

Linked-Based Implementation

Tiziana Ligorio
tligorio@hunter.cuny.edu

Today's Plan



Announcements

A quick review of
pointers

Linked-Based
Implementation

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

```

Make sure you do this if not assigning a value!

• • •

Run-time Stack

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

```

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

```

Make sure you do this if not assigning a value!

```

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

```

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
...

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

Make sure you do this if not assigning a value!

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

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 instantiated objects
```

• • •

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

To access member functions
in place of . operator

Deallocating Memory

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

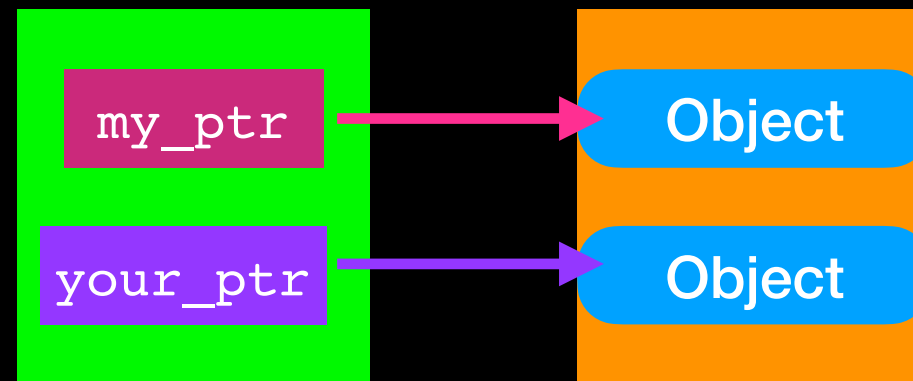
Deletes **the object**
pointed to by p

Must do this!!!

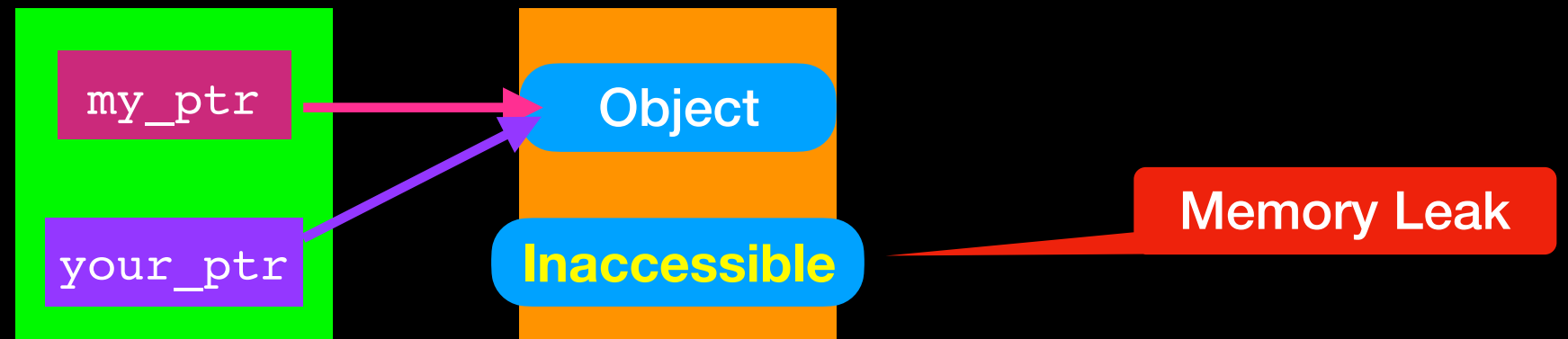
Avoid Memory Leaks (1)

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
```



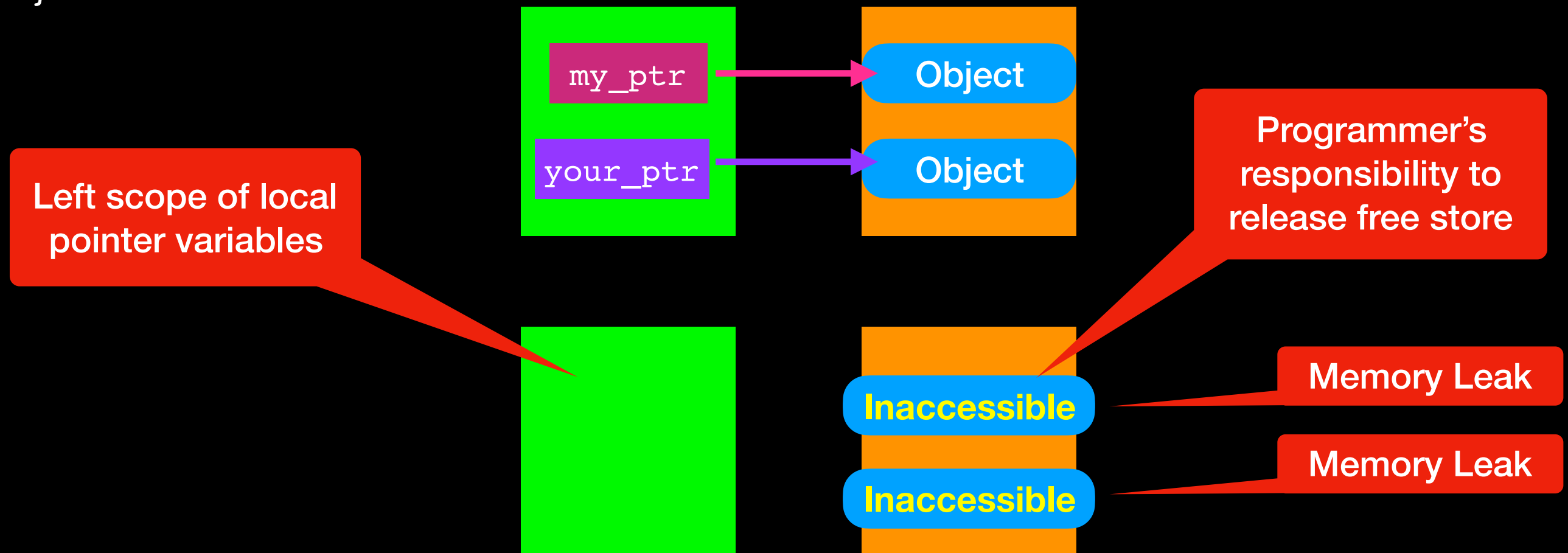
```
your_ptr = my_ptr;
```



Avoid Memory Leaks (2)

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
}
```

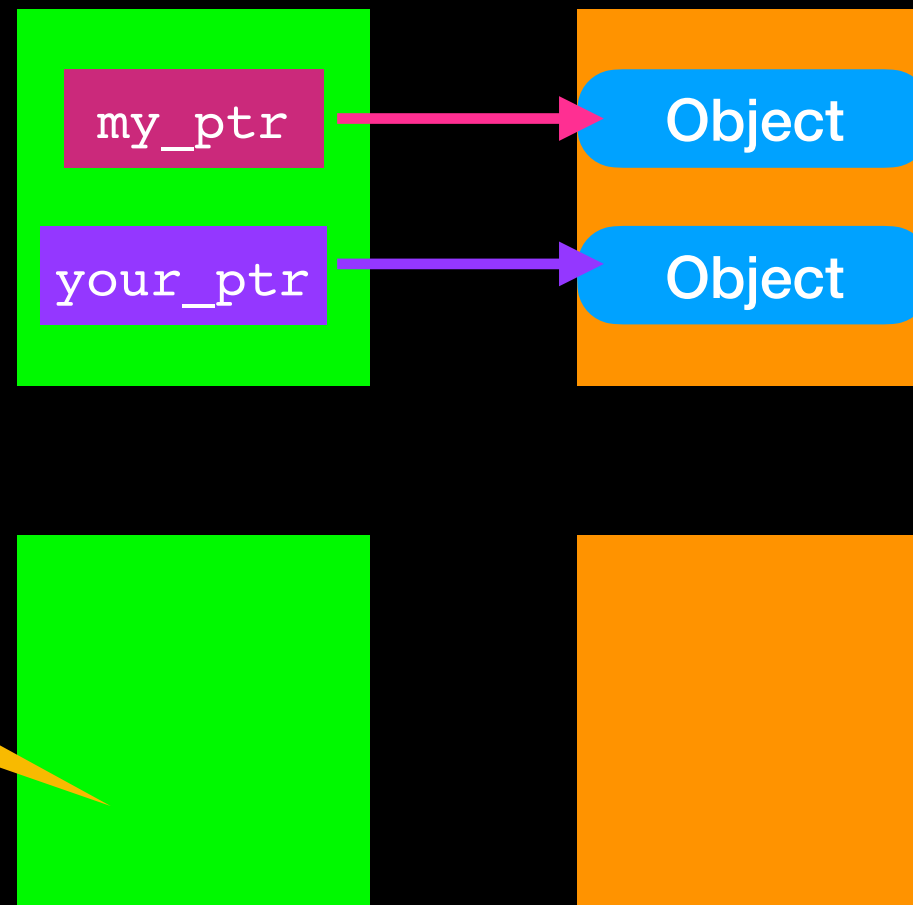


Avoid Memory Leaks (2)

