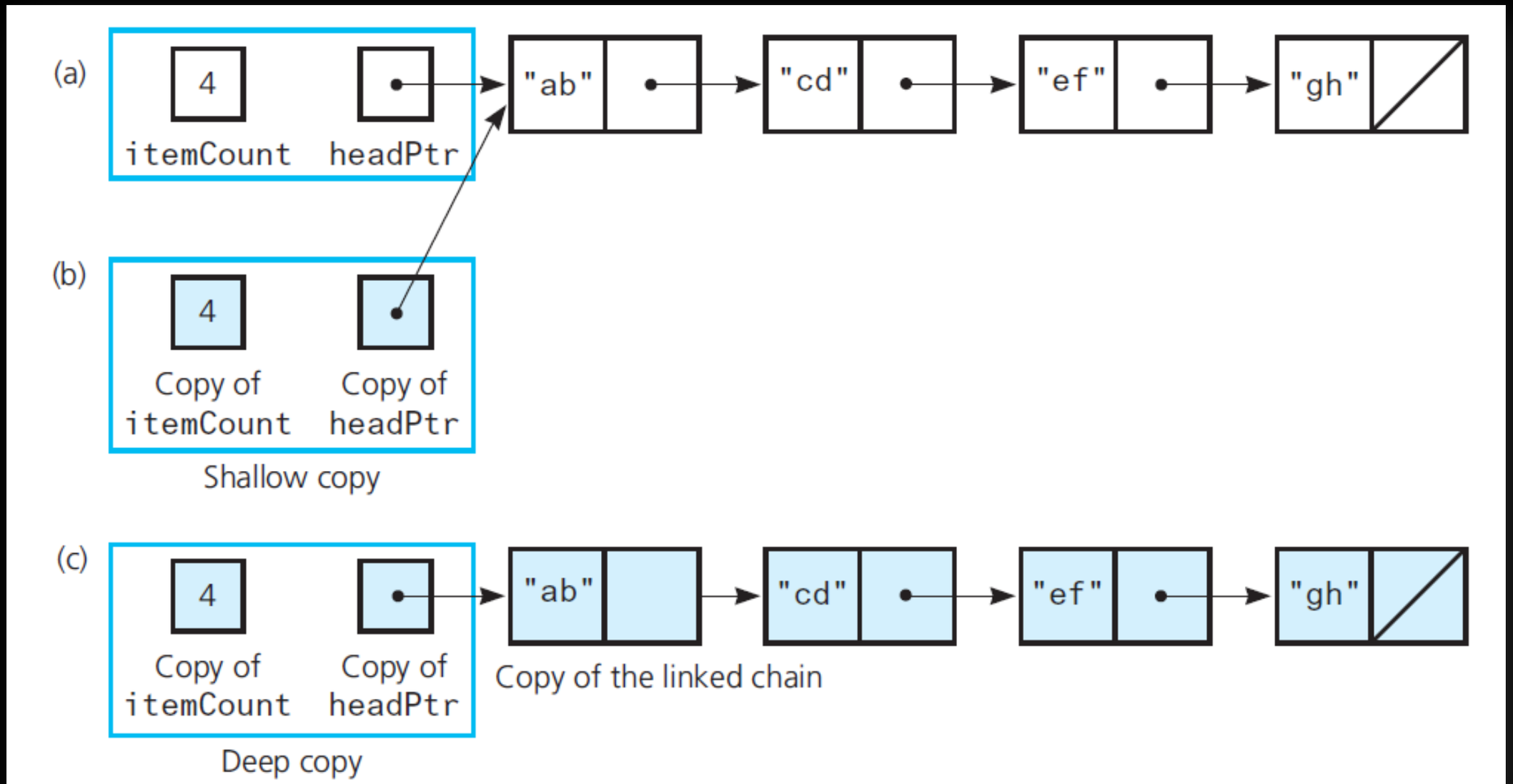
2. Copy an object to **pass by value** as an argument to a function

```
void MyFunction(MyClass arg) {
        /* ... */
}
```

3. Copy an object to be **returned** by a function

```
MyClass MyFunction() {
   MyClass mc;
   return mc;
}
```

Deep vs Shallow Copy



LinkedBag Implementation

```
#include "LinkedBag.h"
template<class ItemType>
LinkedBag<ItemType>::LinkedBag(const LinkedBag<ItemType>& aBag)
{
   itemCount = aBag.itemCount;
   Node<ItemType>* origChainPtr = aBag.headPtr;  // Points to nodes in original chain
   if (origChainPtr == nullptr)
      headPtr = nullptr;  // Original bag is empty
   else
   {
      // Copy first node
      headPtr = new Node<ItemType>();
      headPtr->setItem(origChainPtr->getItem());

      // Copy remaining nodes
      Node<ItemType>* newChainPtr = headPtr;       // Points to last node in new chain
      origChainPtr = origChainPtr->getNext();      // Advance original-chain 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 pointer to new last node
         newChainPtr = newChainPtr->getNext();
         // Advance original-chain pointer
         origChainPtr = origChainPtr->getNext();
      }  // end while
      newChainPtr->setNext(nullptr);  // Flag end of chain
   }  // end if
}  // end copy constructor
```

The `copy constructor`
A constructor whose parameter is an object of the same class

Called when object is initialized with a copy of another object, e.g.
`LinkedBag<string> my_bag = your_bag;`

