

Copy Constructor

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

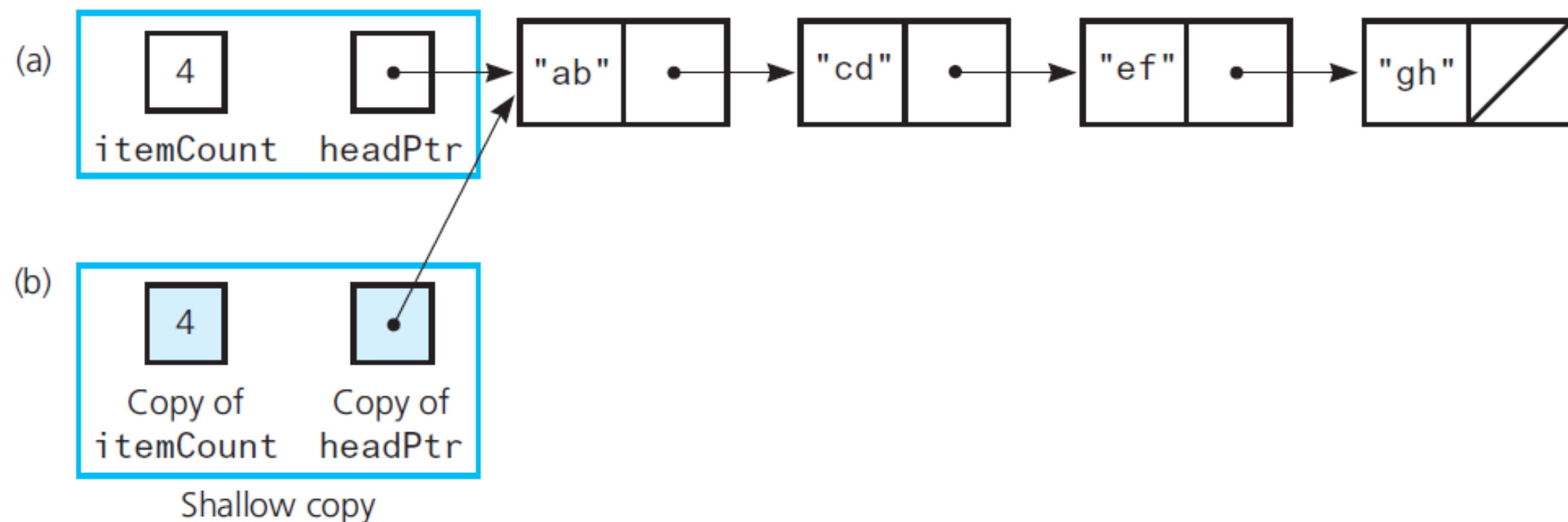
2. Copy an object to **pass by value** as an argument to a function

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

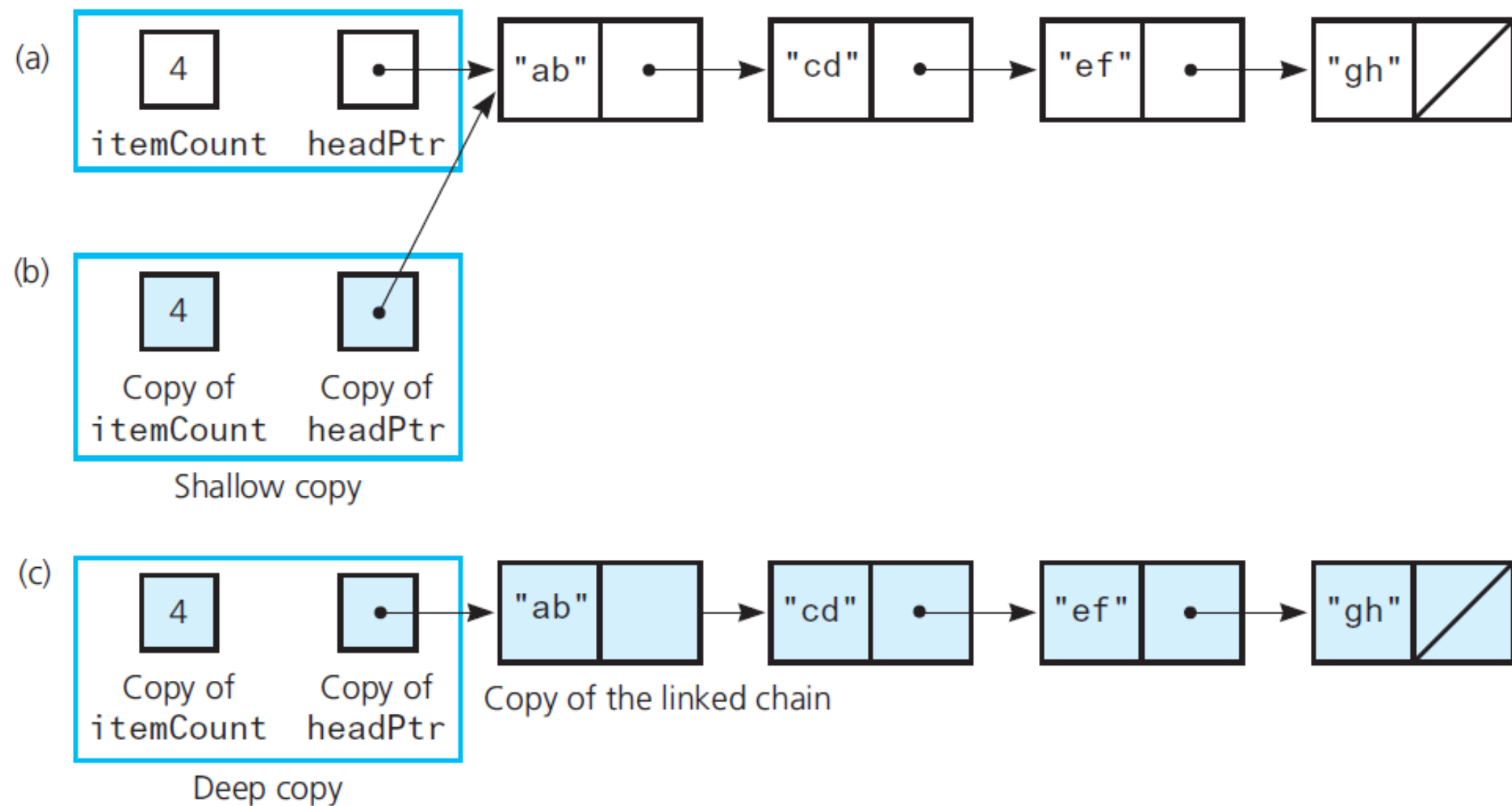
3. Copy an object to be **returned** by a function