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

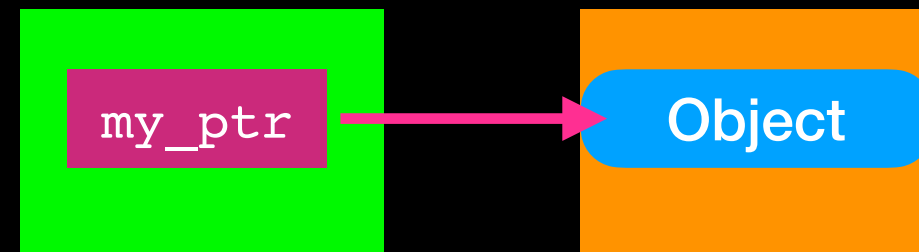
```
void leakyFunction(){  
    dataType *my_ptr = new dataType;  
    dataType *your_ptr = new dataType;  
    // do stuff with my_ptr and your_ptr  
    delete my_ptr;  
    delete your_ptr;  
}
```

Left scope of local
pointer variables
but deleted dynamic
objects first

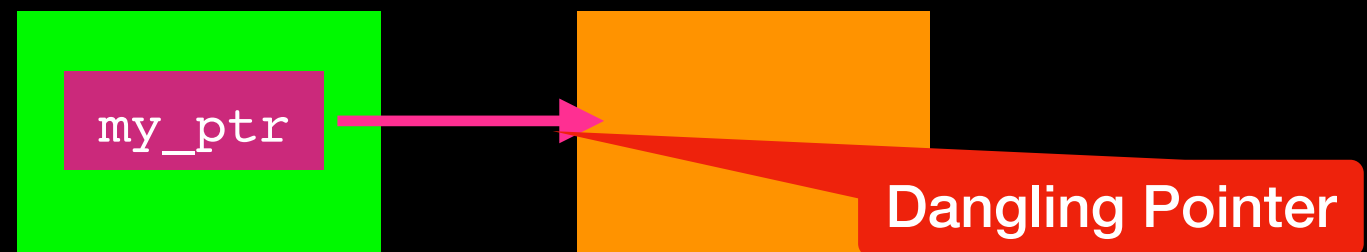


Avoid Dangling Pointers

Pointer variable that no longer references a valid object



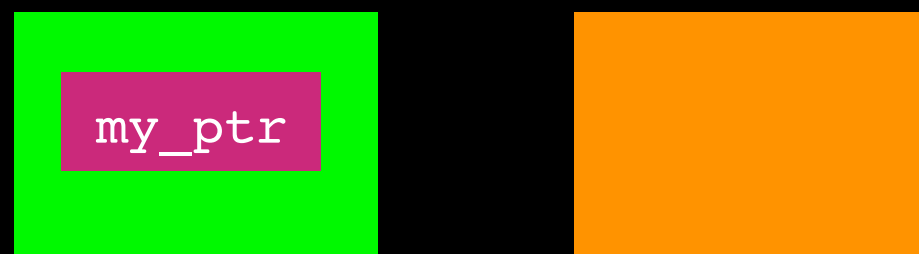
```
delete my_ptr;
```



Fix

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

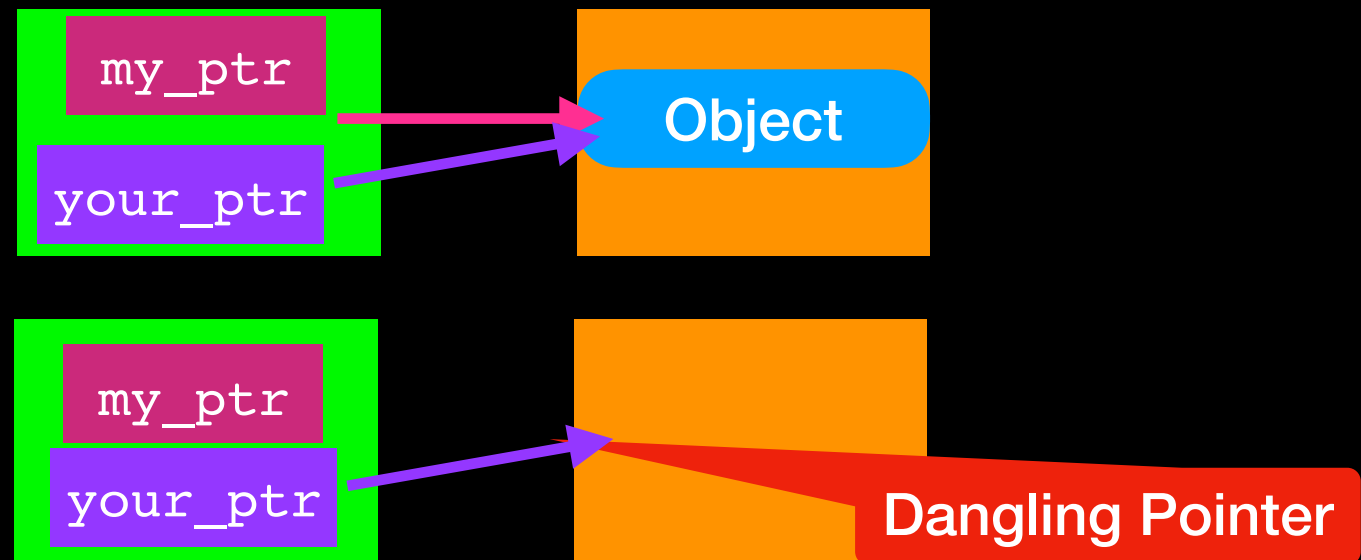
Must do this!!!



Avoid Dangling Pointers

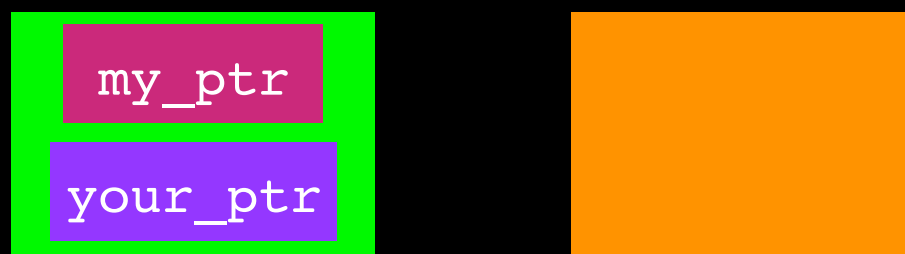
Pointer variable that no longer references a valid object

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



Fix

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



Must set all pointers to nullptr!!!

What is wrong with the following code?

```
void someFunction()  
{  
    int* p = new int[5];  
    int* q = new int[10];  
  
    p[2] = 9;  
    q[2] = p[2]+5;  
    p[0] = 8;  
    q[7] = 15;  
  
    std::cout<< p[2] << " " << q[2] << std::endl;  
    q = p;  
    std::cout<< p[0] << " " << q[7] << std::endl;  
  
}
```


What is wrong with the following code?

```
void someFunction()  
{  
    int* p = new int[5];  
    int* q = new int[10];
```

```
    p[2] = 9;  
    q[2] = p[2]+5;  
    p[0] = 8;  
    q[7] = 15;
```

MEMORY LEAK:
int[10] lost on heap

SEGMENTATION FAULT
int[5] index out of range

```
    std::cout<< p[2] << " " << q[2] << std::endl;  
    q = p;  
    std::cout<< p[0] << " " << q[7] << std::endl;
```

MEMORY LEAK:
Did not delete int[5]
before exiting function

```
}
```

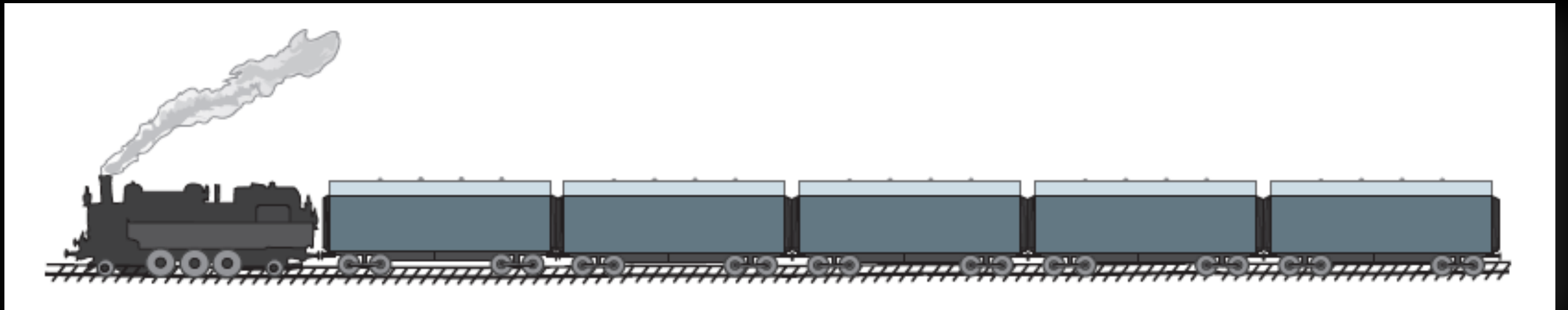
Let's try a different
implementation for Bag