Copy first node

Two traversing pointers
One to new chain, one to original chain

`while`

Copy item from current node

Create new node with item

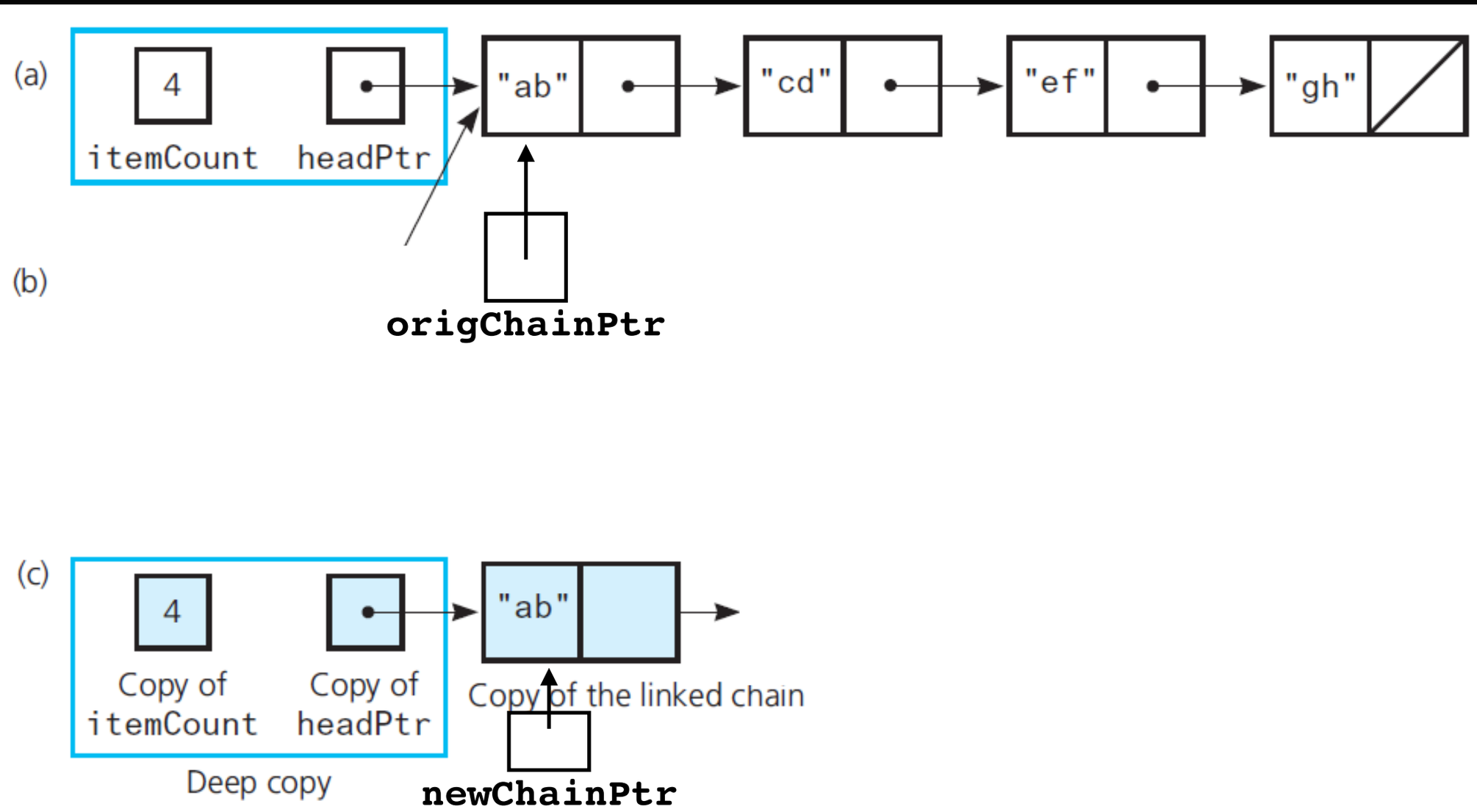
Connect new node to new chain

Advance pointer traversing new chain

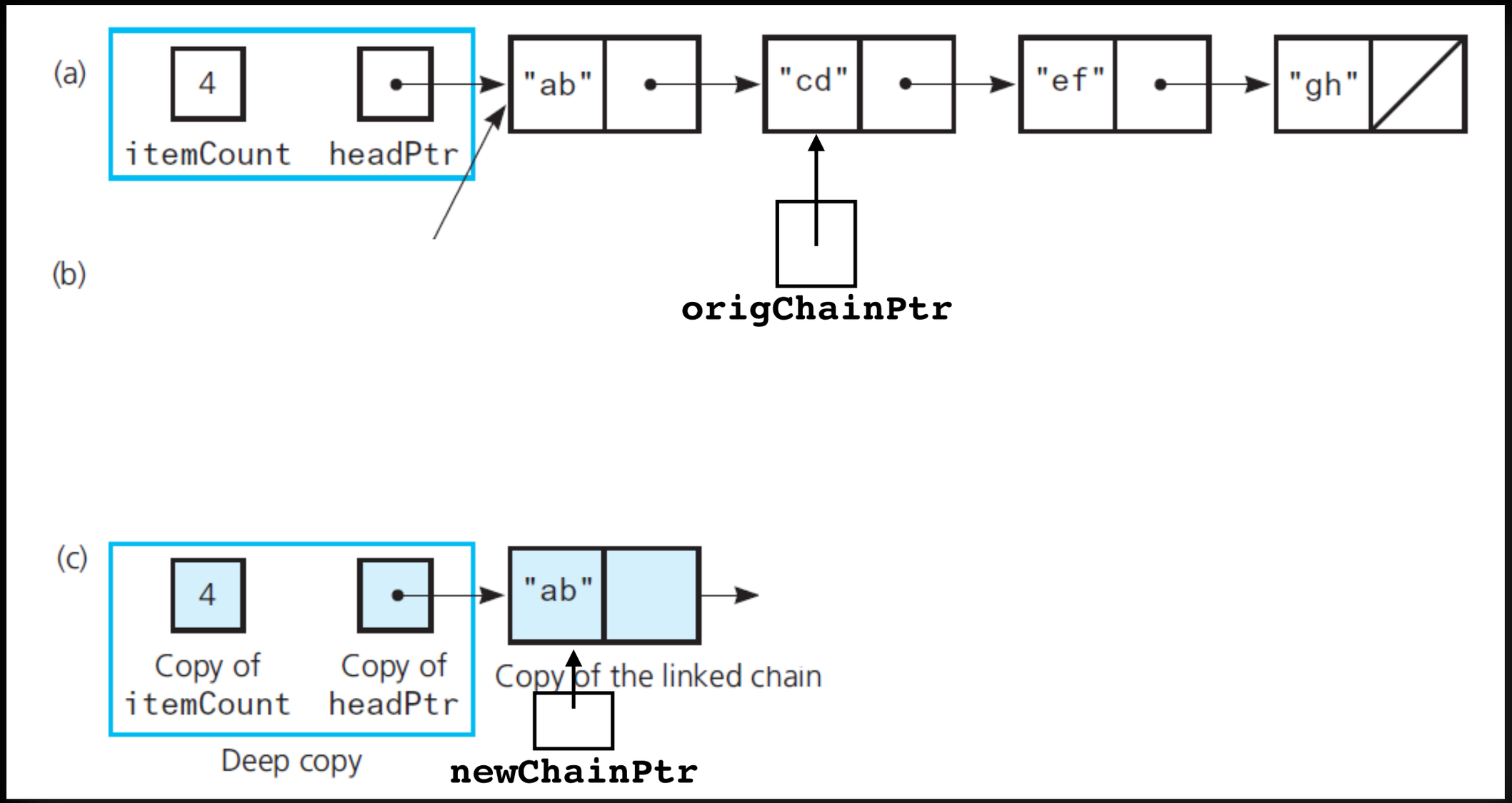
Advance pointer traversing original chain