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

Deep vs Shallow Copy



Deep vs Shallow Copy



Overloaded operator=

```
MyClass one;  
//Stuff here  
MyClass two = one;
```

Instantiation: copy constructor is called

IS DIFFERENT FROM

```
MyClass one, two;  
//Stuff here  
two = one;
```

Assignment, NOT instantiation: no constructor is called, must overload operator= to avoid shallow copy

Copy Constructor Implementation

A constructor whose parameter is an object of the same class

```
#include "LinkBag.hpp"
template<typename ItemType>
LinkBag<ItemType>::LinkBag(const LinkBag<ItemType>& a_bag)
{
    item_count_ = a_bag.item_count_;
    Node<ItemType>* orig_chain_ptr = a_bag.head_ptr_; // Points to nodes in original chain
    if (orig_chain_ptr == nullptr)
        head_ptr_ = nullptr; // Original bag is empty
    else
    {
        // Copy first node
        head_ptr_ = new Node<ItemType>;
        head_ptr_>setItem(orig_chain_ptr->getItem());
        // Copy remaining nodes
        Node<ItemType>* new_chain_ptr = head_ptr_; // Points to last node in new chain
        orig_chain_ptr = orig_chain_ptr->getNext(); // Advance original-chain pointer
        while (orig_chain_ptr != nullptr)
        {
            // Get next item from original chain
            ItemType next_item = orig_chain_ptr->getItem();
            // Create a new node containing the next item
            Node<ItemType>* new_node_ptr = new Node<ItemType>(next_item);
            // Link new node to end of new chain
            new_chain_ptr->setNext(new_node_ptr);
            // Advance pointer to new last node
            new_chain_ptr = new_chain_ptr->getNext();
            // Advance original-chain pointer
            orig_chain_ptr = orig_chain_ptr->getNext();
        } // end while
        new_chain_ptr->setNext(nullptr); // Flag end of chain
    } // end if
} // end copy constructor
```