Link-Based Implementation

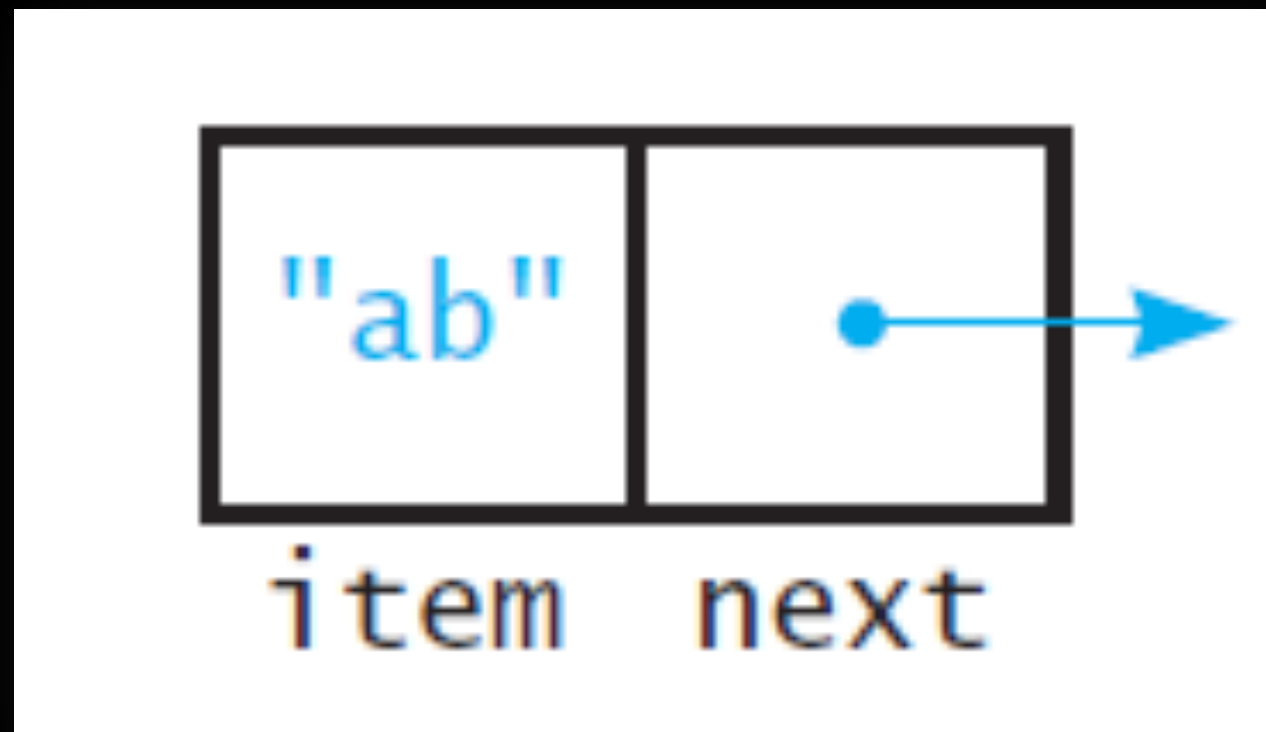
Data Organization

Place data within a **Node** object

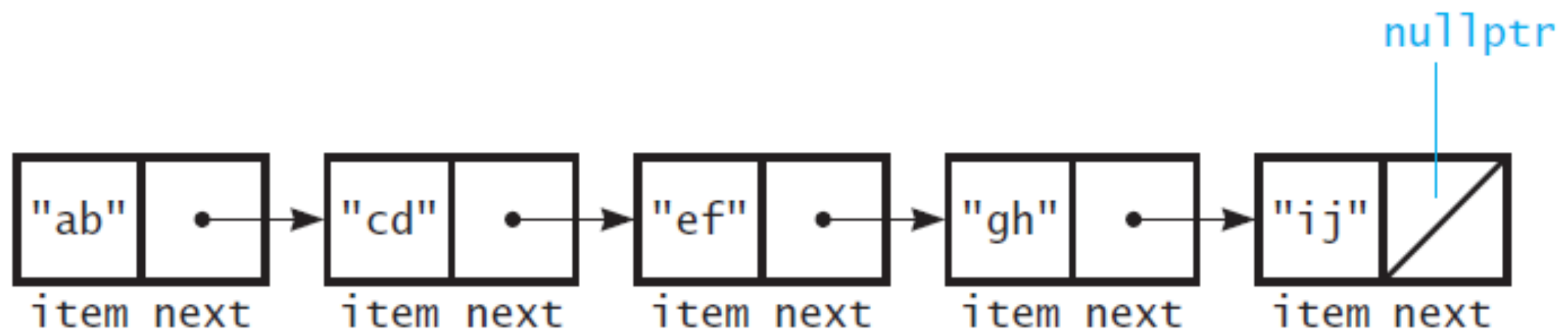
Link nodes into a **chain**



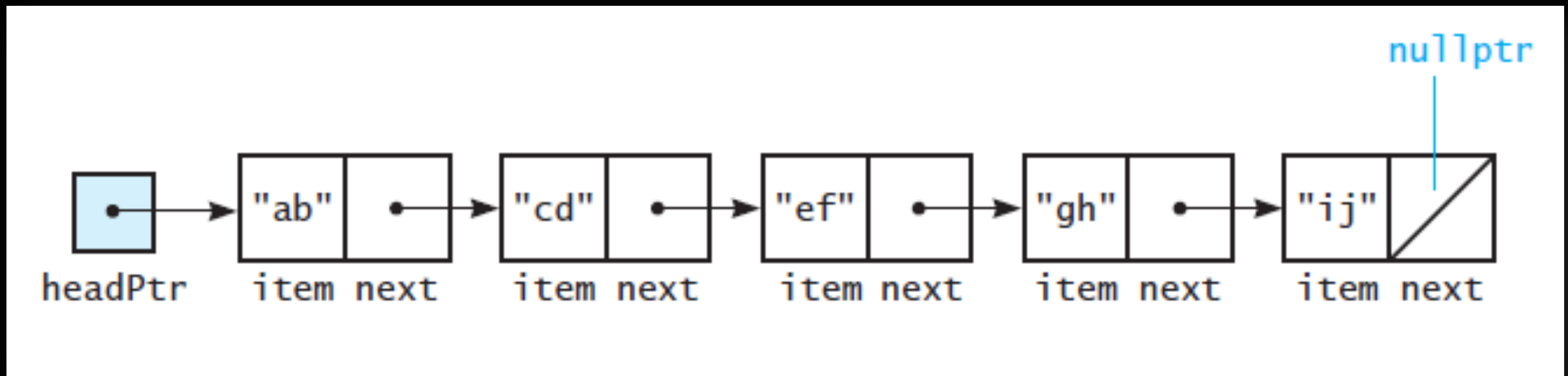
Node



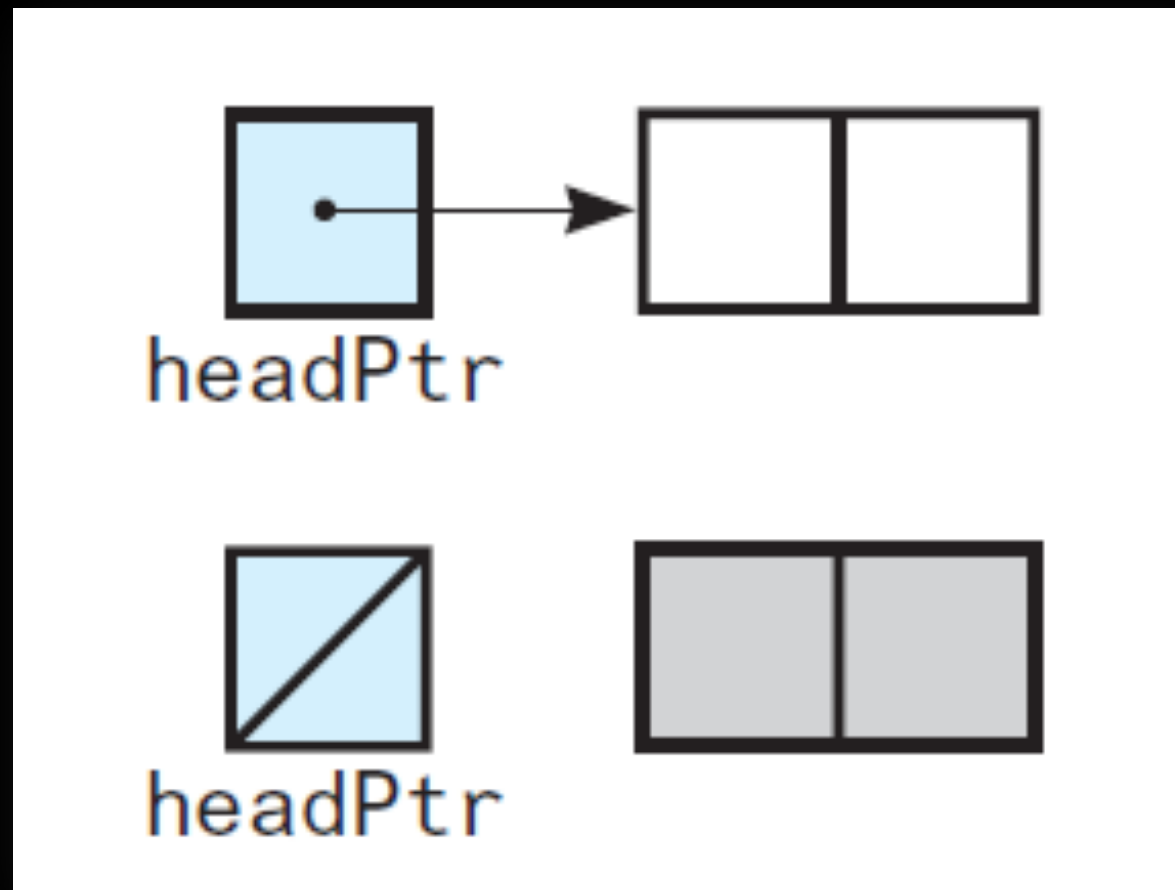
Chain



Entering the Chain



The Empty Chain



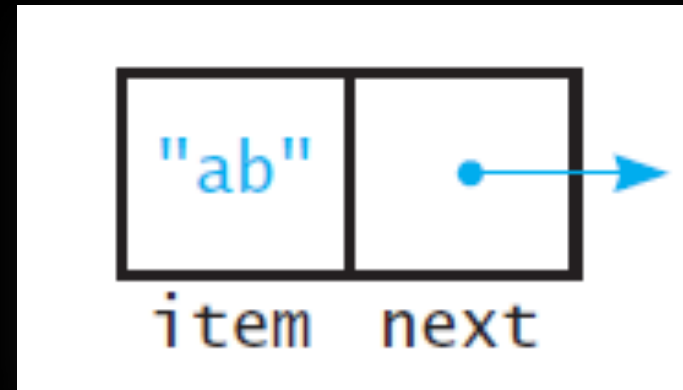
The Class Node

```
#ifndef NODE_H_
#define NODE_H_

template<class T>
class Node
{
public:
    Node();
    Node(const T& an_item);
    Node(const T& an_item, Node<T>* next_node_ptr);
    void setItem(const T& an_item);
    void setNext(Node<T>* next_node_ptr);
    T getItem() const;
    Node<T>* getNext() const;

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

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

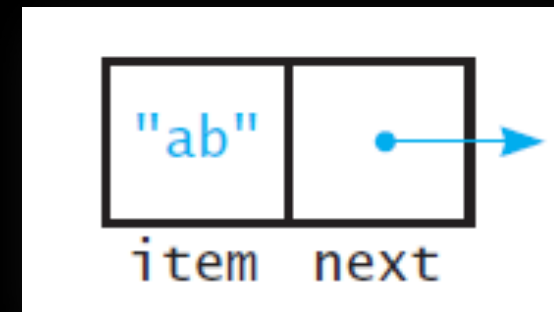


Node Implementation

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

The Constructors

```
template<class T>
Node<T>::Node() : next_(nullptr)
{
} // end default constructor
```



```
template<class T>
Node<T>::Node(const T& an_item) : item_(an_item), next_(nullptr)
{
} // end constructor
```

```
template<class T>
Node<T>::Node(const T& an_item, Node<T>* next_node_ptr) :
    item_(an_item), next_(next_node_ptr)
{
} // end constructor
```

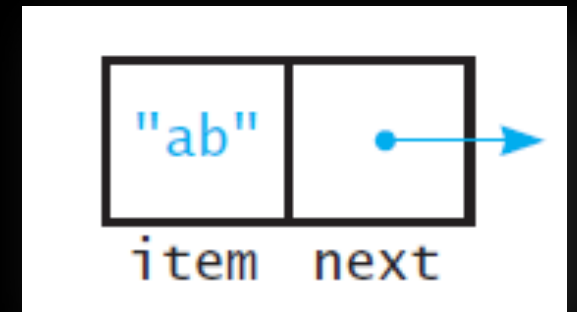
Node Implementation

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

The “setData” members

```
template<class T>
