

# Copy and Move



Tiziana Ligorio  
Hunter College of The City University of New York

# Today's Plan



Announcements

Copy operations

Move operations

# Announcements

## Project 3:

- `Network` is a template, it manipulates objects of type `T` (or `ItemType`) not accounts.
- We will only ever replace `T` with `Account`, so ok to do things like `T(username, password)` in `Network`.
- A bit of a hack, but stick with it!

# Recap

```
LinkedBag();  
~LinkedBag();    // Destructor  
int  getCurrentSize() const;  
bool isEmpty()  const;  
bool add(const T& new_entry);  
bool remove(const T& an_entry);  
void clear();  
bool contains(const T& an_entry) const;  
int  getFrequencyOf(const T& an_entry) const;  
std::vector<T> toVector() const;
```

What if we need a copy of  
the bag?

# 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

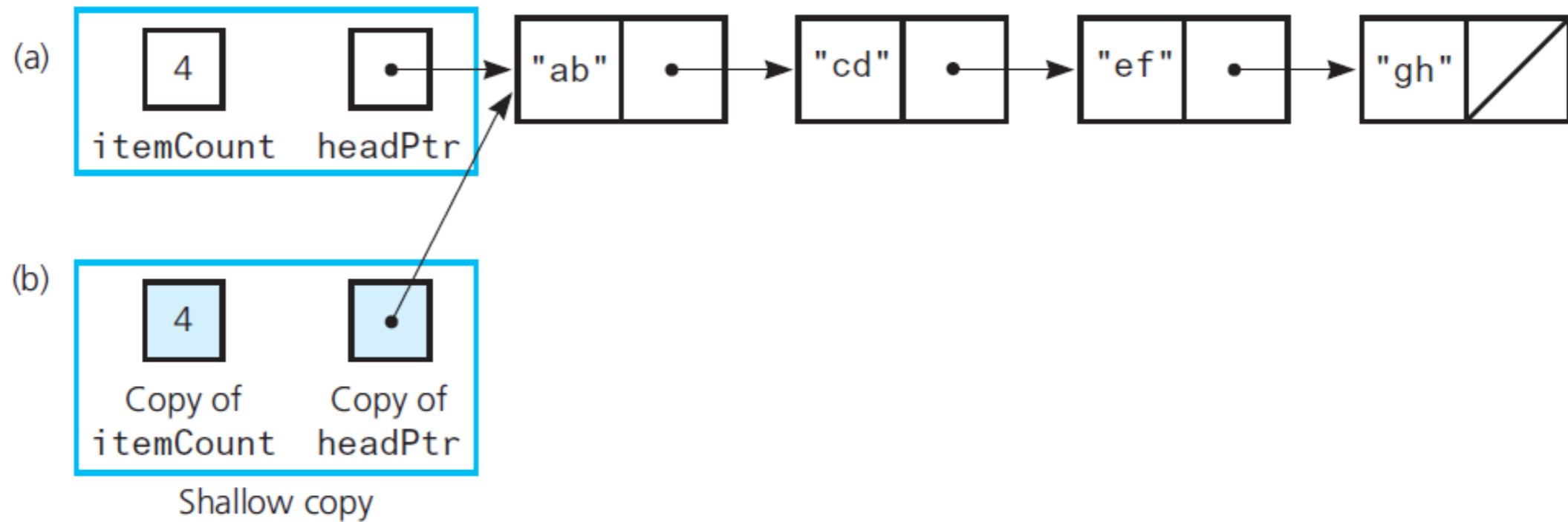
2. Copy an object to **pass by value** as an argument to a function

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

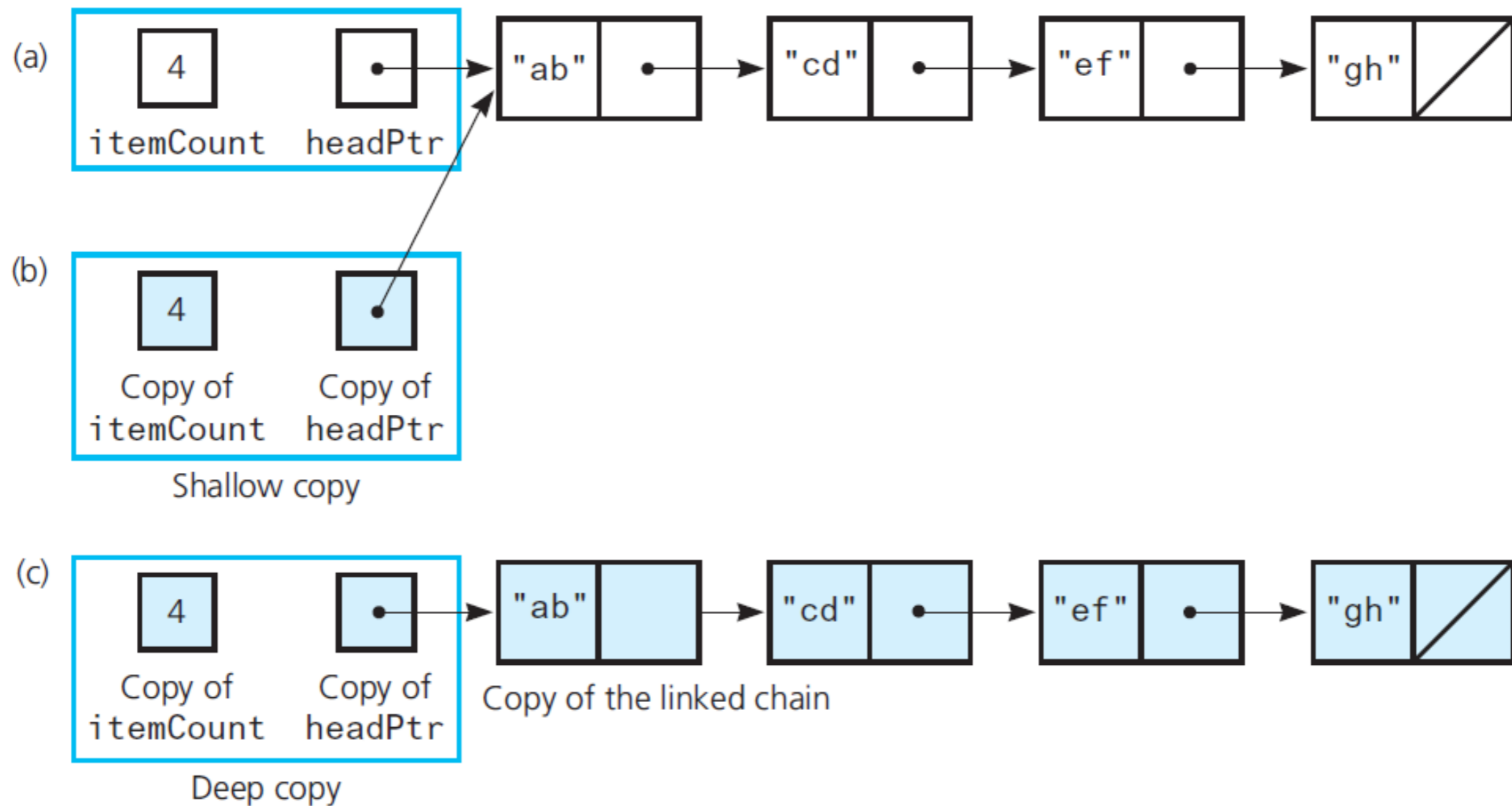
3. Copy an object to be **returned** by a function

```
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

Different functions/call same implementation

Class must explicitly define  
deep copy behavior when  
memory is dynamically allocated

# LinkedList Implementation