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

Copy first node

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

Copy item from current node

while

Create new node with item

Connect new node to new chain

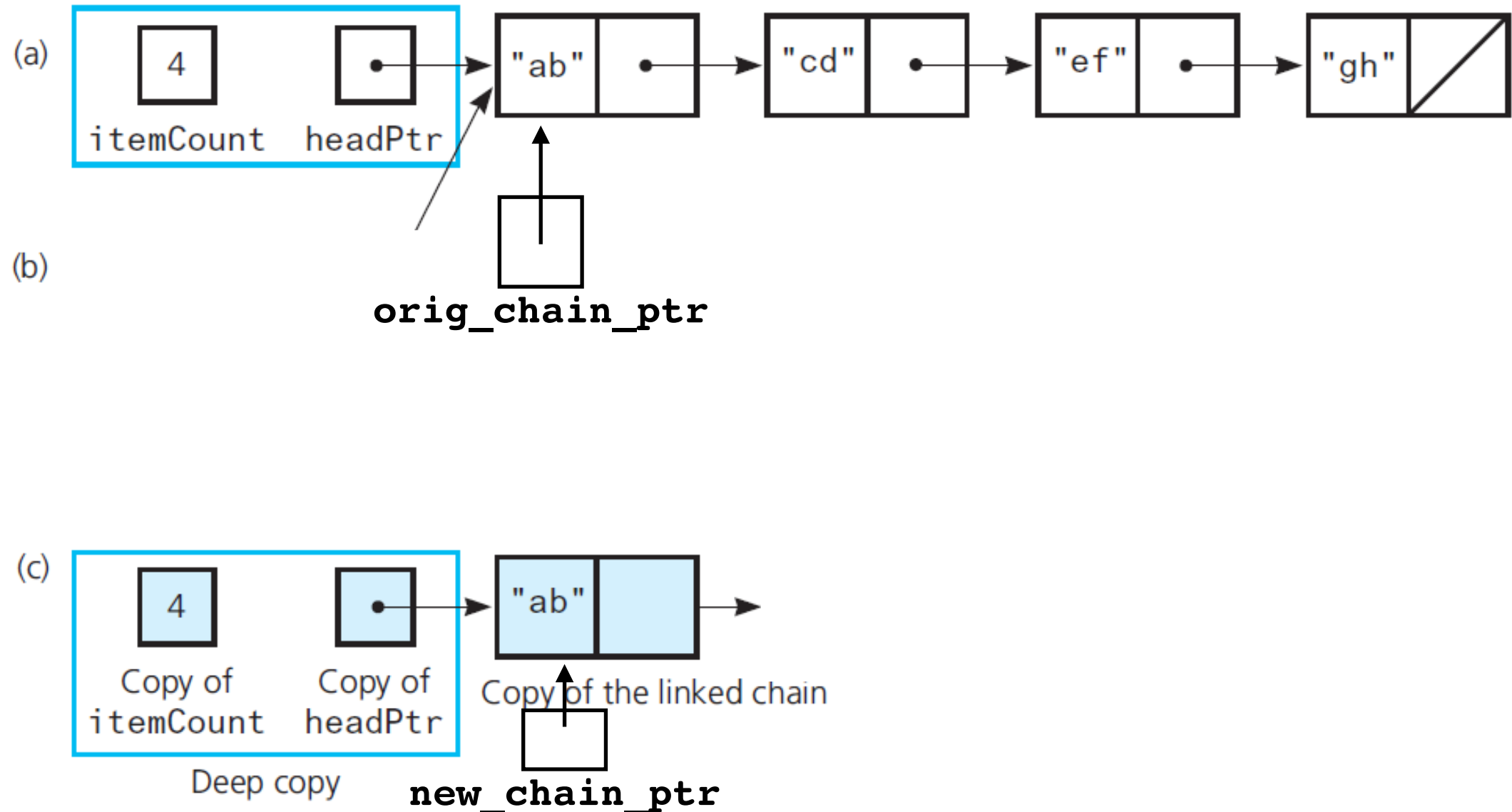
Advance pointer traversing new chain

Advance pointer traversing original chain

Signal last node

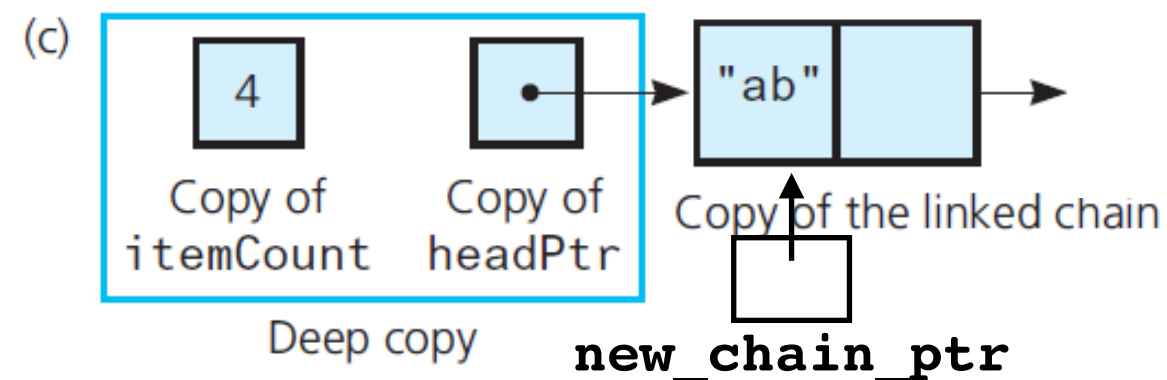
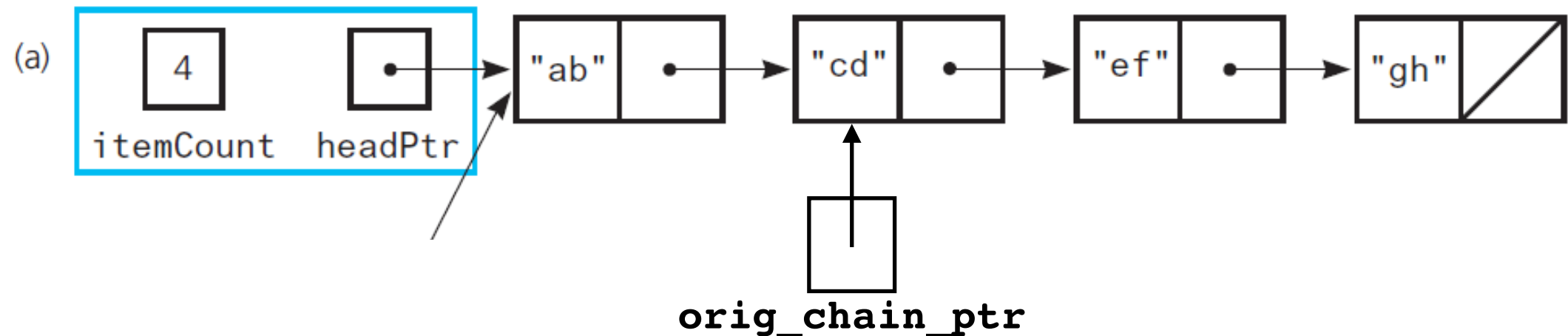
Deep vs Shallow Copy

```
// Copy first node
head_ptr_ = new Node<ItemType>();
head_ptr_->setItem(orig_chain_ptr->getItem());
// Copy remaining nodes
Node<ItemType>* new_chain_ptr = head_ptr_;
// Points to last node in new chain
orig_chain_ptr = orig_chain_ptr->getNext();
```



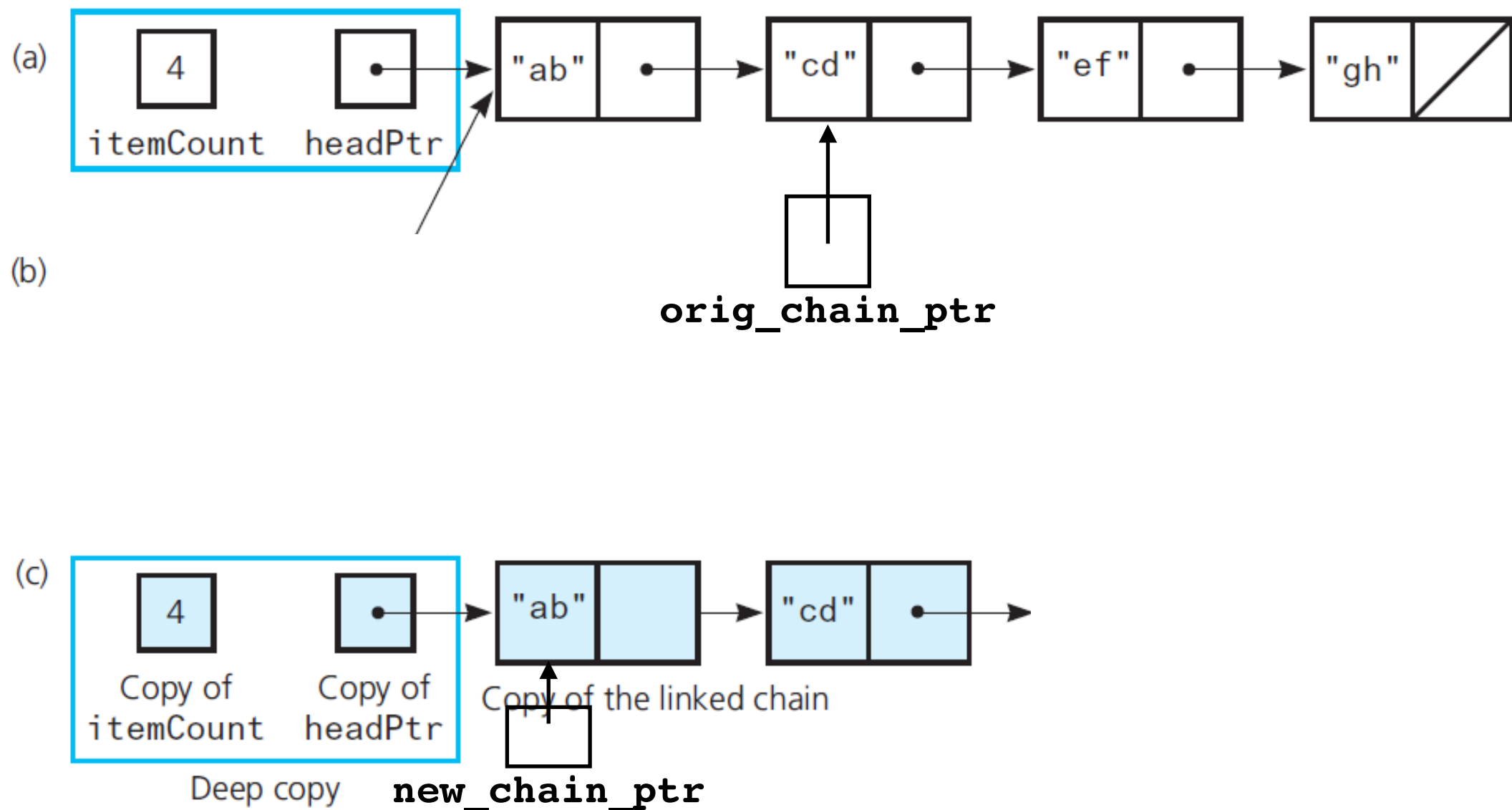
Deep vs Shallow Copy

```
// Copy first node
head_ptr_ = new Node<ItemType>();
head_ptr_->setItem(orig_chain_ptr->getItem());
// Copy remaining nodes
Node<ItemType>* new_chain_ptr = head_ptr_;
// Points to last node in new chain
orig_chain_ptr = orig_chain_ptr->getNext();
```