void Node<T>::setItem(const T& an_item)
{
    item_ = an_item;
} // end setItem
```

```
template<class T>
void Node<T>::setNext(Node<T>* next_node_ptr)
{
    next_ = next_node_ptr;
} // end setNext
```



Node Implementation

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

```
template<class T>  
T Node<T>::getItem() const  
{
```

```
    return item_;
```

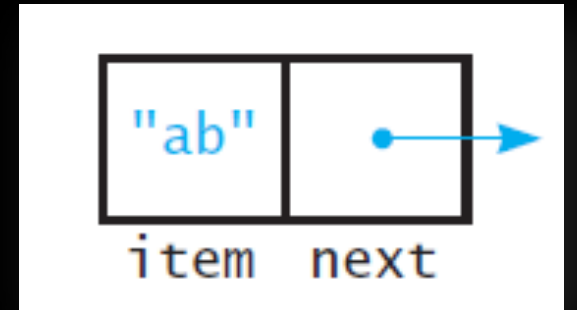
```
} // end getItem
```

```
template<class T>  
Node<T>* Node<T>::getNext() const  
{
```

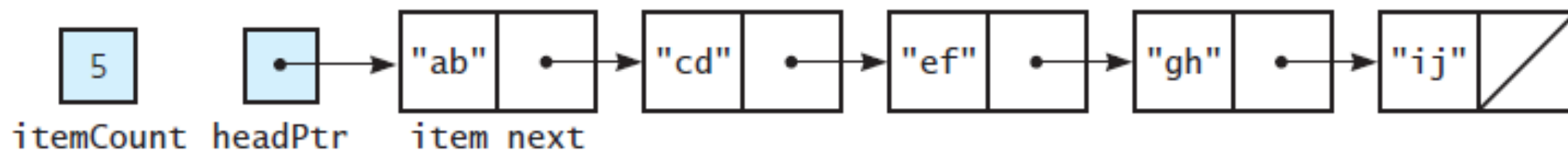
```
    return next_;
```

```
} // end getNext
```

The “get*Data*” members



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.hpp"
#include "Node.hpp"

template<class T>
class LinkedBag
{
public:
    LinkedBag();
    LinkedBag(const LinkedBag<T>& a_bag); // Copy 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:
    ???

}; // end LinkedBag

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

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(); // 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_
```

More than one public method will need to know if there is 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.hpp"
```

The default constructor

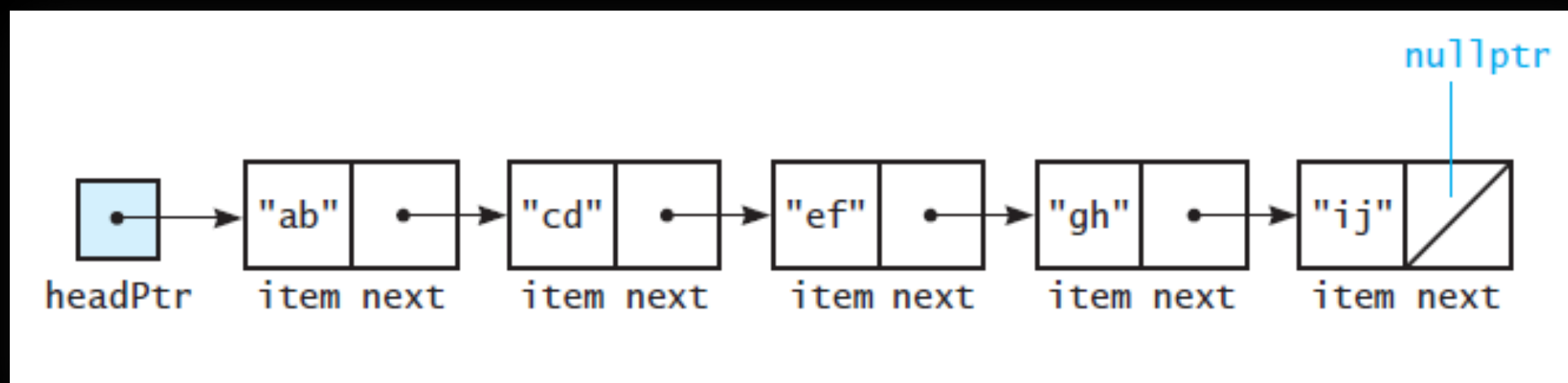
```
template<class T>
LinkedBag<T>::LinkedBag() : head_ptr_(nullptr),
item_count_(0)
{

} // end default constructor
```

Private data member
initialization

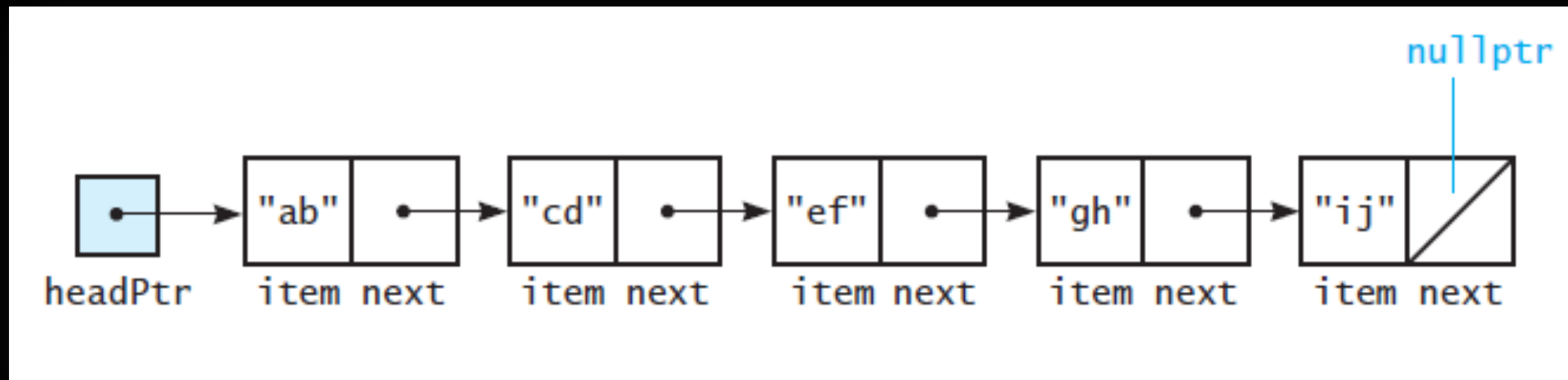