Signal last node

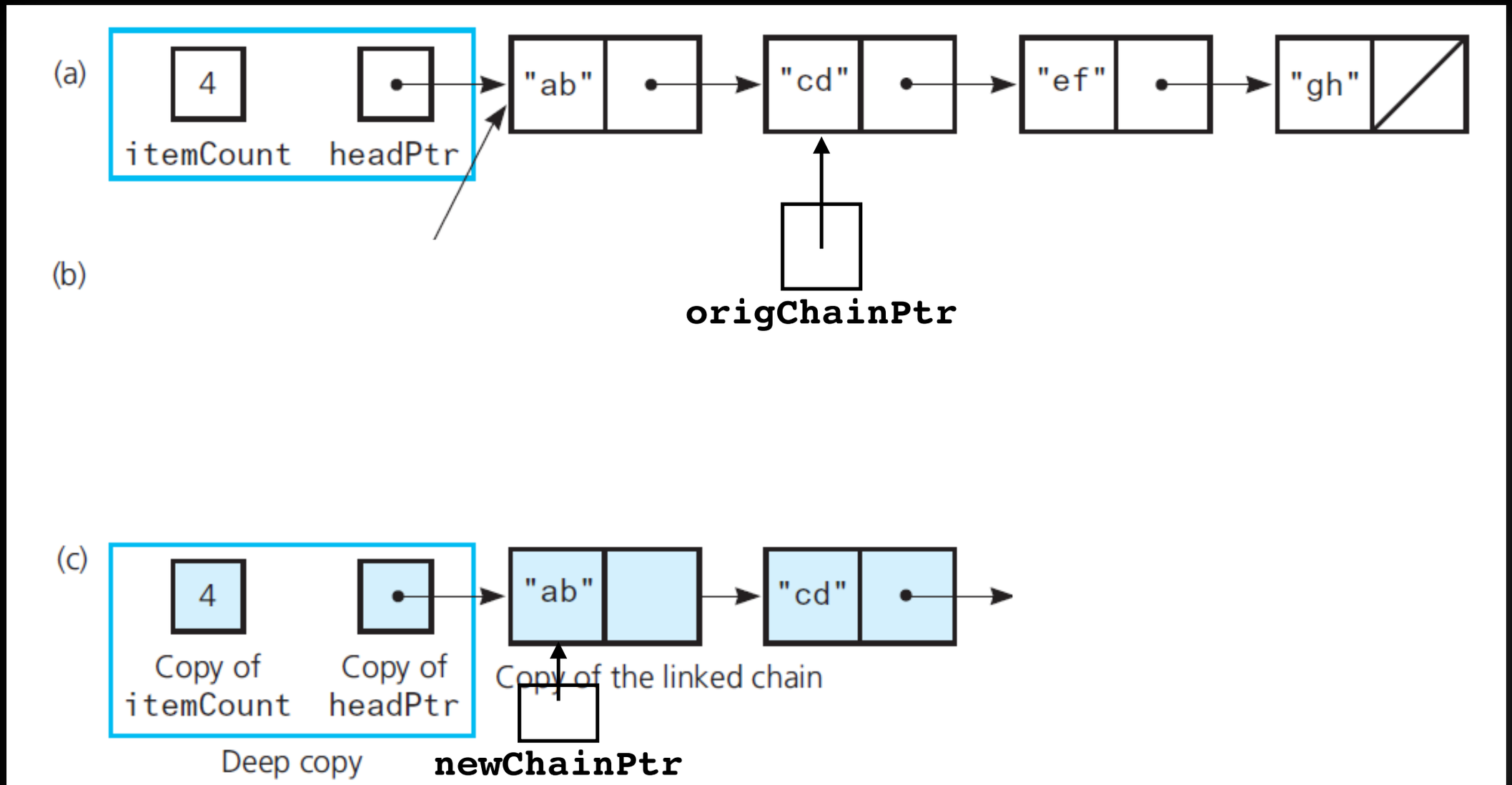
Deep vs Shallow Copy



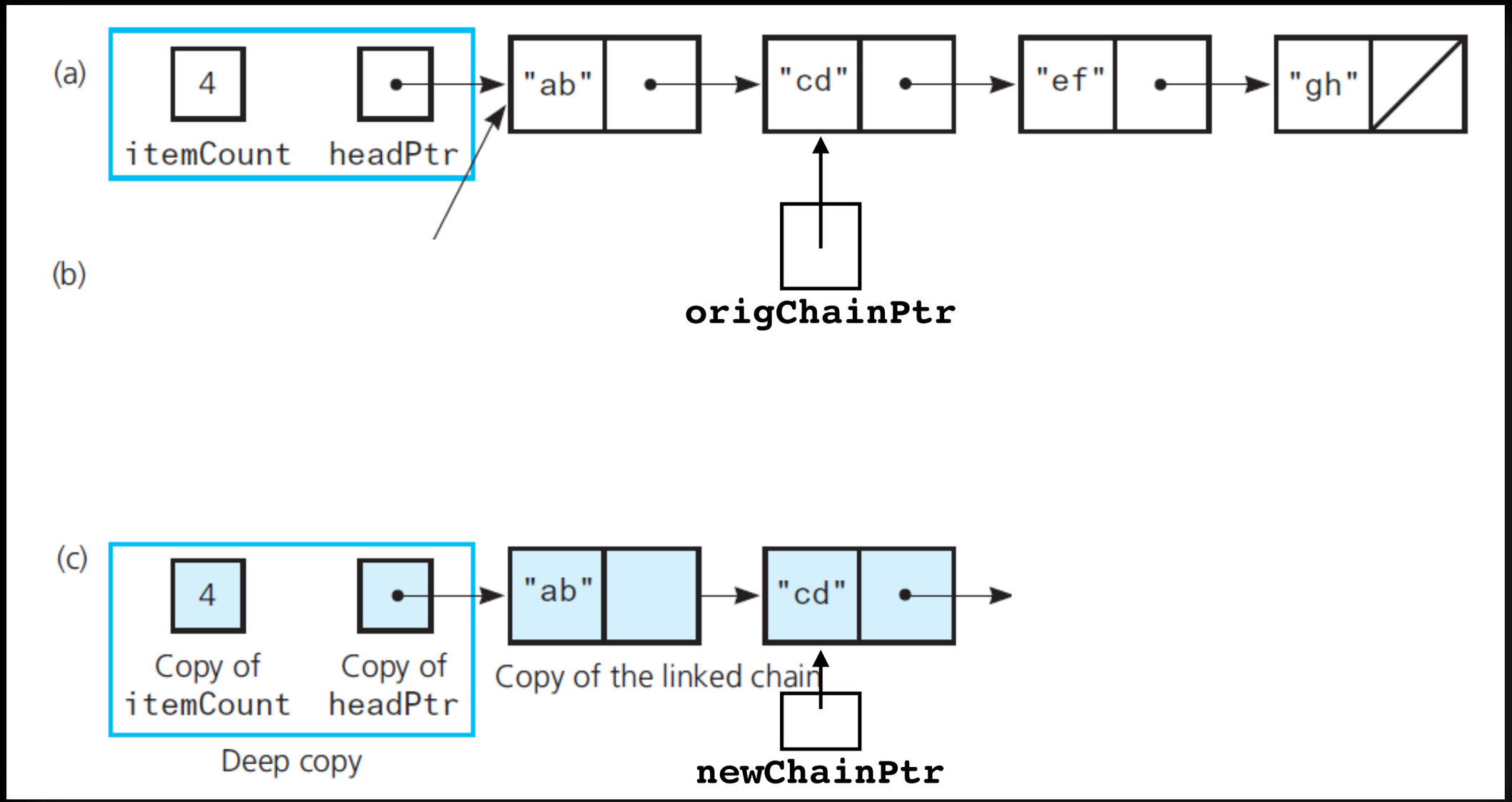