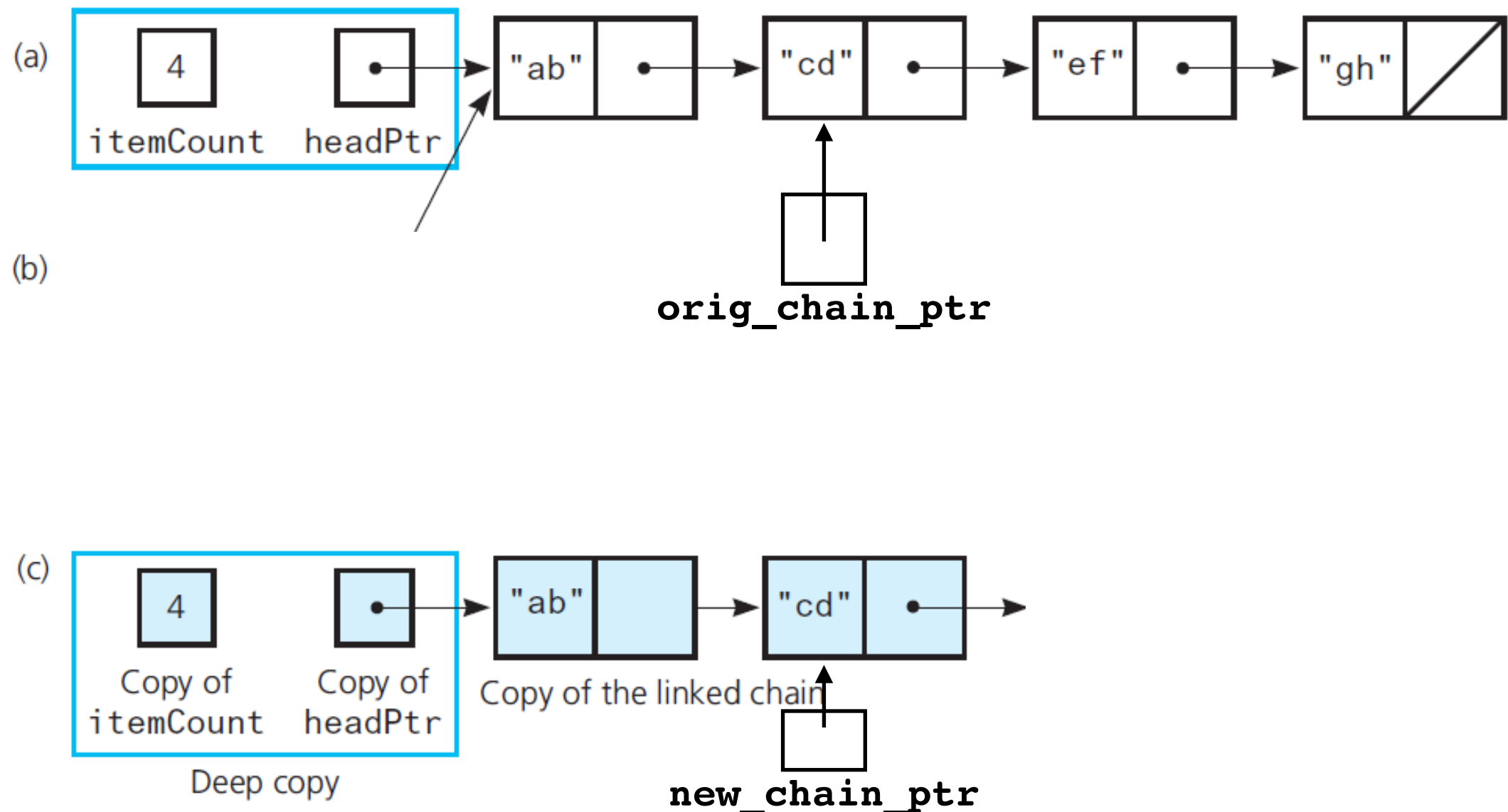
Deep vs Shallow Copy

```
while (orig_chain_ptr != nullptr) {  
    // Get next item from original chain  
    ItemType next_item = orig_chain_ptr->getItem();  
    // Create a new node containing the next item  
    Node<ItemType>* new_node_ptr = new Node<C>(next_item);  
    // Link new node to end of new chain  
    new_chain_ptr->setNext(new_node_ptr);  
    // Advance pointer to new last node  
    new_chain_ptr = new_chain_ptr->getNext();  
    // Advance original-chain pointer  
    orig_chain_ptr = orig_chain_ptr->getNext();  
}
```