```
#include "LinkedList.hpp"
template<class T>
LinkedList<T>::LinkedList(const LinkedList<T>& a_bag)
{
    item_count_ = a_bag.item_count_;
    Node<T>* 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<T>();
        head_ptr_->setItem(orig_chain_ptr->getItem());

        // Copy remaining nodes
        Node<T>* 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
            T next_item = orig_chain_ptr->getItem();
            // Create a new node containing the next item
            Node<T>* new_node_ptr = new Node<T>(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
```

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.  
`LinkedList<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

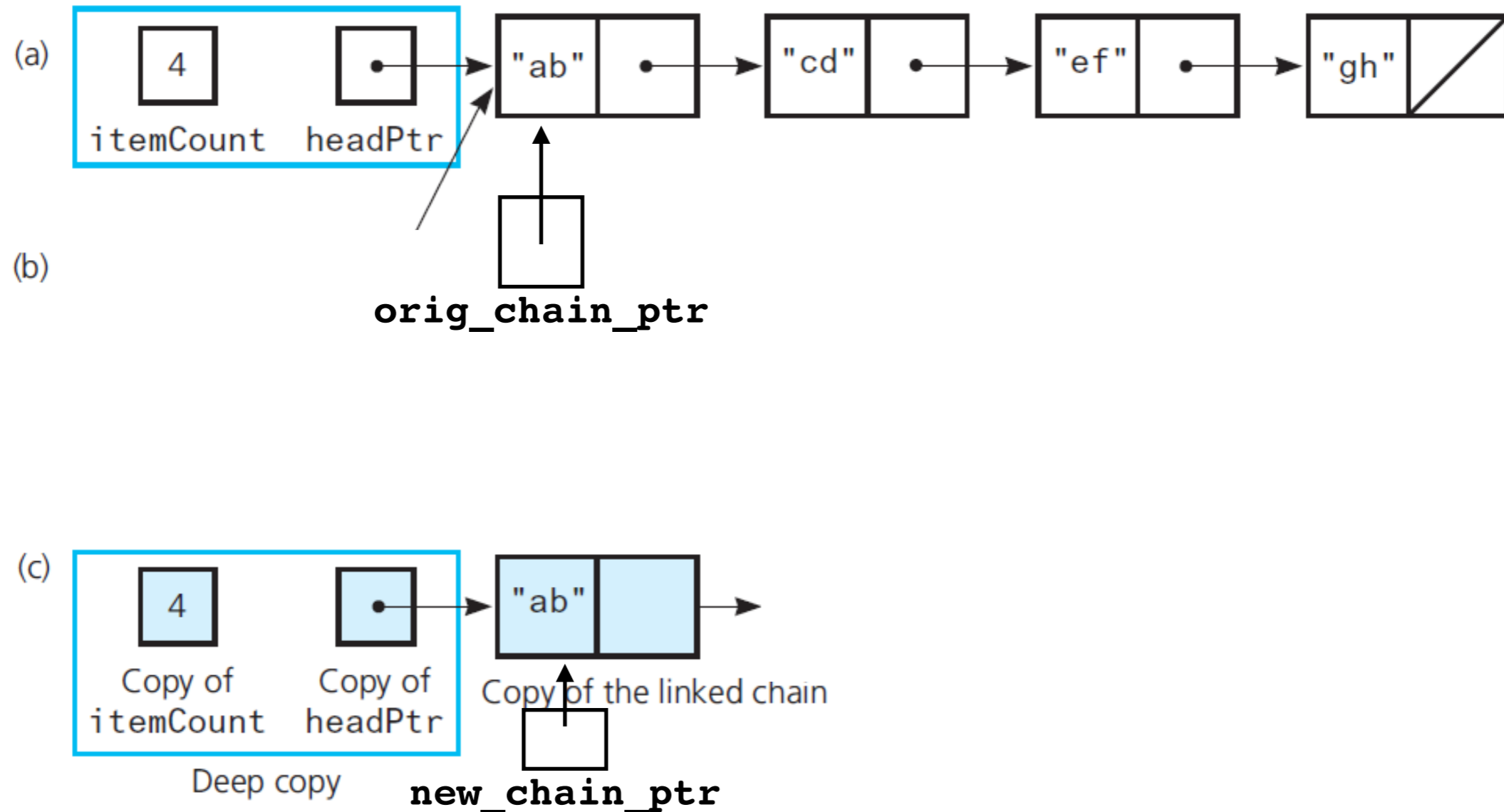
Advance pointer traversing original chain

Signal last node

# Deep vs Shallow Copy

```
// Copy first node
head_ptr_ = new Node<T>();
head_ptr_->setItem(orig_chain_ptr->getItem());