```
add(const T& new_entry)
```

Where should we add?

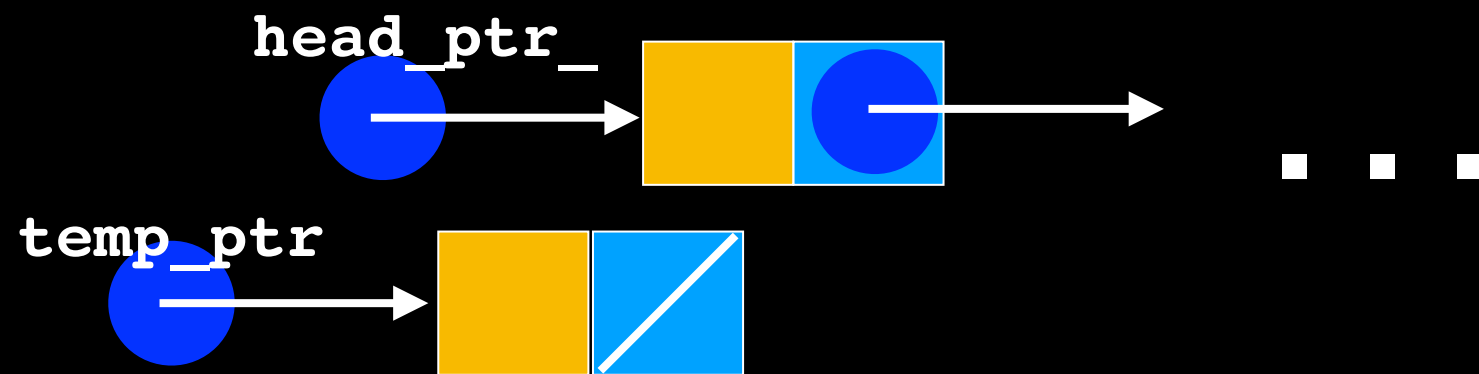


Lecture Activity

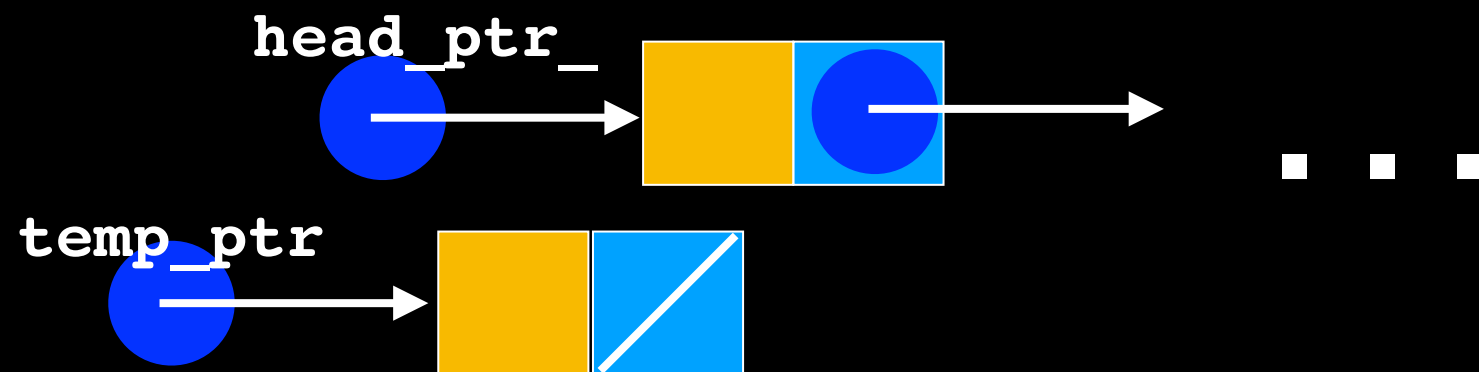
Write **pseudocode** for a sequence of steps to add to the **front** of the chain



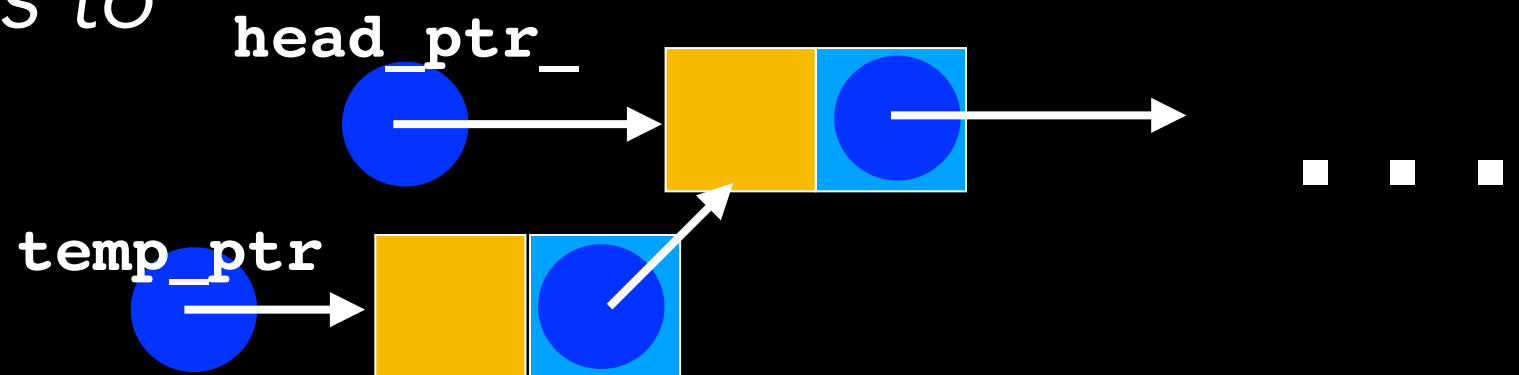
*Instantiate a **new** node and let a **temp pointer** point to it*



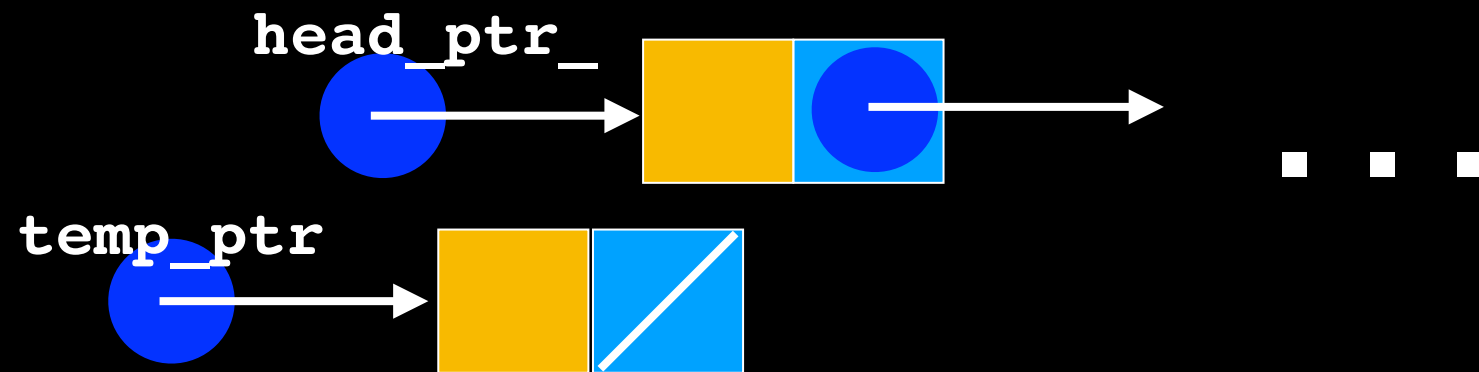
Instantiate a *new* node and let a *temp pointer* point to it



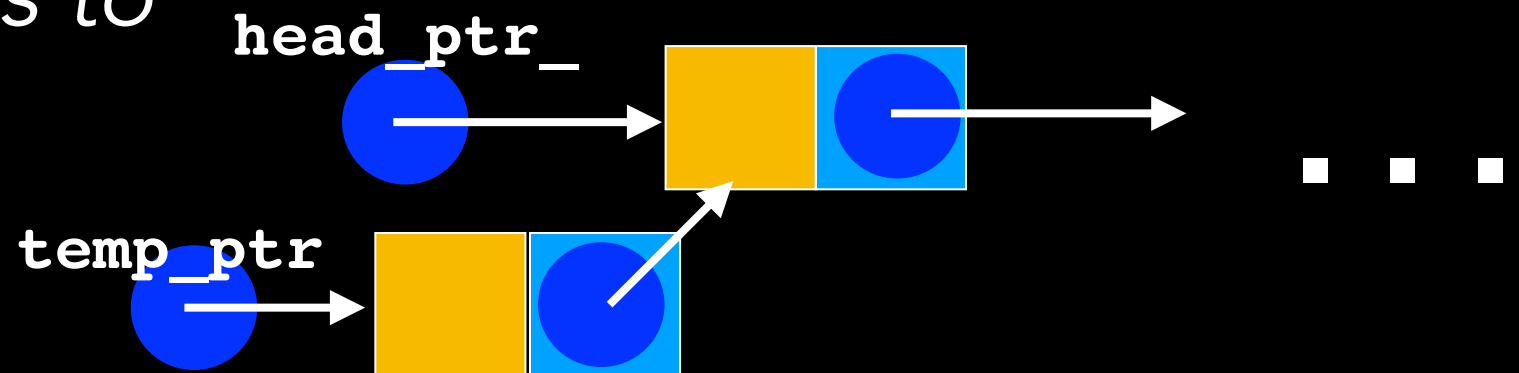
Let the *next pointer* of the *new node* point to the same node *head_ptr_* points to



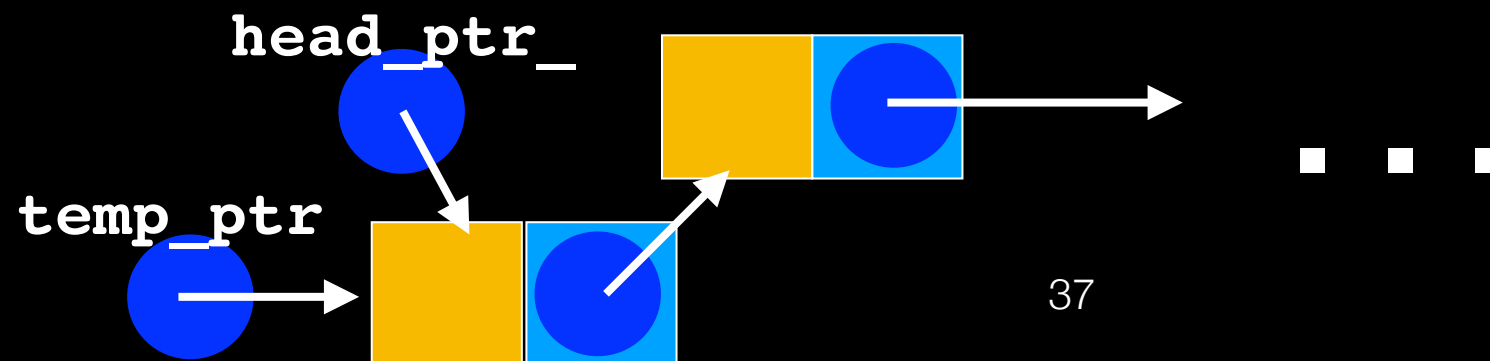
Instantiate a *new* node and let a *temp pointer* point to it