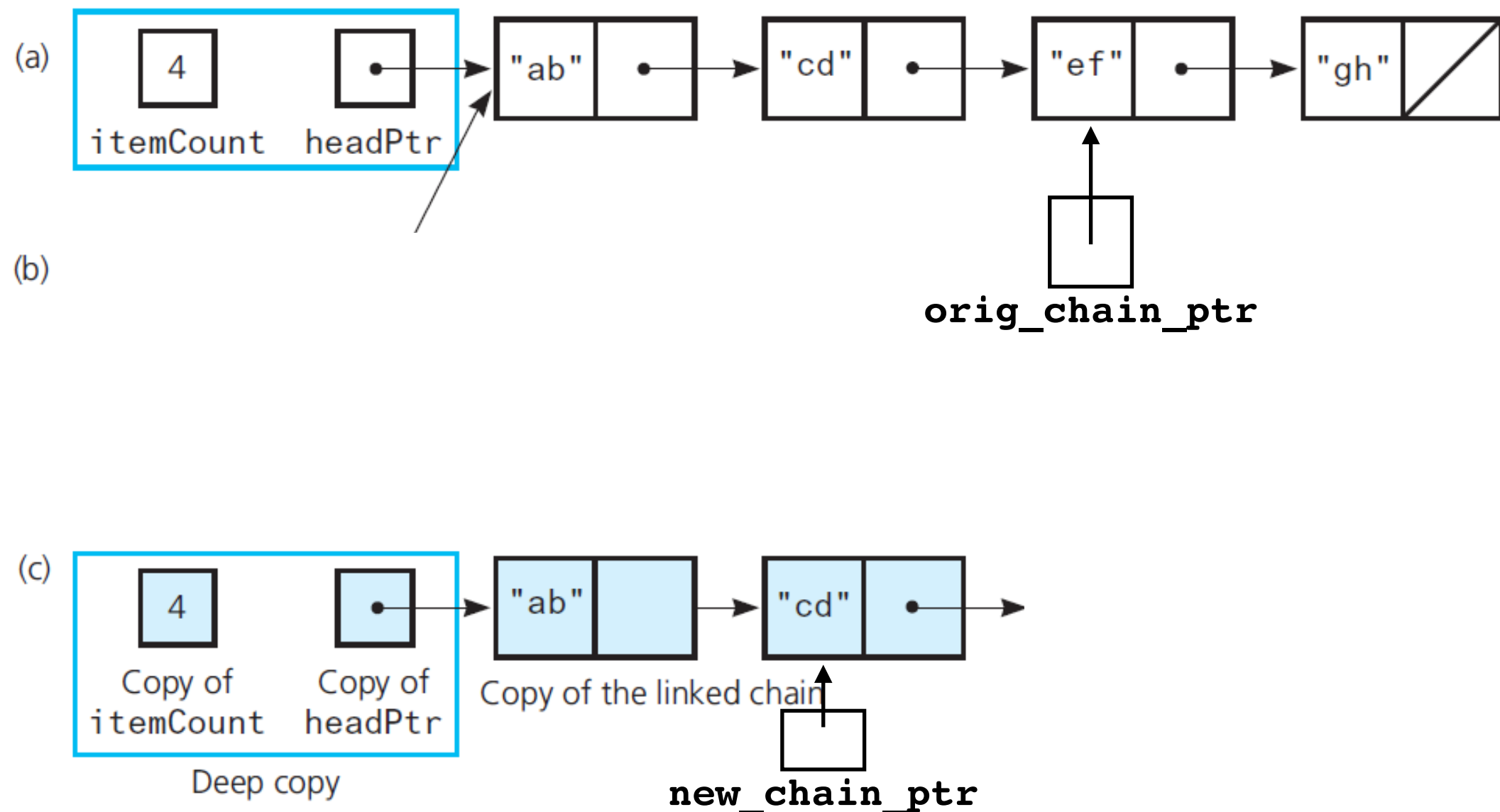
Deep vs Shallow Copy

```
while (orig_chain_ptr != nullptr) {  
    // Get next item from original chain  
    ItemType next_item = orig_chain_ptr->getItem();  
    // Create a new node containing the next item  
    Node<ItemType>* new_node_ptr = new Node<ItemType>(next_item);  
    // Link new node to end of new chain  
    new_chain_ptr->setNext(new_node_ptr);  
    // Advance pointer to new last node  
    new_chain_ptr = new_chain_ptr->getNext();  
    // Advance original-chain pointer  
    orig_chain_ptr = orig_chain_ptr->getNext();  
}
```



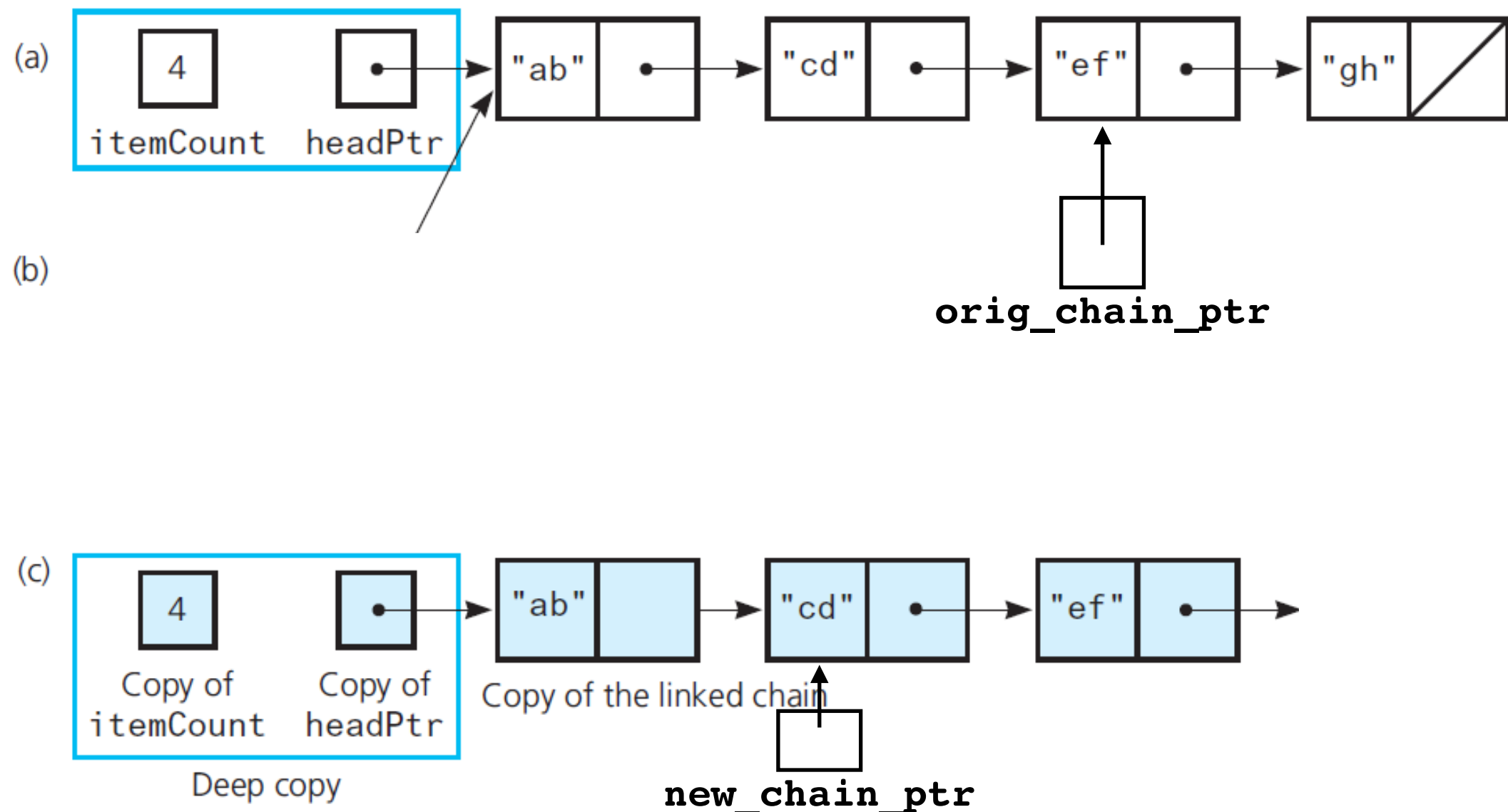
Deep vs Shallow Copy

```
while (orig_chain_ptr != nullptr) {  
    // Get next item from original chain  
    ItemType next_item = orig_chain_ptr->getItem();  
    // Create a new node containing the next item  
    Node<ItemType>* new_node_ptr = new Node<ItemType>(next_item);  
    // Link new node to end of new chain  
    new_chain_ptr->setNext(new_node_ptr);  
    // Advance pointer to new last node  
    new_chain_ptr = new_chain_ptr->getNext();  
    // Advance original-chain pointer  
    orig_chain_ptr = orig_chain_ptr->getNext();  
}
```