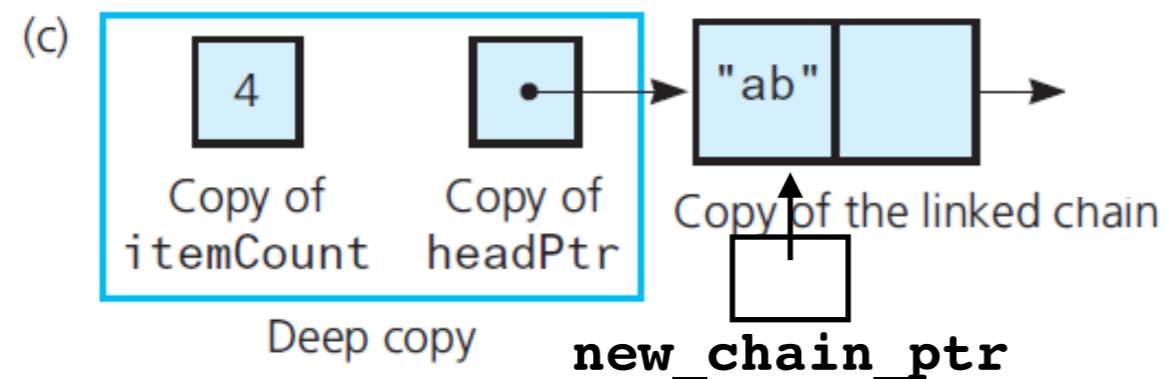
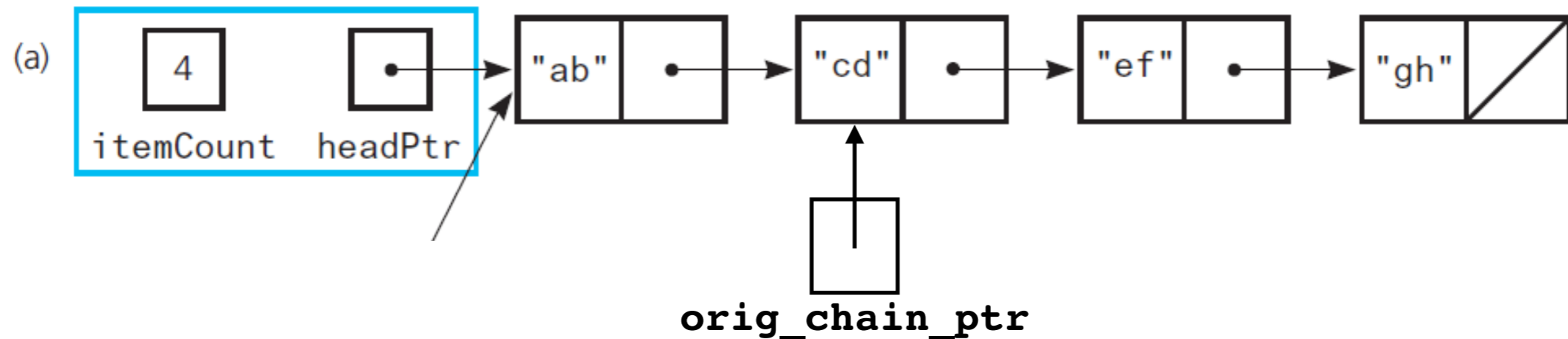
// Copy remaining nodes
Node<T>* new_chain_ptr = head_ptr_; //
Points to last node in new chain
orig_chain_ptr = orig_chain_ptr->getNext();
```



# Deep vs Shallow Copy

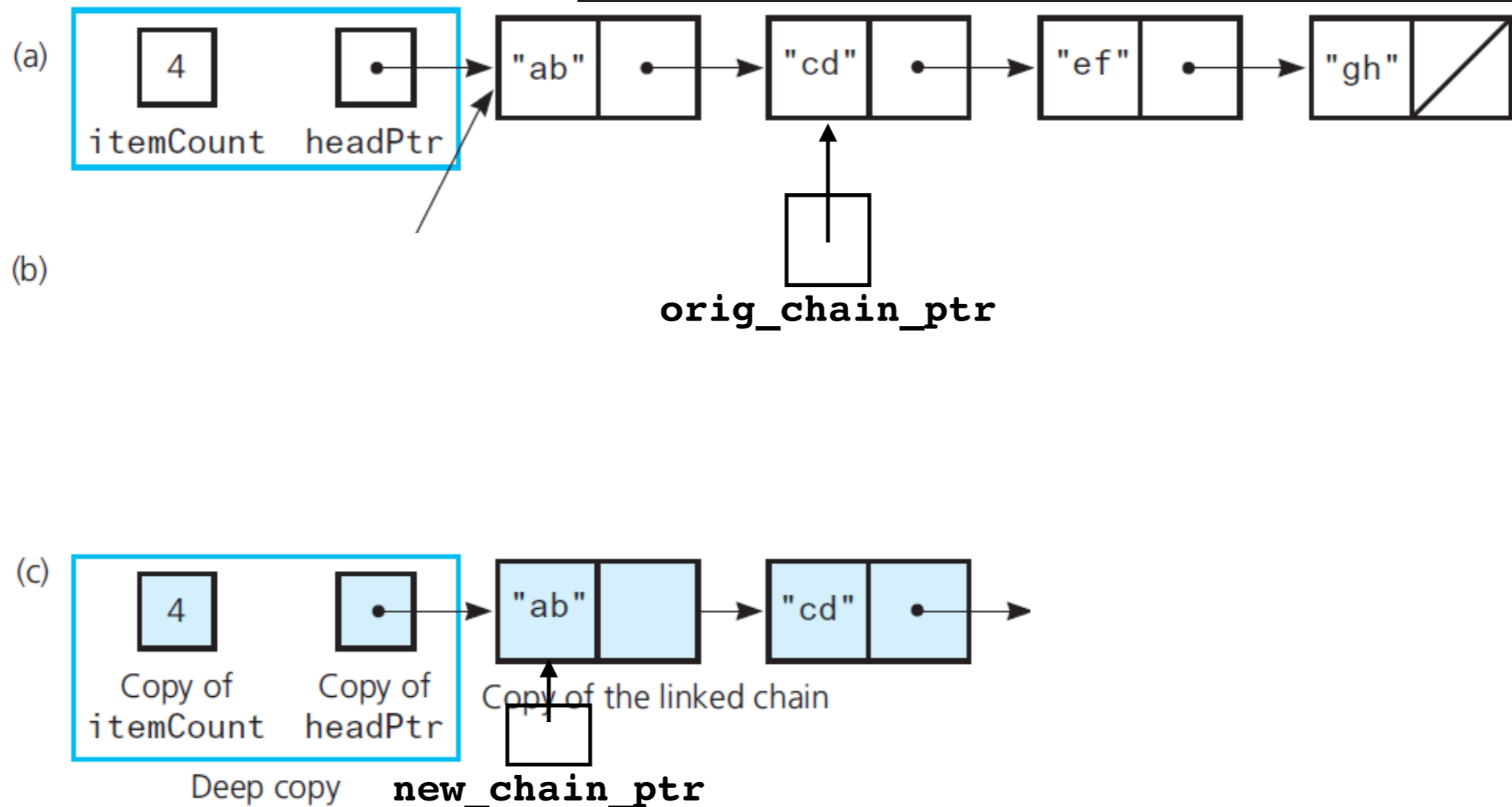
```
// Copy first node
head_ptr_ = new Node<T>();
head_ptr_->setItem(orig_chain_ptr->getItem());