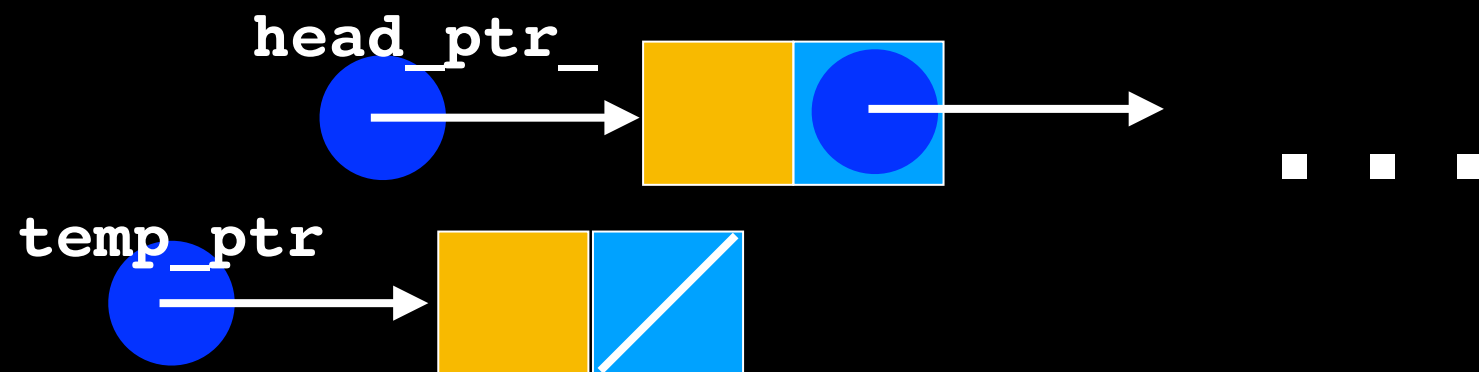
Let the *next pointer* of the *new node* point to the same node *head_ptr_* points to



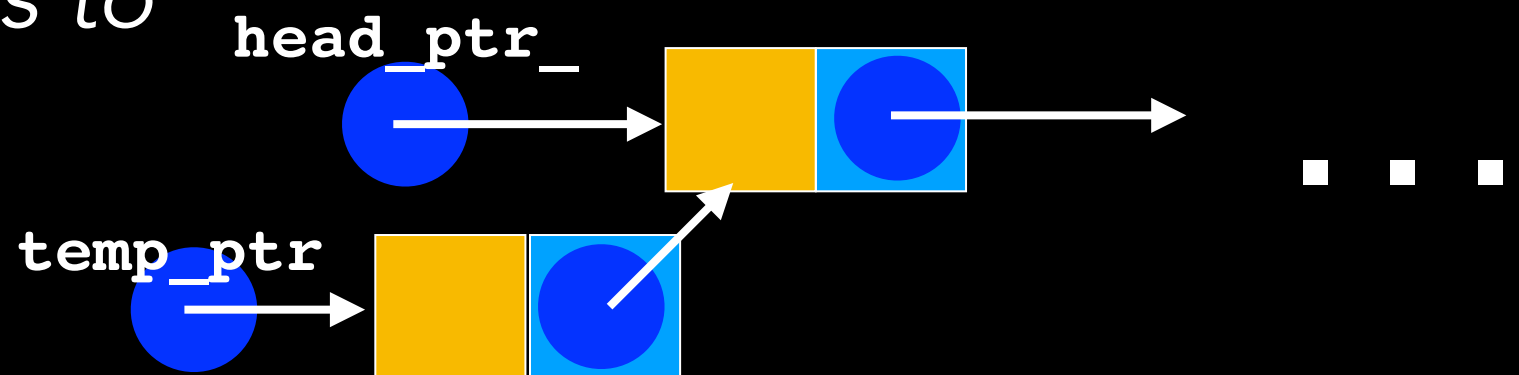
Let *head_ptr_* point to the *new node*



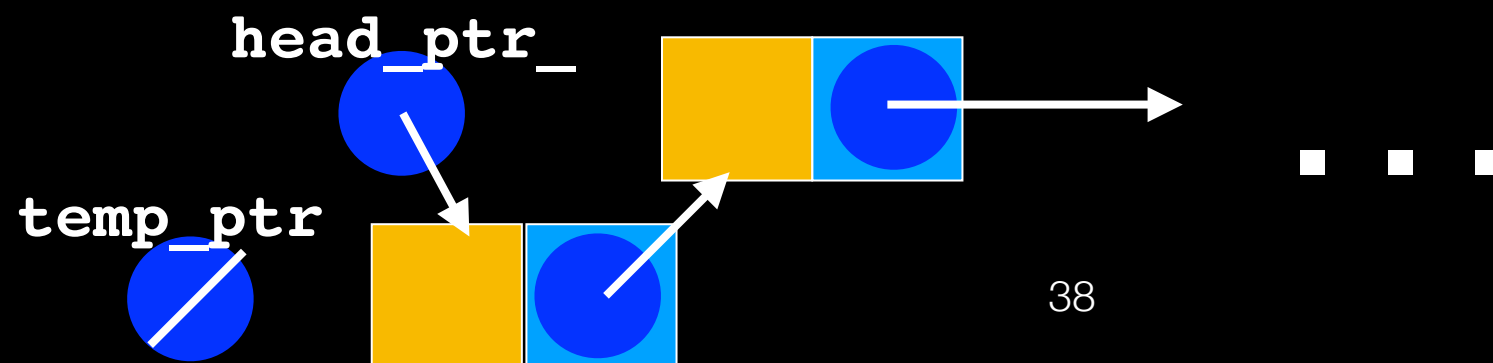
Instantiate a *new* node and let a *temp pointer* point to it



Let the *next pointer* of the *new node* point to the same node *head_ptr_* points to



Let *head_ptr_* point to the *new node*



Pseudocode (English-like)

- Instantiate a new node and let `temp_ptr` point to it
- Set `temp_ptr->next` to point to the same node
`head_ptr_` points to
- Set `head_ptr` to point to the same node
`temp_ptr` points to
- Set `temp_ptr` to `nullptr`

Pseudocode (Code-like)