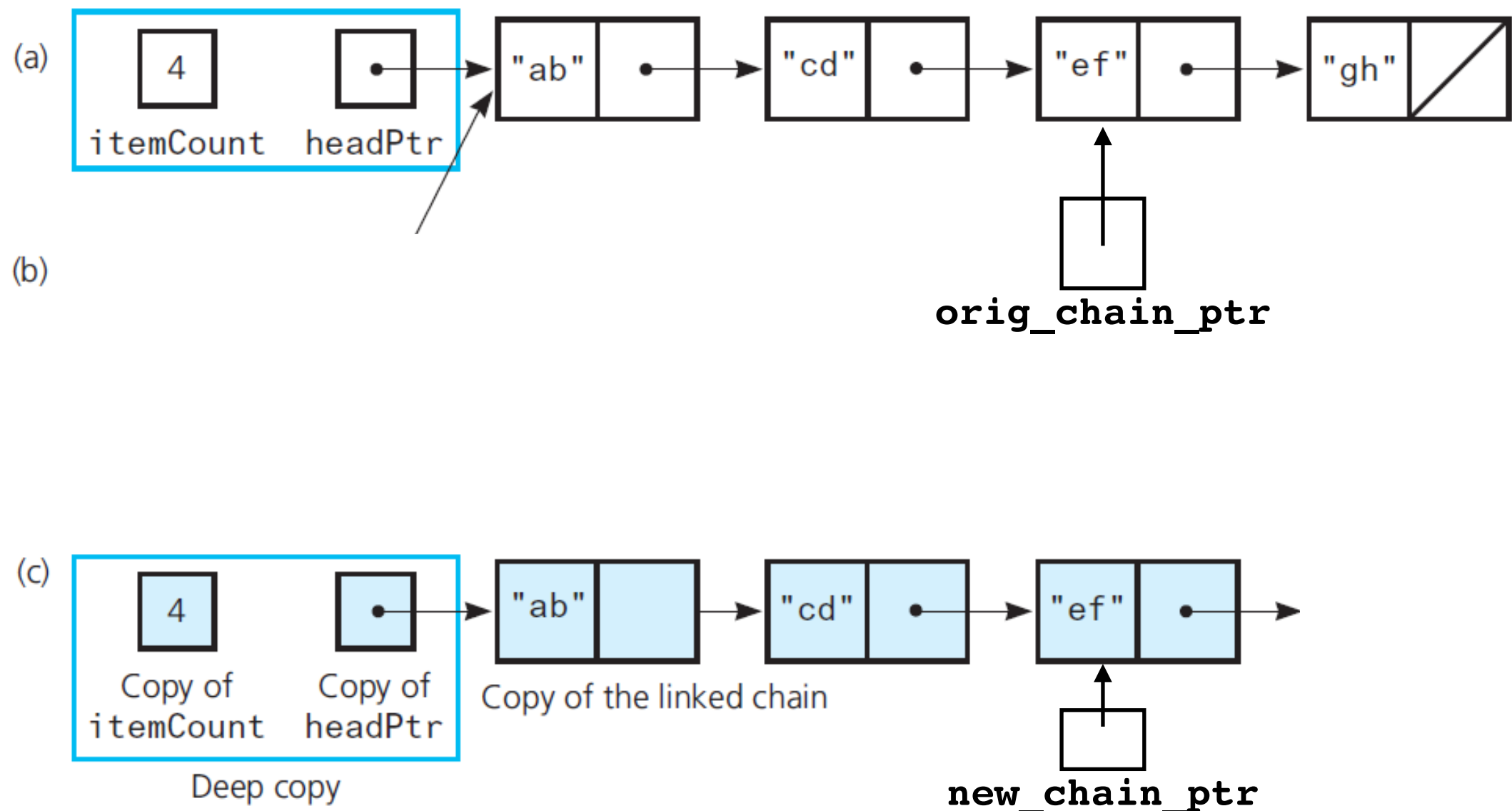
Deep vs Shallow Copy

```
while (orig_chain_ptr != nullptr) {  
    // Get next item from original chain  
    ItemType next_item = orig_chain_ptr->getItem();  
    // Create a new node containing the next item  
    Node<ItemType>* new_node_ptr = new Node<ItemType>(next_item);  
    // Link new node to end of new chain  
    new_chain_ptr->setNext(new_node_ptr);  
    // Advance pointer to new last node  
    new_chain_ptr = new_chain_ptr->getNext();  
    // Advance original-chain pointer  
    orig_chain_ptr = orig_chain_ptr->getNext();  
}
```