Deep vs Shallow Copy



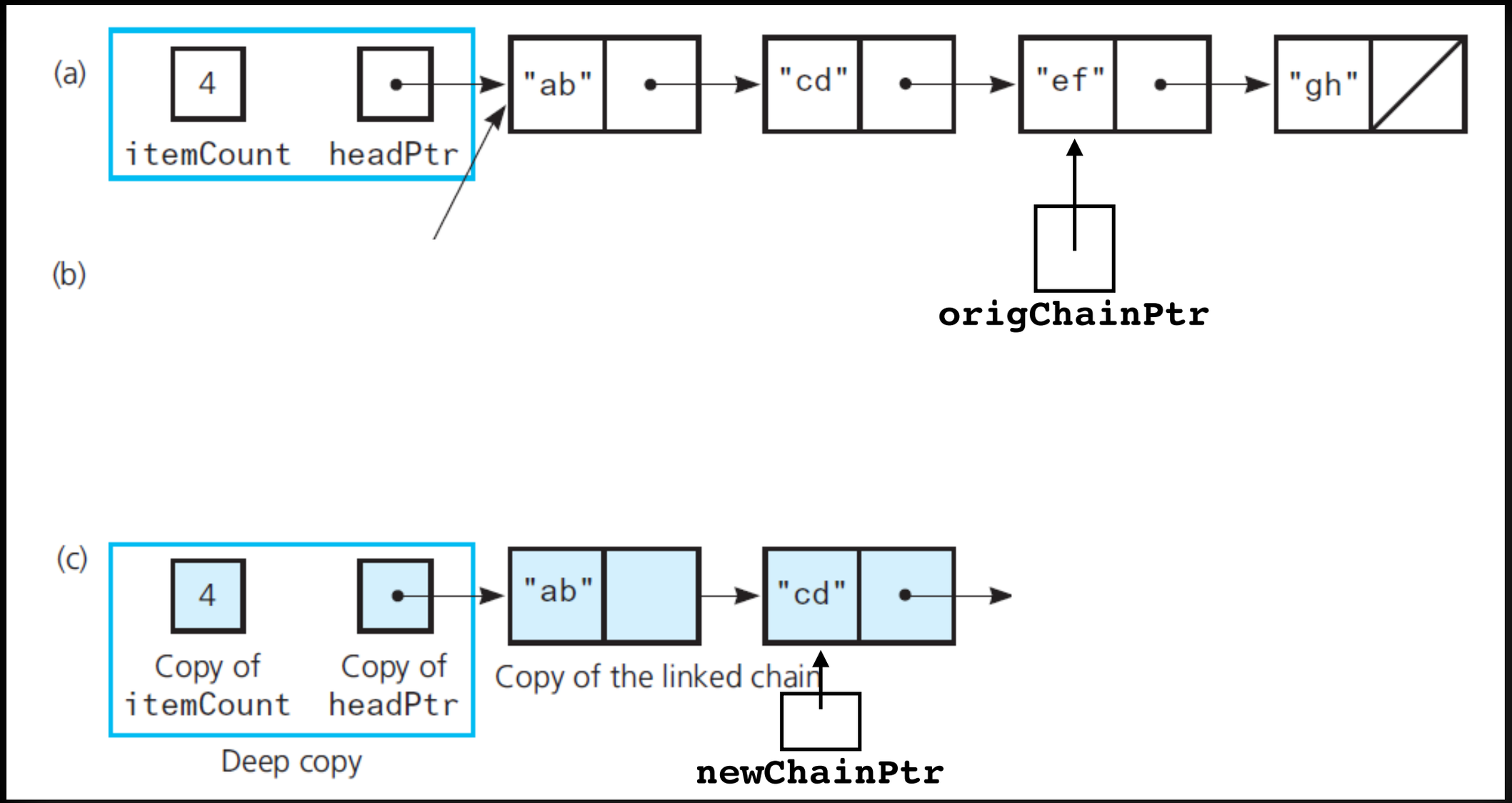
Deep vs Shallow Copy



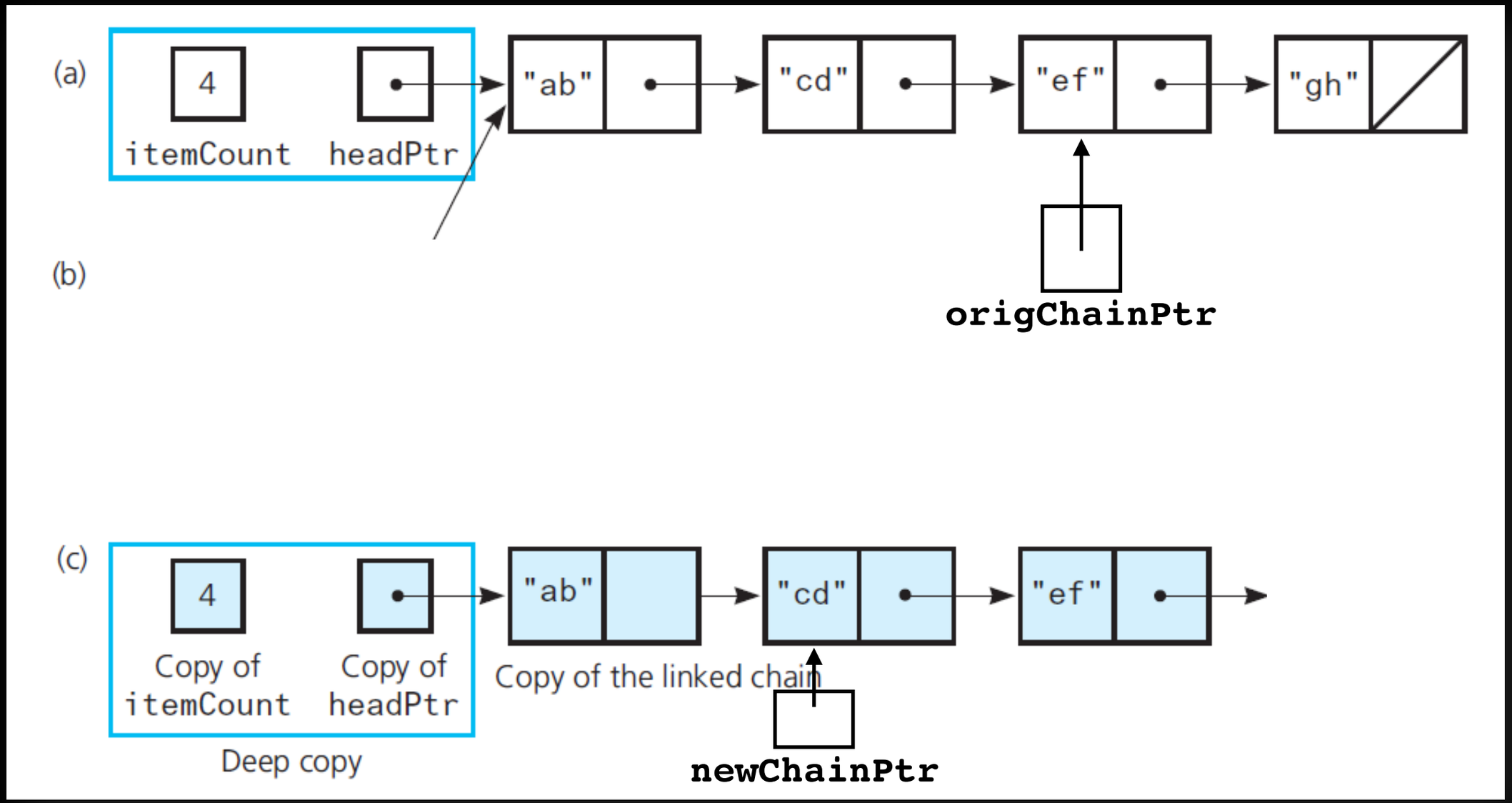
Deep vs Shallow Copy