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



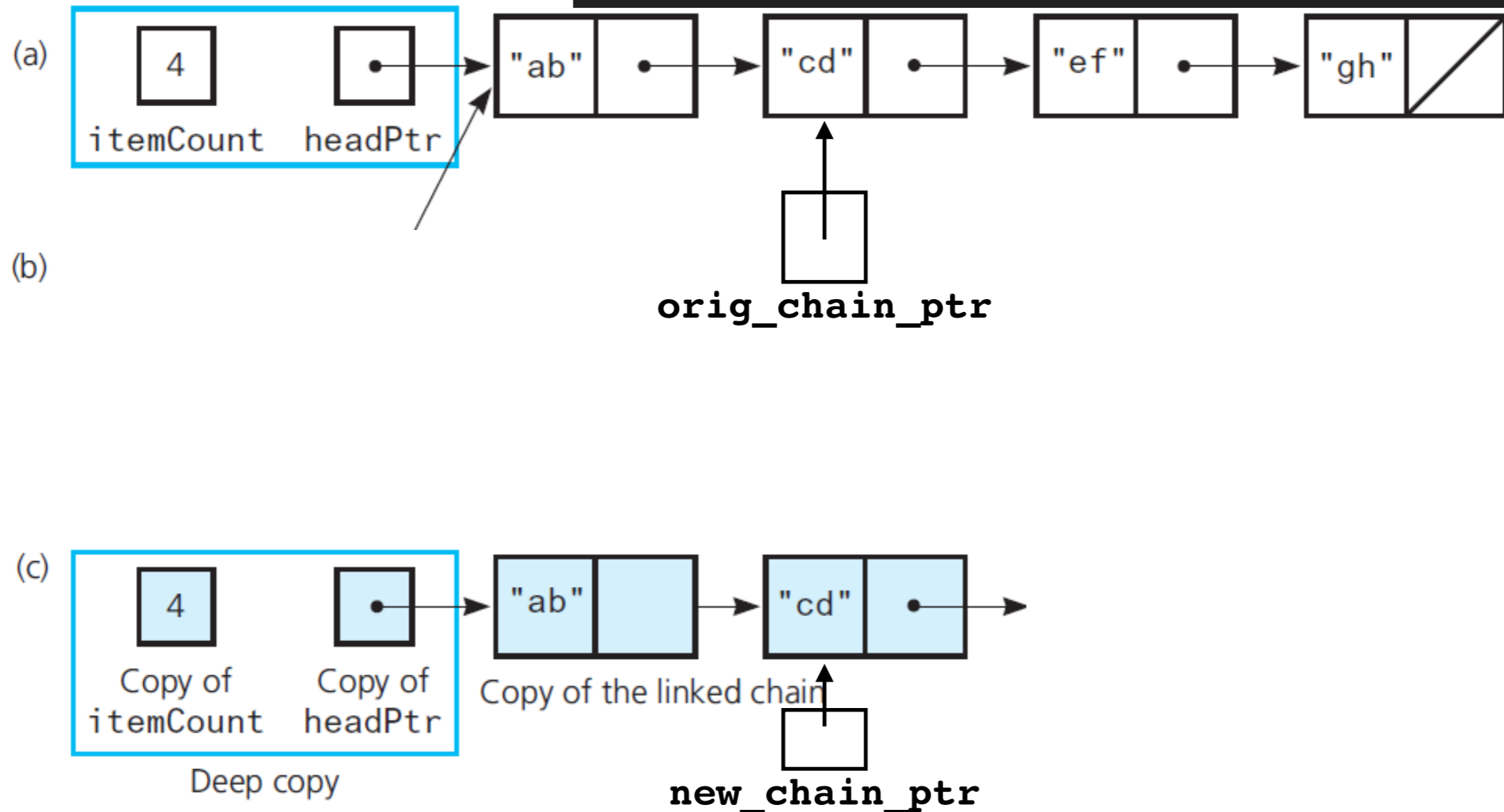
# Deep vs Shallow Copy

```
while (orig_chain_ptr != nullptr)
{
    // Get next item from original chain
    T next_item = orig_chain_ptr->getItem();
    // Create a new node containing the next item
    Node<T>* new_node_ptr = new Node<T>(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();
}
```



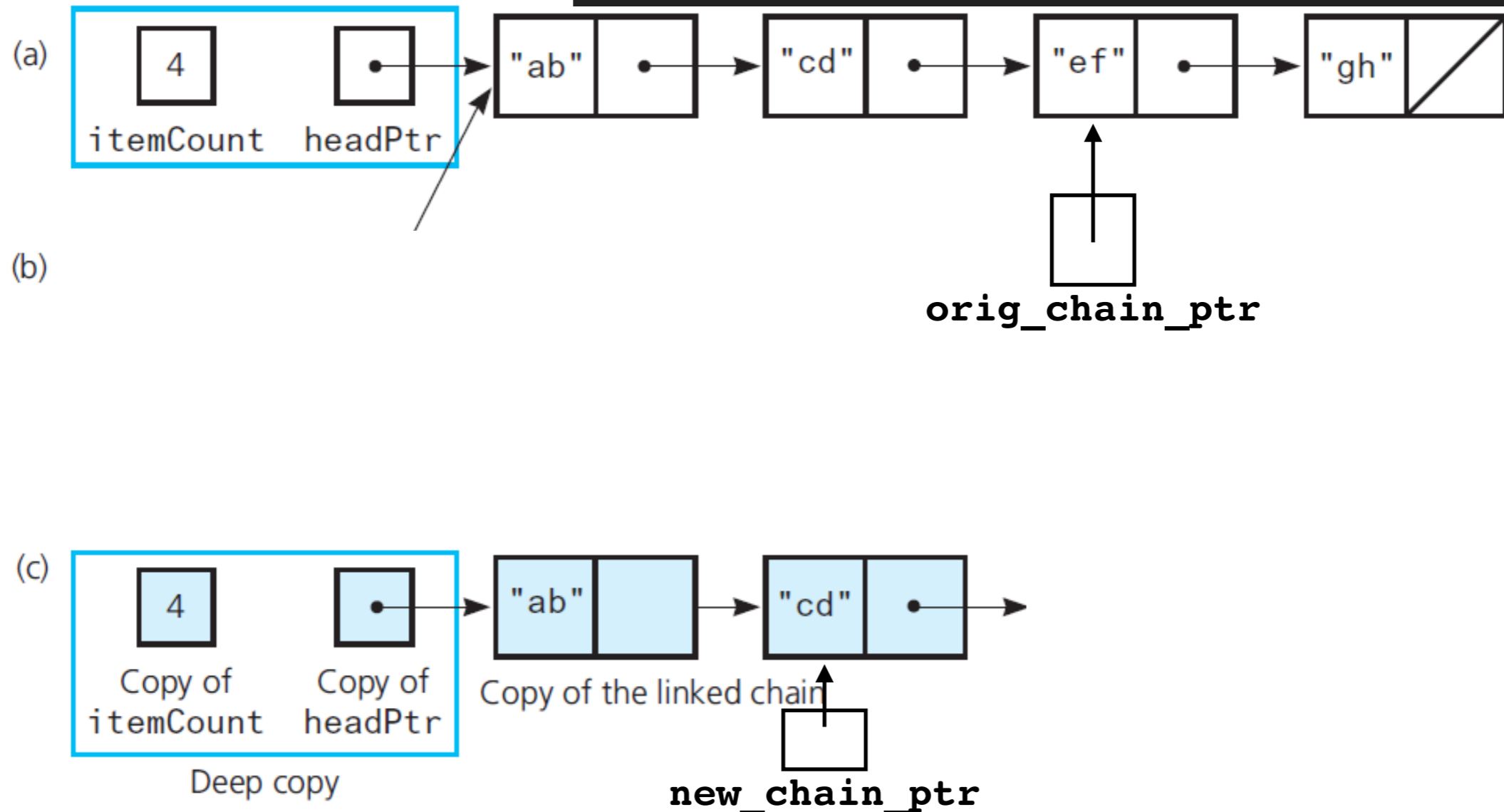
# Deep vs Shallow Copy

```
while (orig_chain_ptr != nullptr)
{
    // Get next item from original chain
    T next_item = orig_chain_ptr->getItem();
    // Create a new node containing the next item
    Node<T>* new_node_ptr = new Node<T>(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

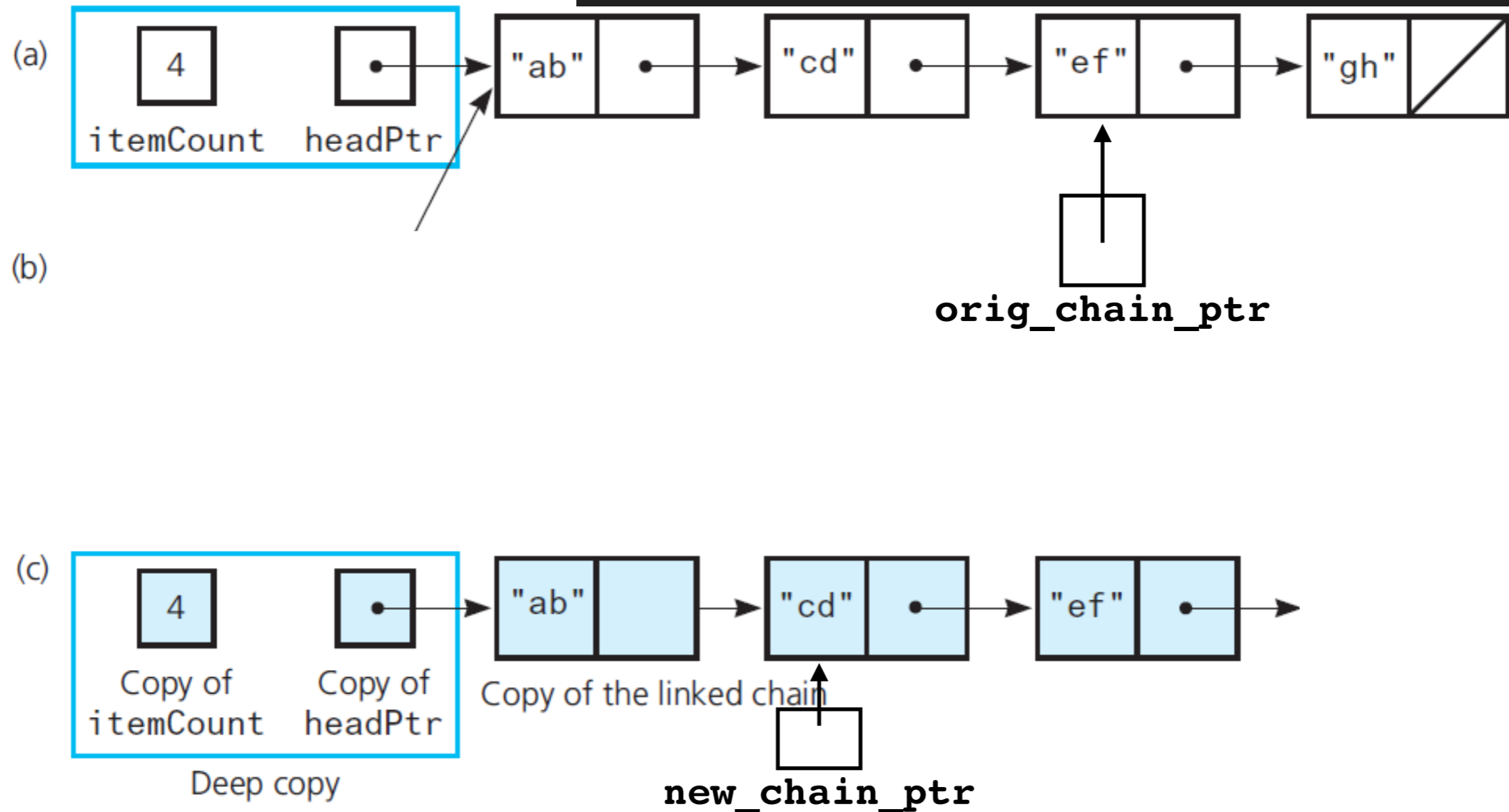
```
while (orig_chain_ptr != nullptr)
{
    // Get next item from original chain
    T next_item = orig_chain_ptr->getItem();
    // Create a new node containing the next item
    Node<T>* new_node_ptr = new Node<T>(next_item);
    // Link new node to end of new chain
    new_chain_ptr->setNext(new_node_ptr);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