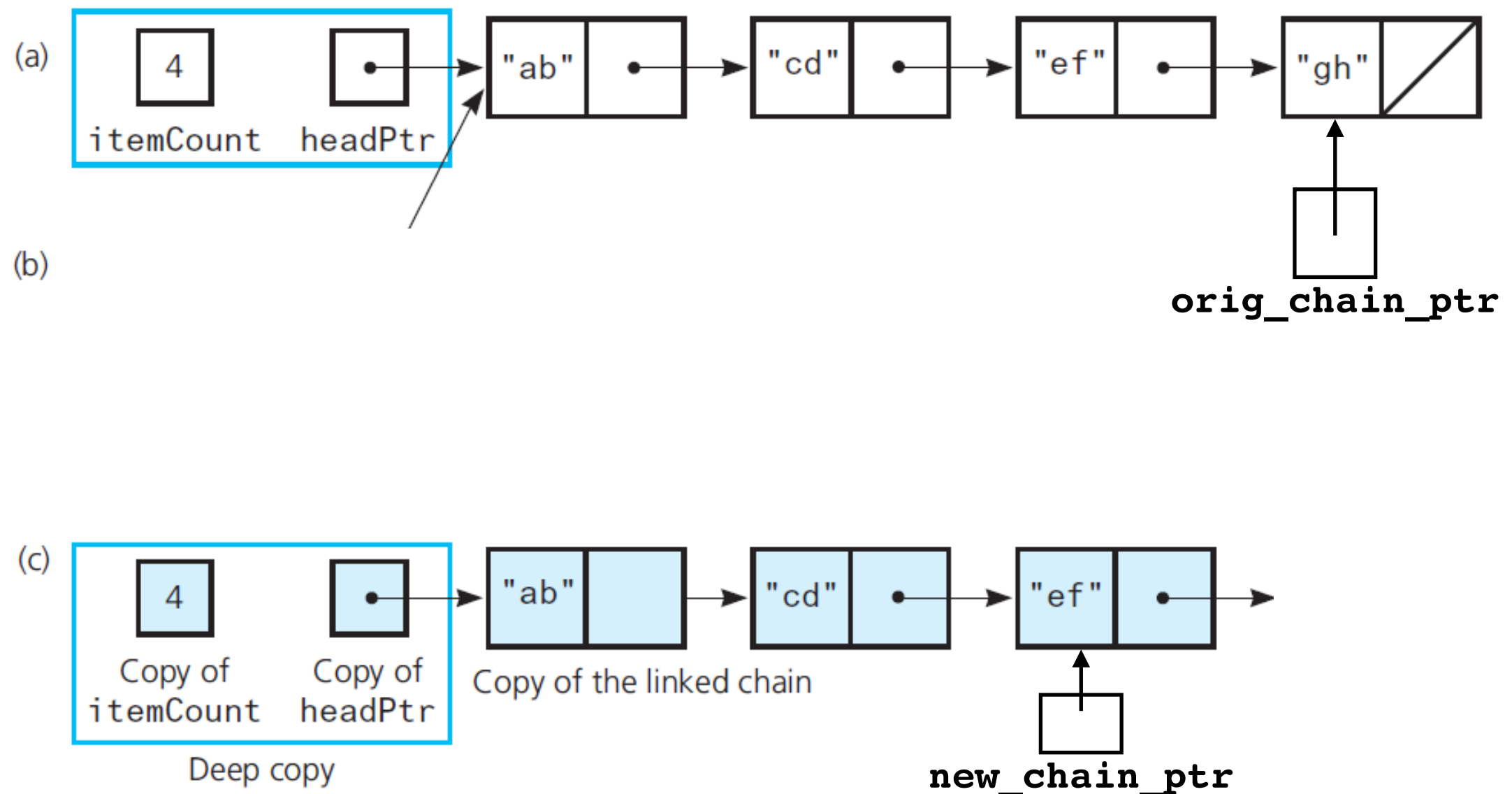
Deep vs Shallow Copy

```
while (orig_chain_ptr != nullptr) {  
    // Get next item from original chain  
    ItemType next_item = orig_chain_ptr->getItem();  
    // Create a new node containing the next item  
    Node<ItemType>* new_node_ptr = new Node<ItemType>(next_item);  
    // Link new node to end of new chain  
    new_chain_ptr->setNext(new_node_ptr);  
    // Advance pointer to new last node  
    new_chain_ptr = new_chain_ptr->getNext();  
    // Advance original-chain pointer  
    orig_chain_ptr = orig_chain_ptr->getNext();  
}
```



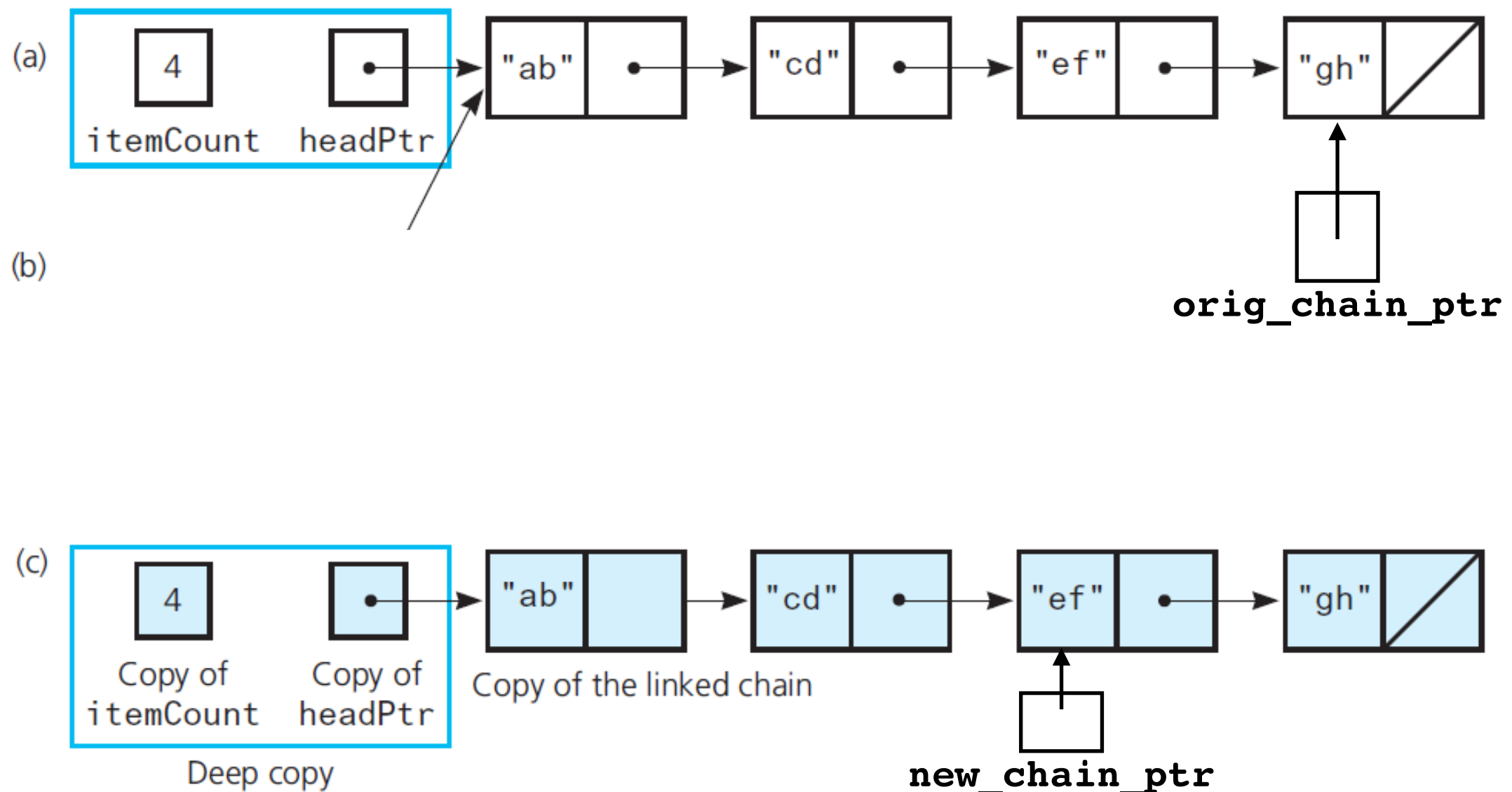
Deep vs Shallow Copy

```
while (orig_chain_ptr != nullptr) {  
    // Get next item from original chain  
    ItemType next_item = orig_chain_ptr->getItem();  
    // Create a new node containing the next item  
    Node<ItemType>* new_node_ptr = new Node<ItemType>(next_item);  
    // Link new node to end of new chain  
    new_chain_ptr->setNext(new_node_ptr);  
    // Advance pointer to new last node  
    new_chain_ptr = new_chain_ptr->getNext();  
    // Advance original-chain pointer  
    orig_chain_ptr = orig_chain_ptr->getNext();  
}
```