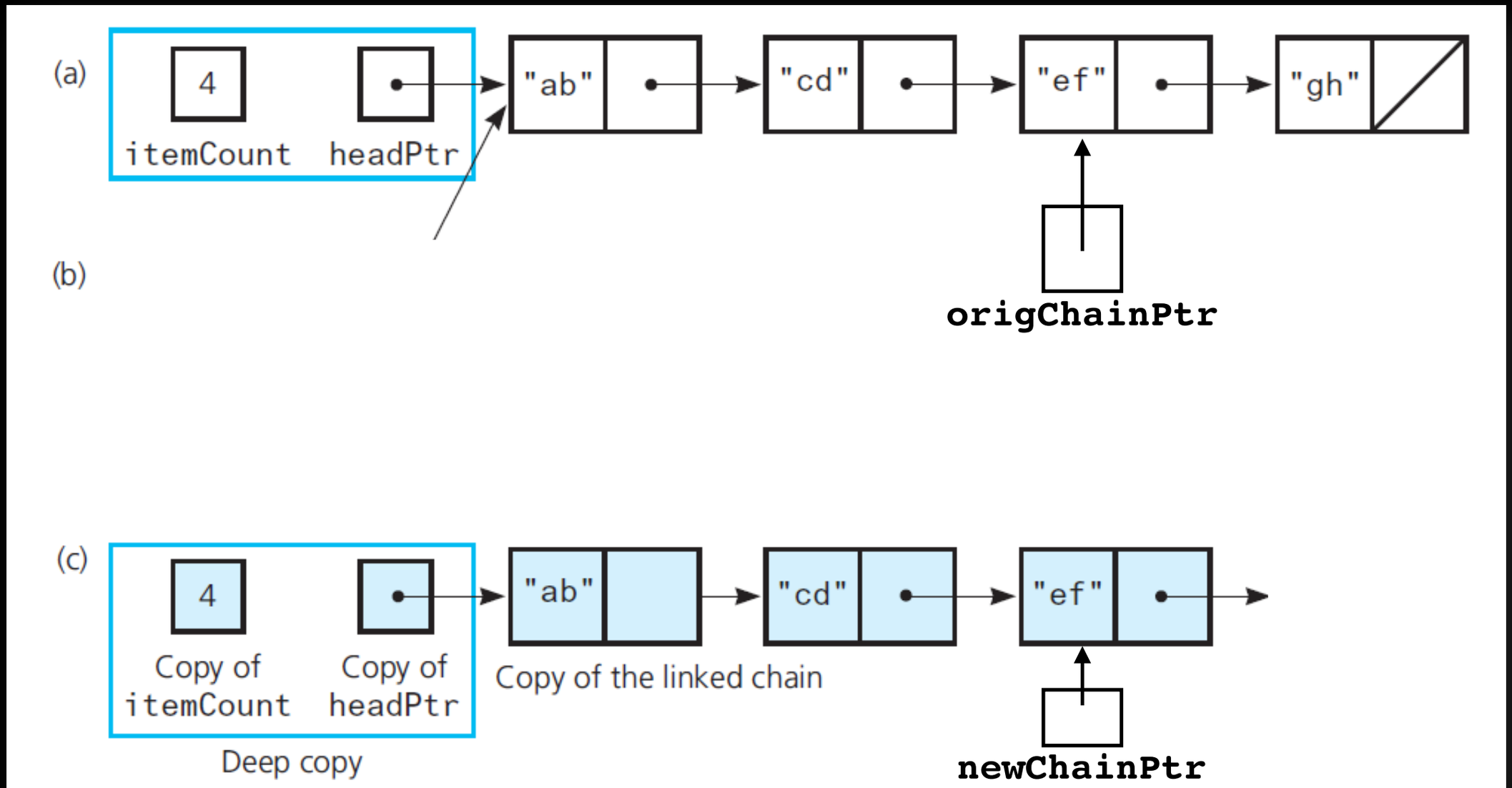
Deep vs Shallow Copy



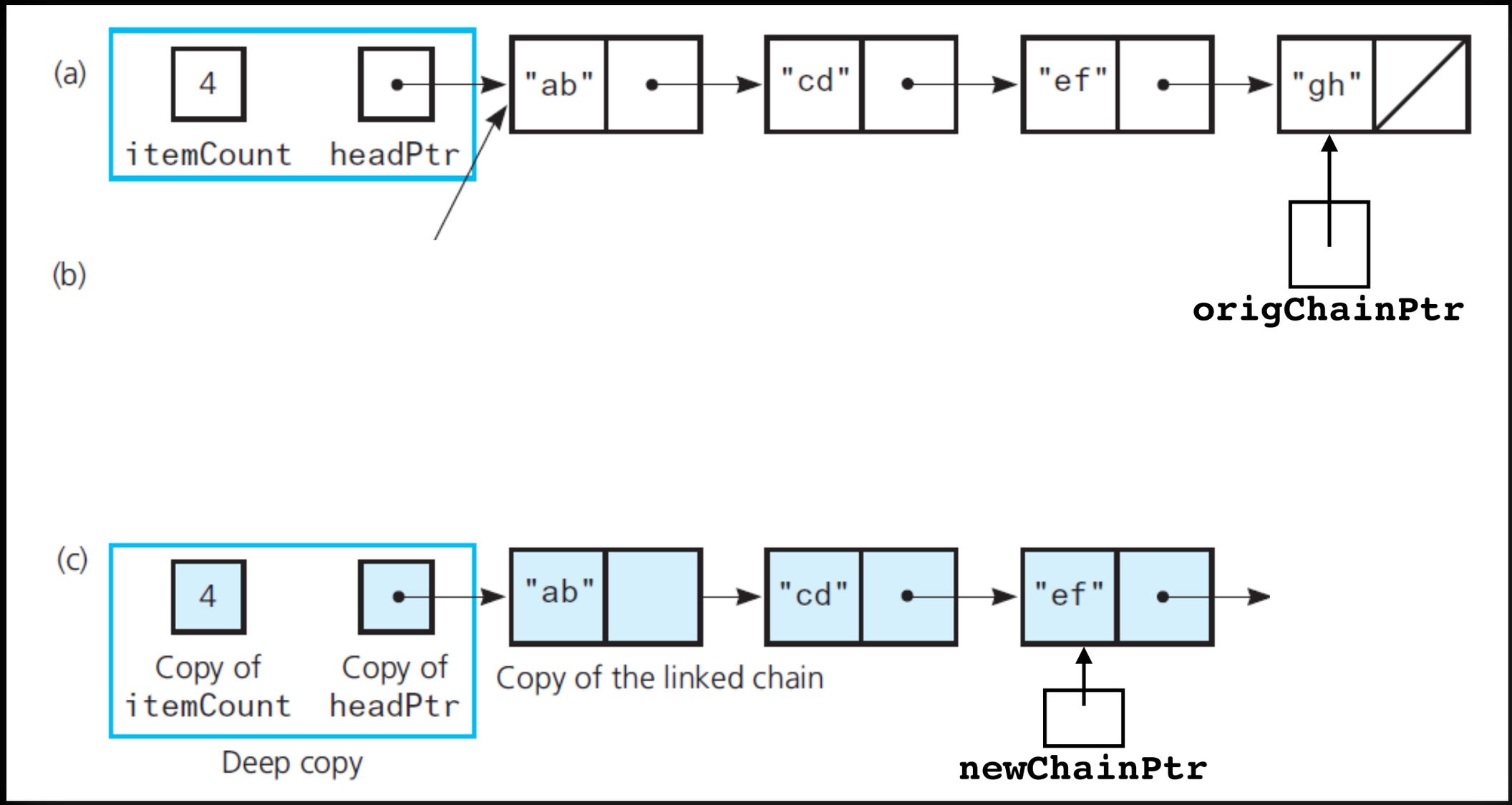
Deep vs Shallow Copy