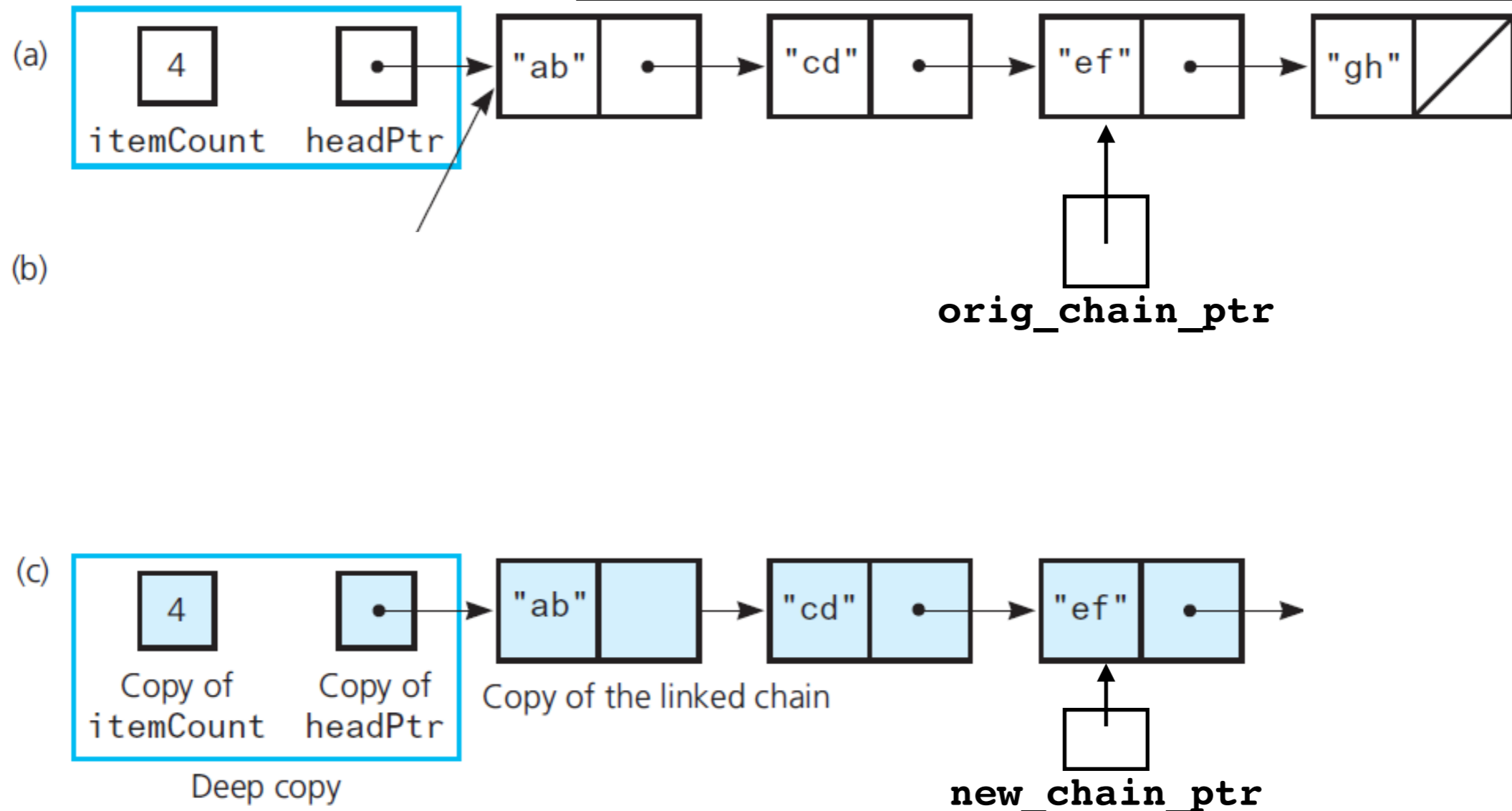
# Deep vs Shallow Copy

```
while (orig_chain_ptr != nullptr)
{
    // Get next item from original chain
    T next_item = orig_chain_ptr->getItem();
    // Create a new node containing the next item
    Node<T>* new_node_ptr = new Node<T>(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();
}
```