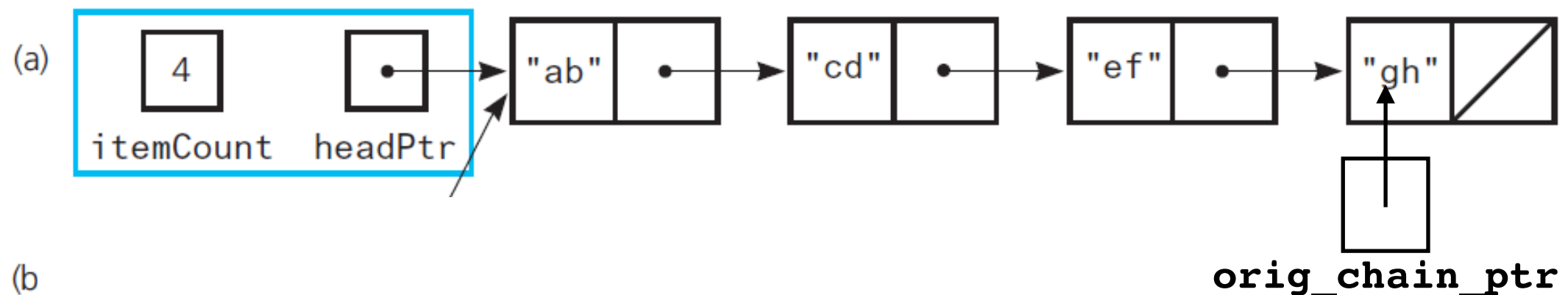
Deep vs Shallow Copy

```
while (orig_chain_ptr != nullptr) {  
    // Get next item from original chain  
    ItemType next_item = orig_chain_ptr->getItem();  
    // Create a new node containing the next item  
    Node<ItemType>* new_node_ptr = new Node<ItemType>(next_item);  
    // Link new node to end of new chain  
    new_chain_ptr->setNext(new_node_ptr);  
    // Advance pointer to new last node  
    new_chain_ptr = new_chain_ptr->getNext();  
    // Advance original-chain pointer  
    orig_chain_ptr = orig_chain_ptr->getNext();  
}
```



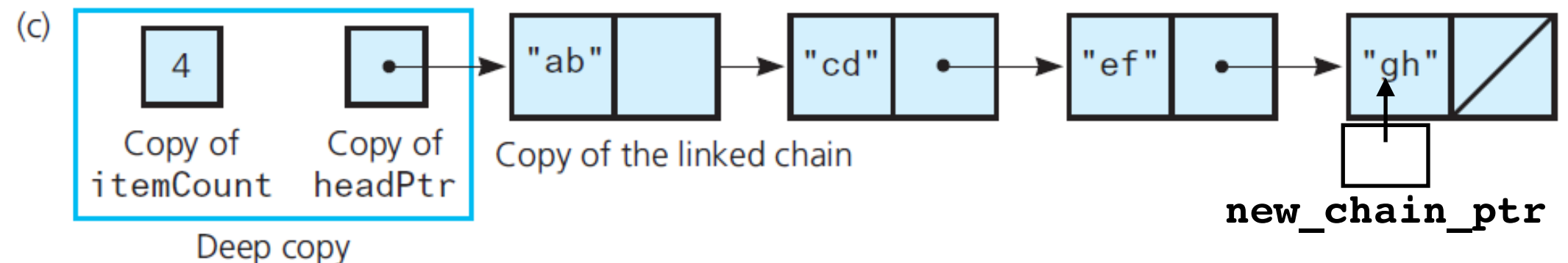
Deep vs Shallow Copy

```
while (orig_chain_ptr != nullptr) {  
    // Get next item from original chain  
    ItemType next_item = orig_chain_ptr->getItem();  
    // Create a new node containing the next item  
    Node<ItemType>* new_node_ptr = new Node<ItemType>(next_item);  
    // Link new node to end of new chain  
    new_chain_ptr->setNext(new_node_ptr);  
    // Advance pointer to new last node  
    new_chain_ptr = new_chain_ptr->getNext();  
    // Advance original-chain pointer  
    orig_chain_ptr = orig_chain_ptr->getNext();  
}
```



(b)

Shallow copy



Efficiency Considerations

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*")

Preferred:

```
myFunction(const MyClass& object);
```

The Class LinkedBag

```
#ifndef LINKED_BAG_H_
#define LINKED_BAG_H_

#include "BagInterface.hpp"
#include "Node.hpp"
```

```
template<typename ItemType>
```

```
class LinkedBag
```

```
{
```

```
public:
```

```
✓ LinkedBag();
```

```
✗ LinkedBag(const LinkedBag<ItemType>& a_bag); // Copy constructor
```

```
✗ ~LinkedBag(); // Destructor
```

```
✓ int getCurrentSize() const;
```

```
✓ bool isEmpty() const;
```

```
✓ bool add(const ItemType& new_entry);
```

```
✗ bool remove(const ItemType& an_entry);
```

```
✗ void clear();
```

```
✗ bool contains(const ItemType& an_entry) const;
```

```
✗ int getFrequencyOf(const ItemType& an_entry) const;
```

```
✗ std::vector<ItemType> toVector() const;
```

```
private:
```

```
Node<ItemType>* head_ptr_; // Pointer to first node
```

```
int item_count_; // 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_
```



Efficient



Expensive

THINK
WORST CASE