```
temp_ptr = new node
temp_ptr->next = head_ptr_
head_ptr = temp_ptr
temp_ptr = nullptr
```


LinkedList Implementation

```
#include "LinkedList.hpp"
```

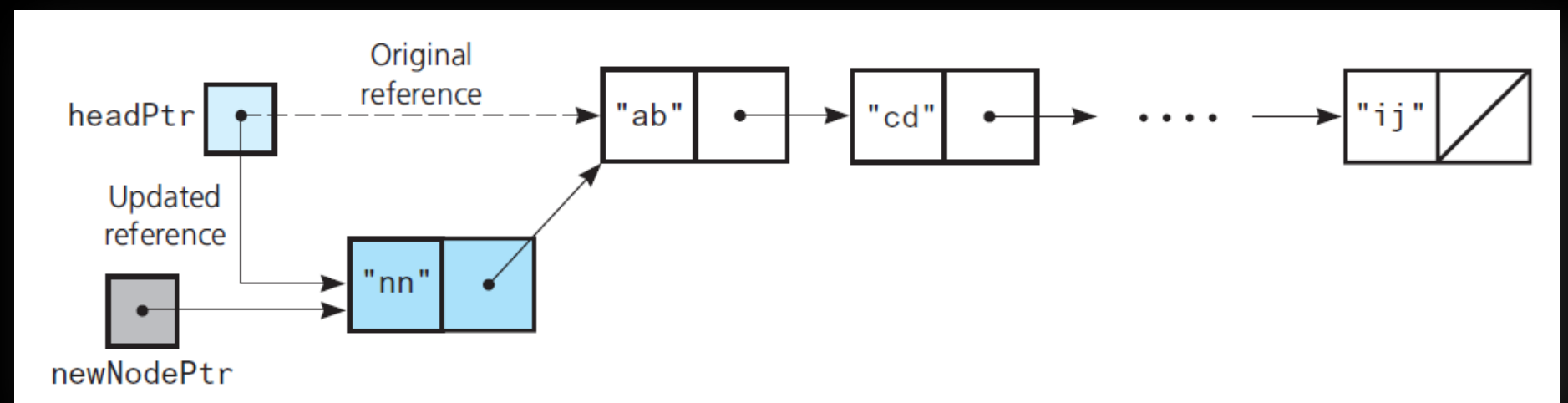
```
template<class T>
bool LinkedList<T>::add(const T& new_entry)
{
    // Add to beginning of chain: new node references rest of chain;
    // (head_ptr_ is null if chain is empty)
    Node<T>* new_node_ptr = new Node<T>();
    new_node_ptr->setItem(new_entry);
    new_node_ptr->setNext(head_ptr_); // New node points to chain

    head_ptr_ = new_node_ptr; // New node is now first node
    item_count_++;

    return true;
} // end add
```

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

Dynamic memory
allocation
Adding nodes to the heap!

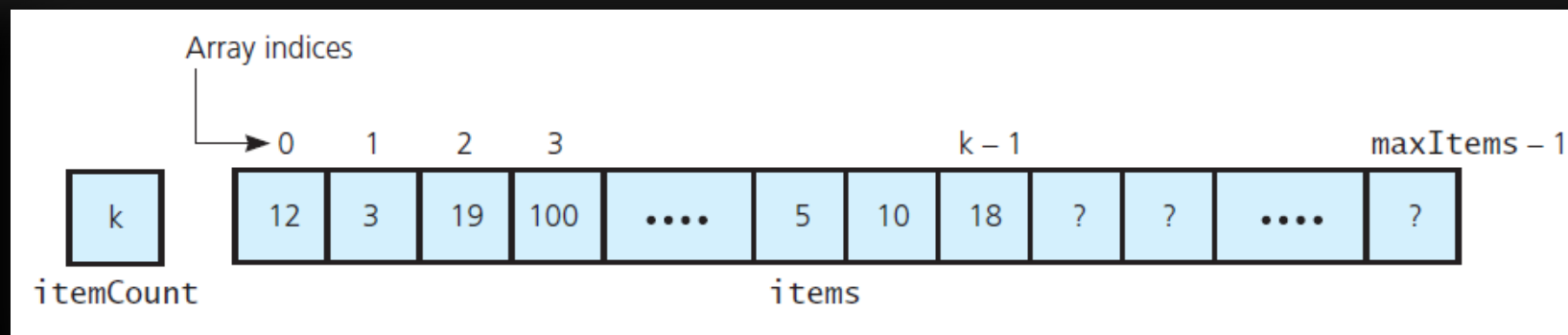
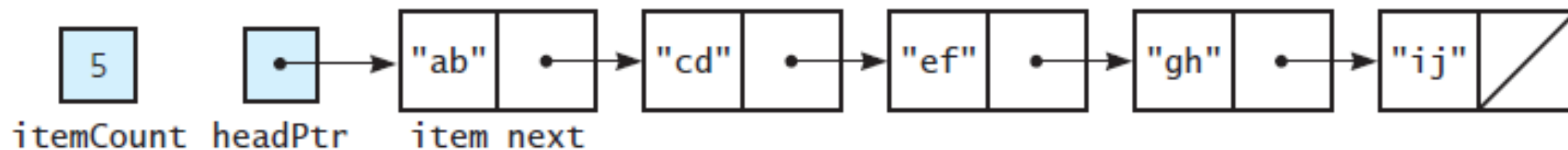


Efficiency

Create a new node and assign two pointers

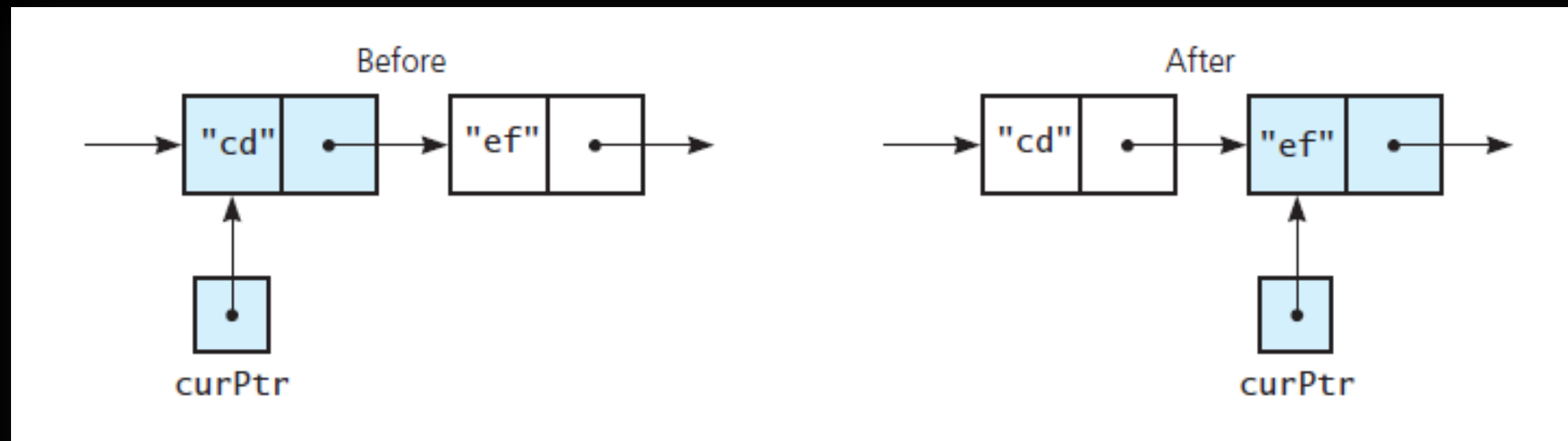
What about adding to end of chain?

What about adding to front of array?



Lecture Activity

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

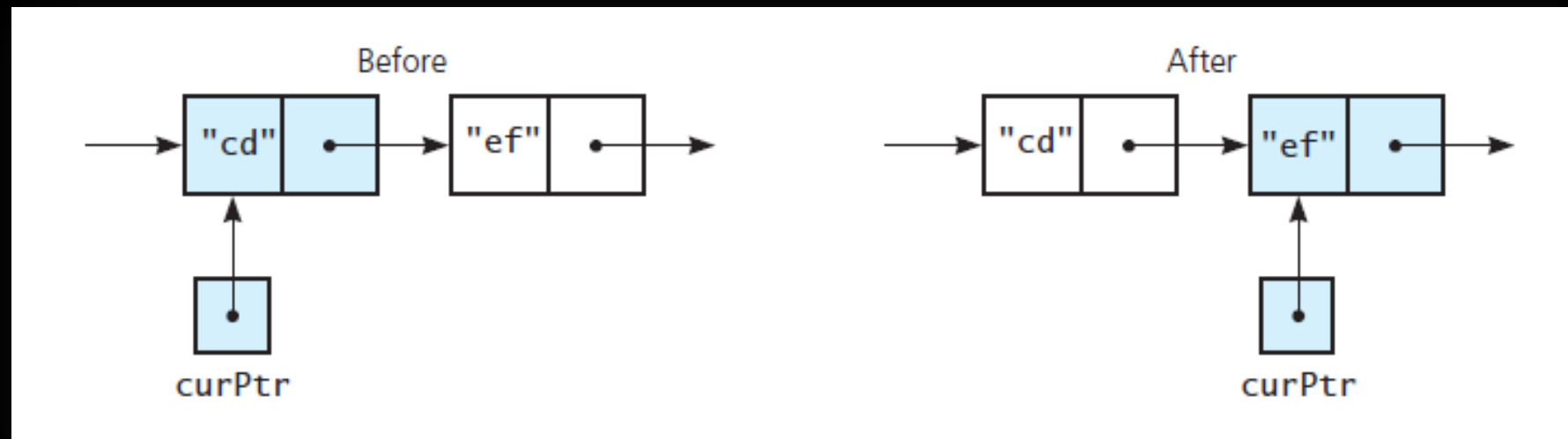
while(the *current pointer* is not the *null pointer*)

{

"visit" the current node

set the *current pointer* to the *next* pointer of the current node

}



LinkedBag Implementation