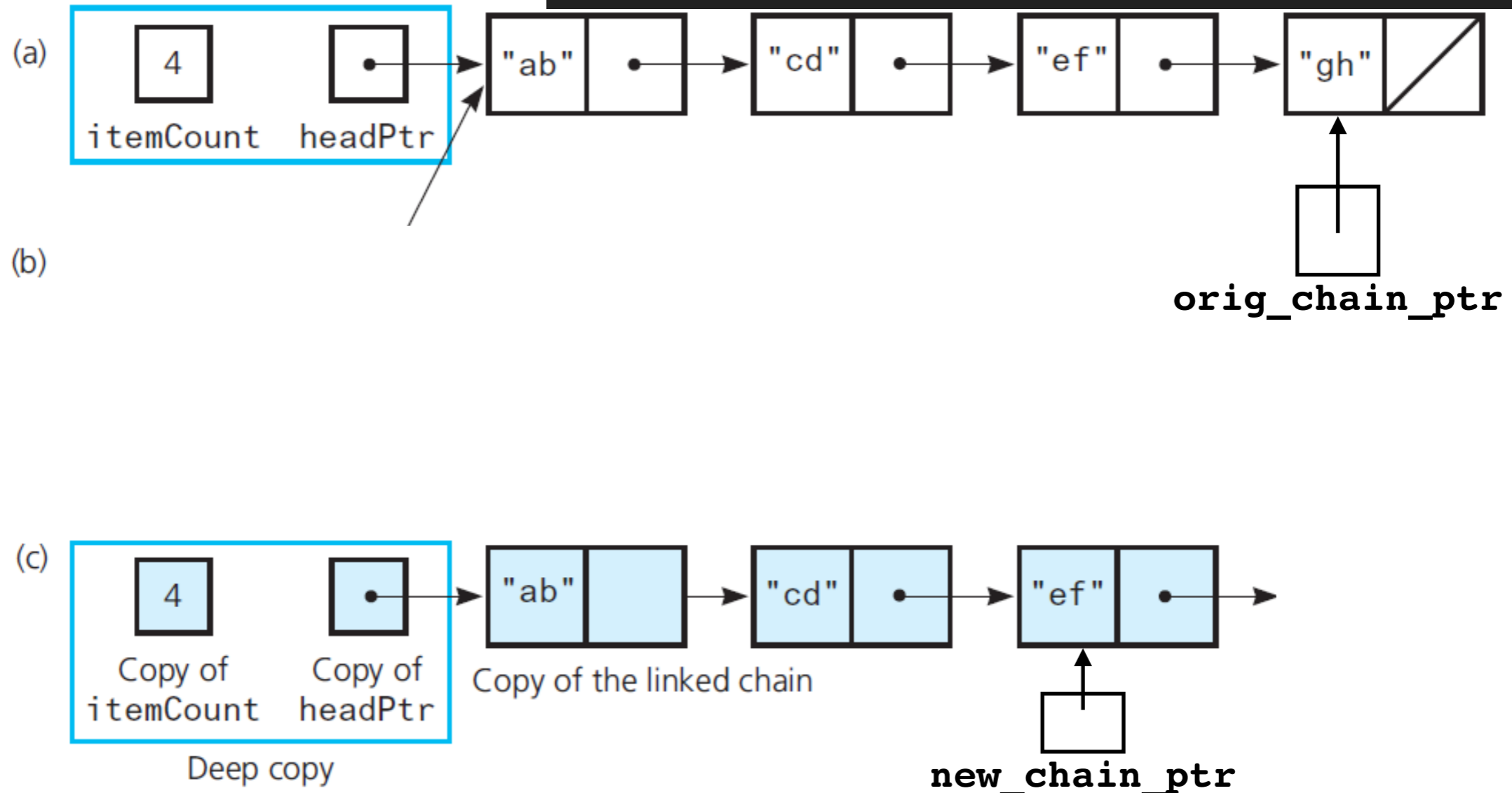
# Deep vs Shallow Copy

```
while (orig_chain_ptr != nullptr)
{
    // Get next item from original chain
    T next_item = orig_chain_ptr->getItem();
    // Create a new node containing the next item
    Node<T>* new_node_ptr = new Node<T>(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();
}
```