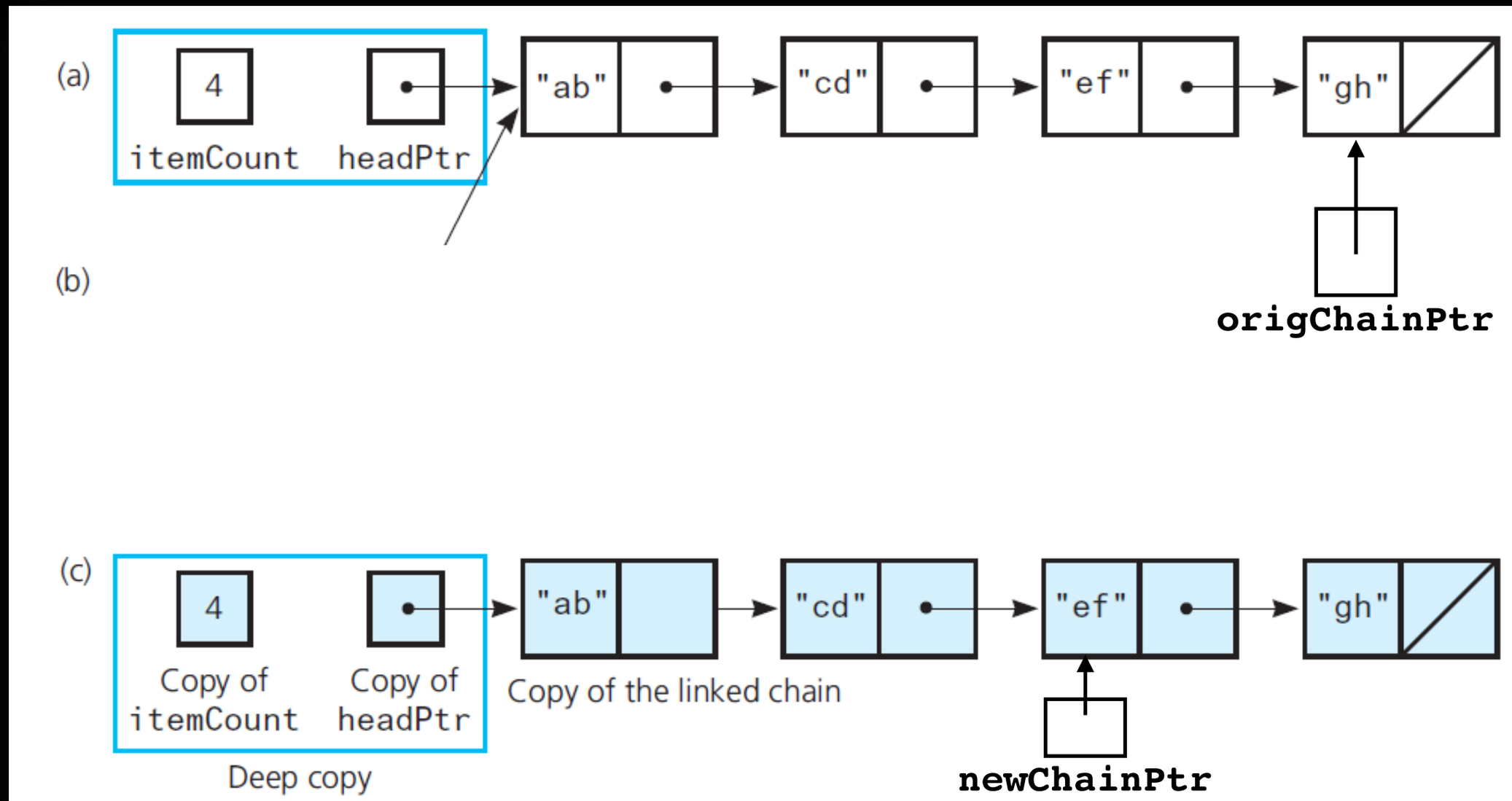
Deep vs Shallow Copy



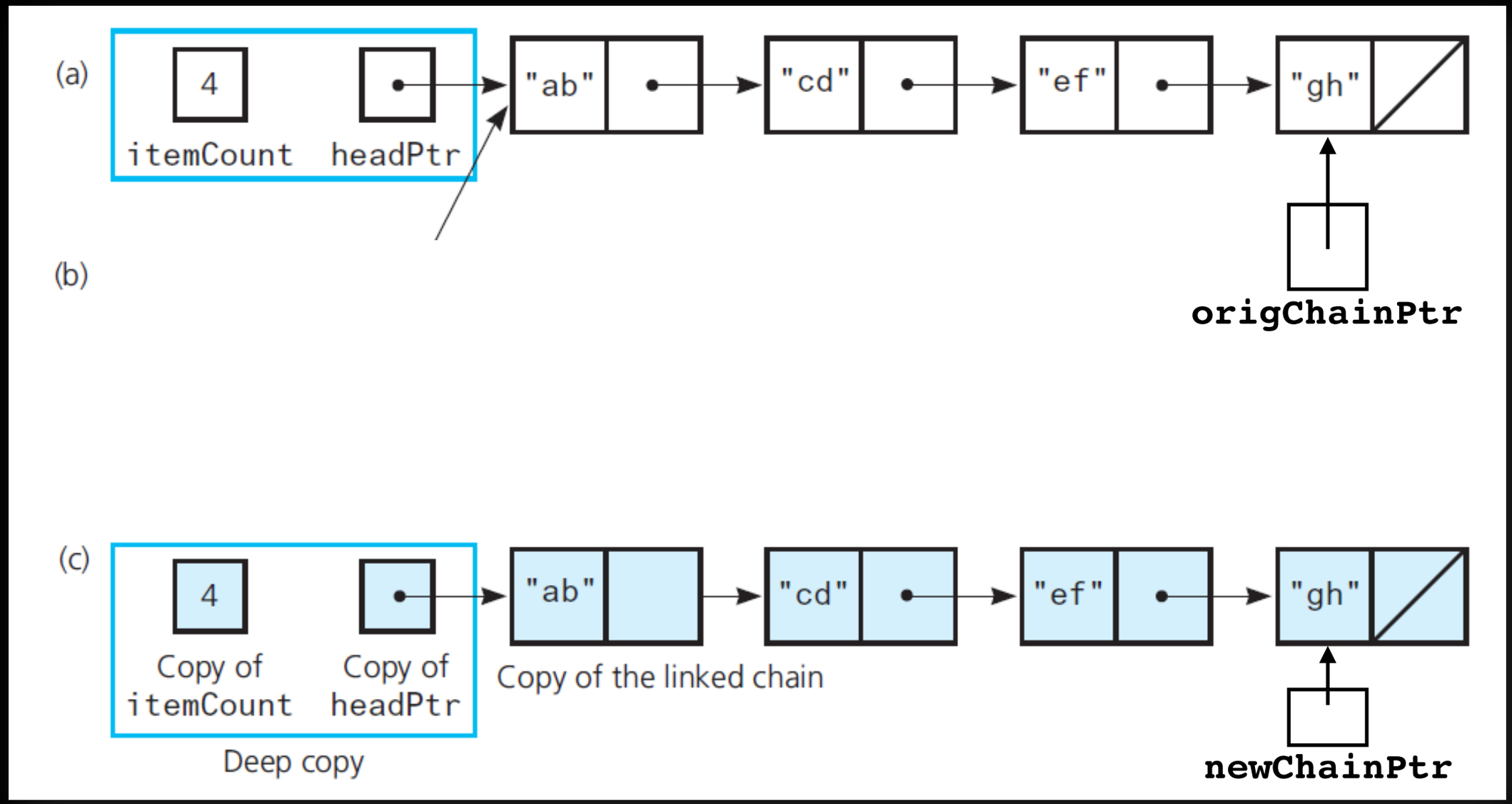
Deep vs Shallow Copy



Deep vs Shallow Copy



Deep vs Shallow Copy



Efficiency

Every time you pass or return an object by value:

- Call copy constructor
- Call destructor

For linked chain:

- Traverse entire chain to copy (`n "steps"`)
- Traverse entire chain to destroy (`n "steps"`)