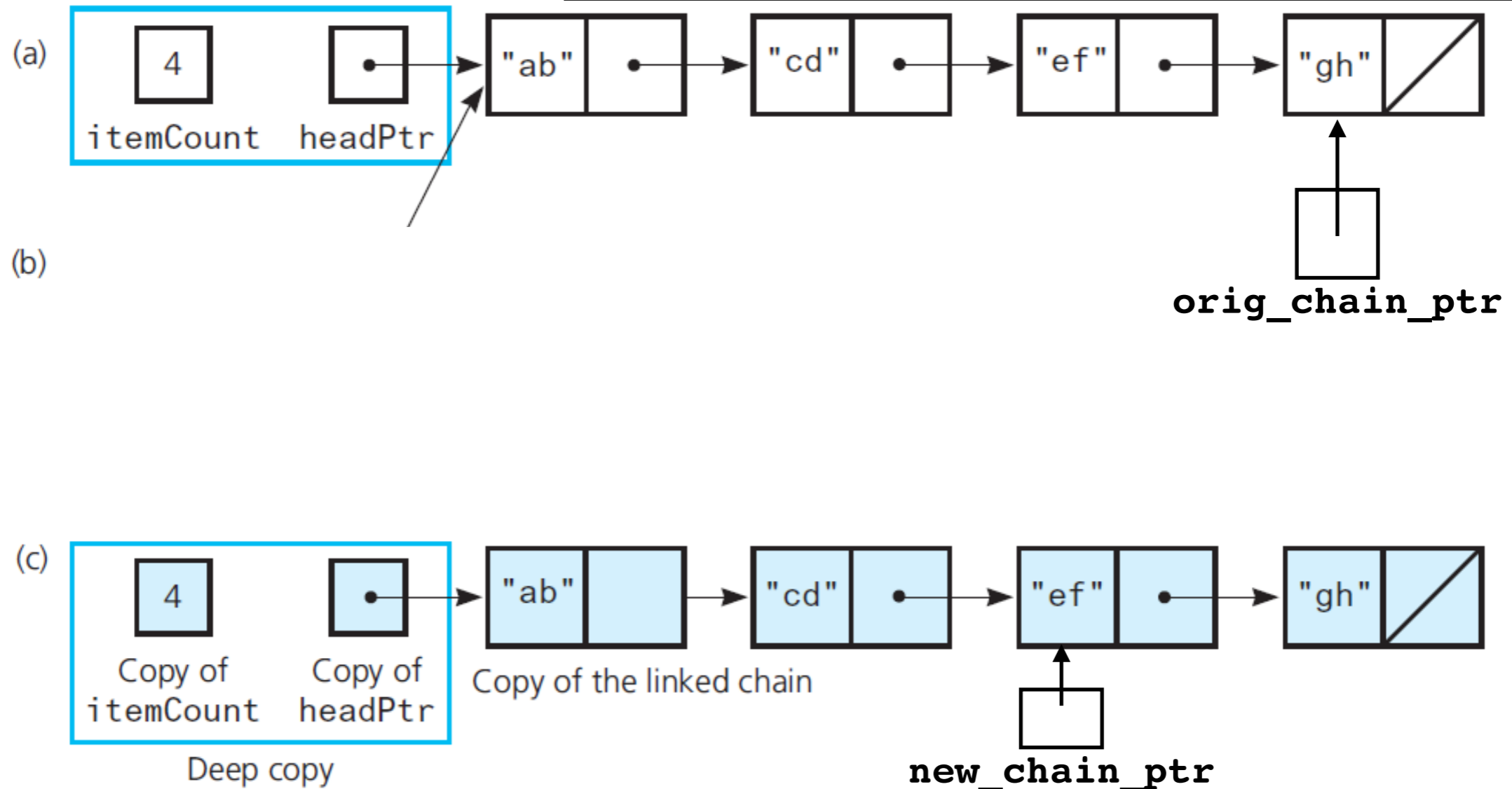
# Deep vs Shallow Copy

```
while (orig_chain_ptr != nullptr)
{
    // Get next item from original chain
    T next_item = orig_chain_ptr->getItem();
    // Create a new node containing the next item
    Node<T>* new_node_ptr = new Node<T>(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();
}
```



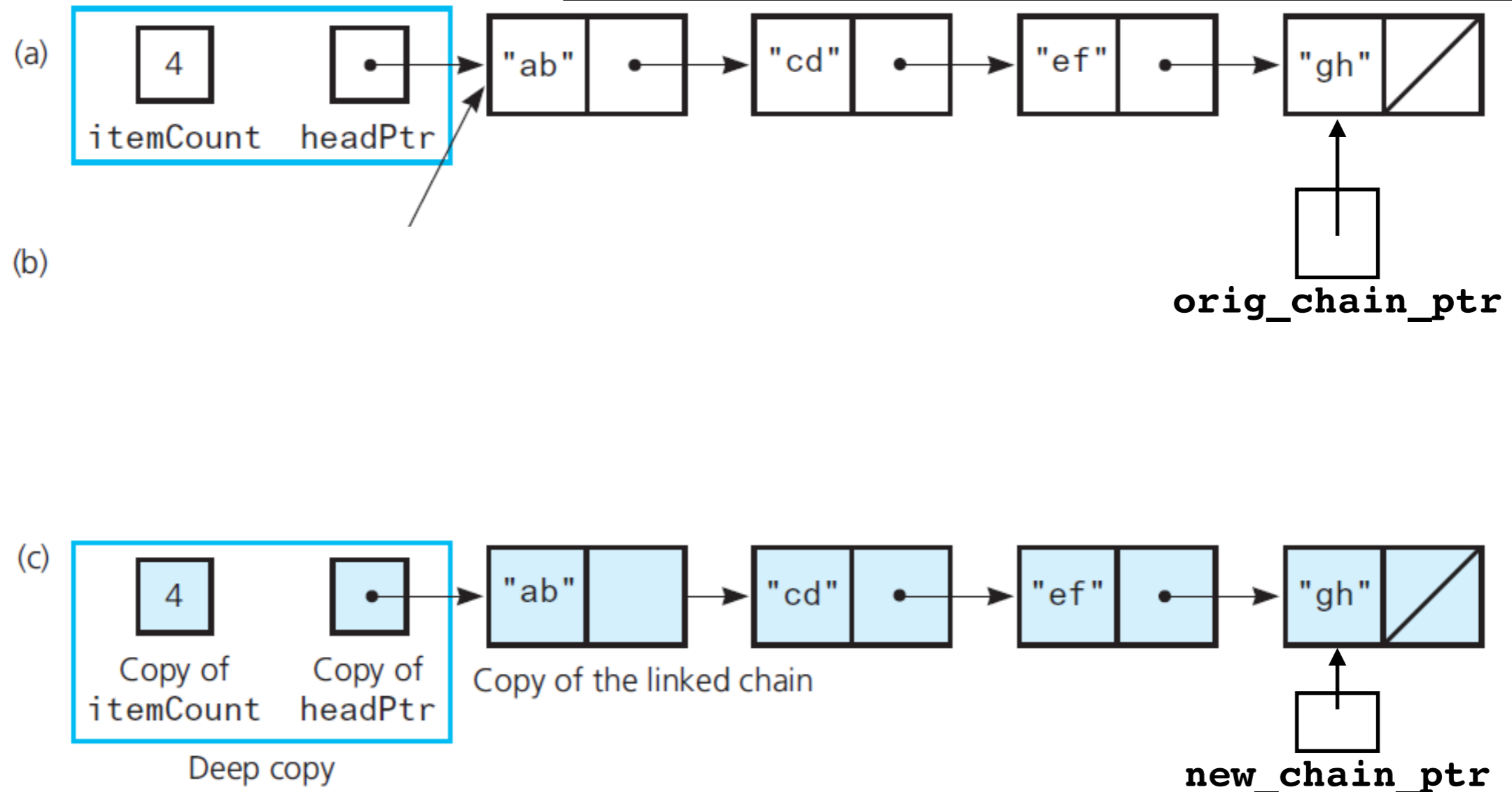
# Deep vs Shallow Copy

```
while (orig_chain_ptr != nullptr)
{
    // Get next item from original chain
    T next_item = orig_chain_ptr->getItem();
    // Create a new node containing the next item
    Node<T>* new_node_ptr = new Node<T>(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
    T next_item = orig_chain_ptr->getItem();
    // Create a new node containing the next item
    Node<T>* new_node_ptr = new Node<T>(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();
}
```



# 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" )  $O(n)$
- Traverse entire chain to destroy (  $n$  "steps" )  $O(n)$

Preferred call by const reference:

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

But what if you want to modify object?

Also, passing to function is not the only situation for copy

(e.g., `MyClass two = one;`

or `one = two;`)

# Move Semantics

Copying can be time/space consuming, especially if large amount of data

Copying often involves making copy and destroying original (e.g, pass by value, return by value, old-school swap with intermediate): **inefficient**

More efficient to **transfer ownership** of resources to another object

# *lvalues* and *rvalues*

## **lvalue = rvalue**

Examples:

```
int x = 2;  
int y = x+1;  
x = y;  
x = y + z;  
string msg = "hello";  
bool pass = computeGrades(student);
```

The return value,  
not the function

**Lvalues** can be referred to by **name**, **pointer** or **lvalue reference**: i.e. they **have an address**

**Rvalues** are **literals** or **temporary objects** copied into or returned by functions, they **don't have an address** and **cannot have a value assigned to it**

Can have **lvalue** and **rvalue** of same type.



# Move Semantics

Rvalues are eligible for **move operations**

After object *y* is **moved** into object *x*:

- *x* is equivalent to the former value of *y*
- *y* is in a special state called the *moved-from state*
- Object in *moved-from state* can only be **reassigned** or **destroyed** (becomes an rvalue)
- **rvalue** is **semantically temporary**, thus it is more likely to be put in temporary memory or optimized



# std::move()

A type-cast

Converts an lvalue to an rvalue

Allows the **efficient transfer of resources** from the moved object to another

```
y = std::move(x);
```

x is now treated as an rvalue

# Old-school swap

```
void swap(vector<string>& x, vector<string>& y)
{
    vector<string> temp = x;
    x = y;
    y = temp;
}
```



x



y

# Old-school swap

```
void swap(vector<string>& x, vector<string>& y)
{
    vector<string> temp = x;
    x = y;
    y = temp;
}
```

x

y

temp

# Old-school swap

```
void swap(vector<string>& x, vector<string>& y)
{
    vector<string> temp = x;
    x = y;
    y = temp;
}
```

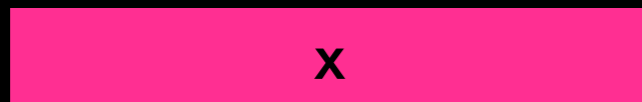
x

y

temp

# Old-school swap

```
void swap(vector<string>& x, vector<string>& y)
{
    vector<string> temp = x;
    x = y;
    y = temp;
}
```



# Old-school swap

```
void swap(vector<string>& x, vector<string>& y)
{
    vector<string> temp = x;
    x = y;
    y = temp;
}
```



# std::swap

```
void swap(vector<string>& x, vector<string>& y)
{
    vector<string> temp = std::move(x);
    x = std::move(y);
    y = std::move(temp);
}
```



x



y



# std::swap

```
void swap(vector<string>& x, vector<string>& y)
{
    vector<string> temp = std::move(x);
    x = std::move(y);
    y = std::move(temp);
}
```

temp

move(x)

y

# std::swap

```
void swap(vector<string>& x, vector<string>& y)
{
    vector<string> temp = std::move(x);
    x = std::move(y);
    y = std::move(temp);
}
```

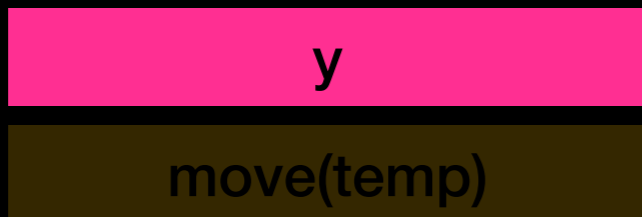
temp

x

move(y)

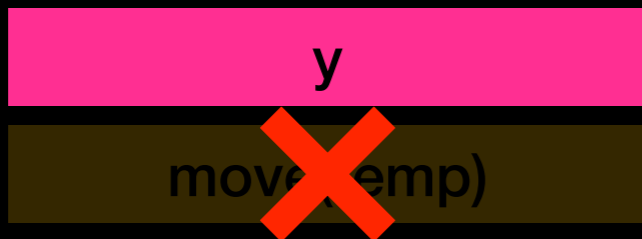
# std::swap

```
void swap(vector<string>& x, vector<string>& y)
{
    vector<string> temp = std::move(x);
    x = std::move(y);
    y = std::move(temp);
}
```



# std::swap

```
void swap(vector<string>& x, vector<string>& y)
{
    vector<string> temp = std::move(x);
    x = std::move(y);
    y = std::move(temp);
}
```



# std::swap

Now part of the STL

Can use for any STL type

Example:

```
vector<string> x;  
vector<string> y;  
//do stuff ...  
  
std::swap(x,y);
```

# Move Constructor and Assignment

Triggered similarly to copy constructor and assignment operator, but **right hand side is rvalue**

Explicitly implement move semantics for the class

When defining one should probably **define all 5**  
(Copy constructor, Move constructor, Assignment operator, Move-assignment operator, Destructor)

More in CSci 335

# Move Constructor

**1. Initialize** one object from rvalue

```
MyClass one = rvalue;
```

Implements move semantics  
instead of copy when:

**2. Pass by rvalue reference**

```
void myFunction(MyClass&& arg) {  
    /* ... */  
}
```

rvalue reference

Performs member-wise moves on non-static members of the class  
Compiler will NOT generate move constructor if any copy operation is explicitly defined

# LinkedBag Implementation

The move constructor  
A constructor whose parameter is an **rvalue** reference of the same class

```
#include "LinkedBag.hpp"
template<class T>
LinkedBag<T>::LinkedBag(LinkedBag<T>&& a_bag) :
item_count_{a_bag.item_count_},
head_ptr_{a_bag.head_ptr_}
{
    a_bag.item_count_ = 0;
    a_bag.head_ptr_ = nullptr;
} // end move constructor
```

rvalue reference

Move nodes to this bag

No longer points to moved bag

**O(1)**



# LinkedBag Implementation

The move assignment operator `Std::swap` is an  $O(1)$  operation, swap with **rvalue** which is about to be destroyed by the system anyway

```
#include "LinkedBag.hpp"
template<class T>
void LinkedBag<T>::operator=(LinkedBag<T>&& rhs)
{
    std::swap(item_count_, rhs.item_count_);
    std::swap(head_ptr_, rhs.head_ptr_);
} // end move assignment operator
```

rvalue reference

Swap bags

**$O(1)$**

# The Class LinkedBag

```
#ifndef LINKED_BAG_H_
#define LINKED_BAG_H_

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

template<class T>
class LinkedBag
{
public:
  ✓ LinkedBag();
  ✗ LinkedBag(const LinkedBag<T>& a_bag); // Copy constructor
  ✓ LinkedBag(LinkedBag<T>&& a_bag); // Move constructor
  ✗ ~LinkedBag(); // Destructor
  ✓ int getCurrentSize() const;
  ✓ bool isEmpty() const;
  ✓ bool add(const T& new_entry);
  ✗ bool remove(const T& an_entry);
  ✗ void clear();
  ✗ bool contains(const T& an_entry) const;
  ✗ int getFrequencyOf(const T& an_entry) const;
  ✗ std::vector<T> toVector() const;

private:
  Node<T>* 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<T>* getPointerTo(const T& target) const;
}; // end LinkedBag

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

O(1)



O(n)