```
#include "LinkedBag.hpp"
```

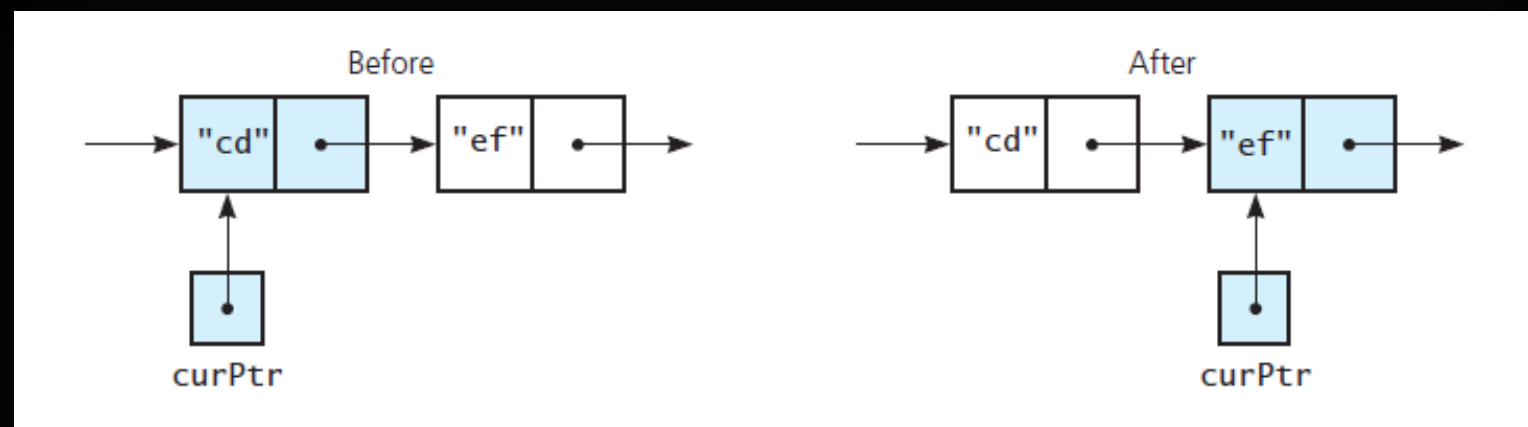
The toVector method

```
template<class T>
std::vector<T> LinkedBag<T>::toVector() const
{
    std::vector<T> bag_contents;
    Node<T>* cur_ptr = head_ptr_;

    while ((cur_ptr != nullptr))
    {
        bag_contents.push_back(cur_ptr->getItem());
        cur_ptr = cur_ptr->getNext();
    } // end while

    return bag_contents;
} // end toVector
```

Traversing:
Visit each node
Copy it



LinkedBag Implementation

Similarly `getFrequencyOf` will:
 count frequency of (count each) `an_entry`

LinkedList Implementation

```
#include "LinkedList.hpp"
```

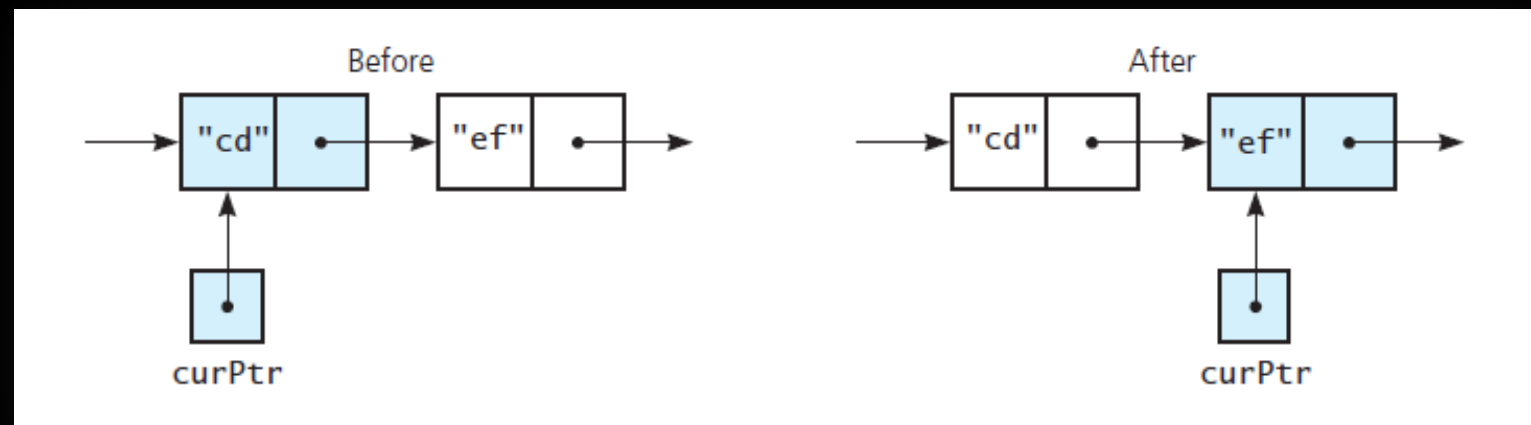
```
template<class T>
Node<T>* LinkedList<T>::getPointerTo(const T& an_entry) const
{
    bool found = false;
    Node<T>* cur_ptr = head_ptr_;

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

    return cur_ptr;
} // end getPointerTo
```

The getPointerTo
method

Traversing:
visit each node
if found what looking for
return

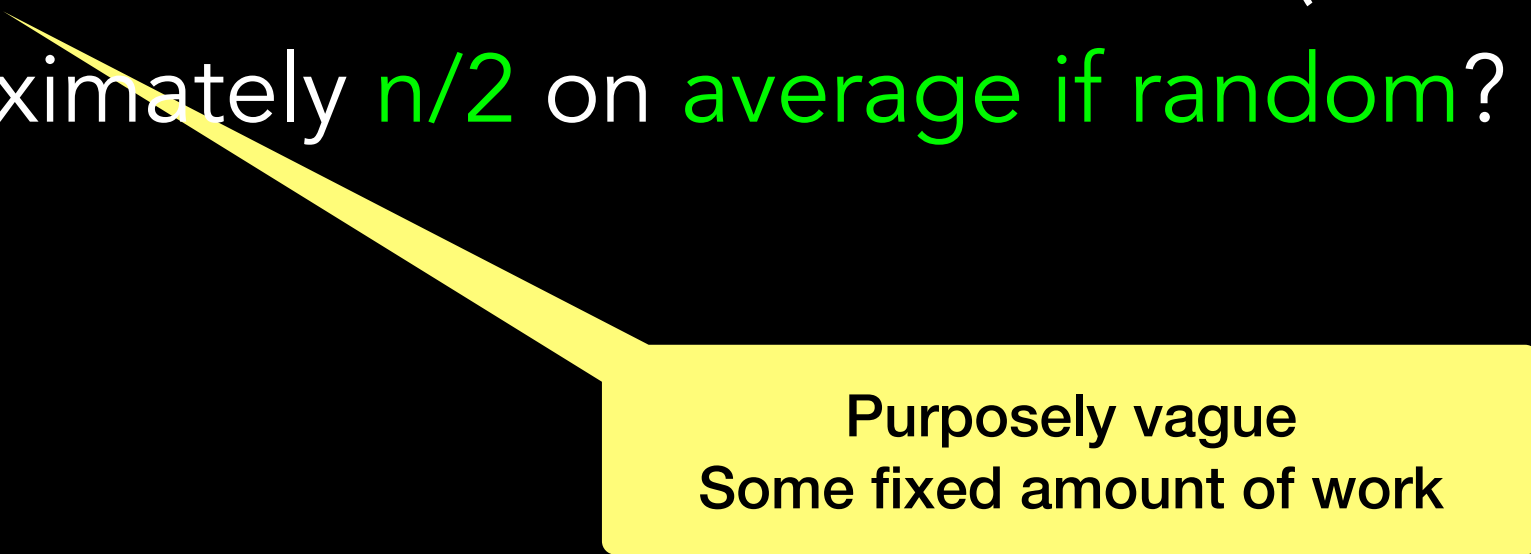


Efficiency

No fixed number of steps

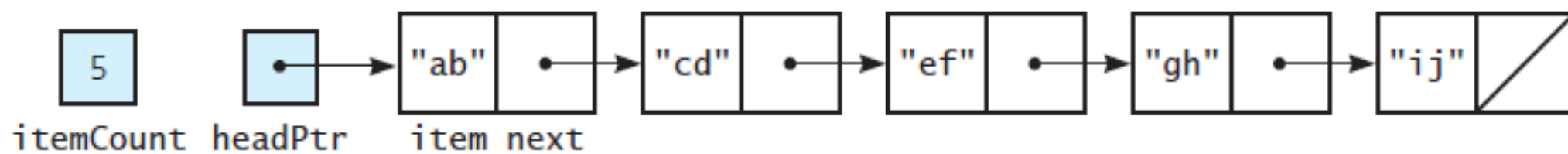
Depends on location of `an_entry`

- 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 if random?



Purposely vague
Some fixed amount of work

What should we do to remove?



LinkedList Implementation

```
#include "LinkedList.hpp"
```

The remove method

```
template<class T>
bool LinkedList<T>::remove(const T& an_entry)
{
    Node<T>* entry_ptr = getPointerTo(an_entry);
    bool can_remove = !isEmpty() && (entry_ptr != nullptr);
    if (can_remove)
    {
        // Copy data from first node to located node
        entry_ptr->setItem(head_ptr->getItem());
        // Delete first node
        Node<T>* node_to_delete_ptr = head_ptr;
        head_ptr = head_ptr->getNext();
        // Return node to the system
        node_to_delete_ptr->setNext(nullptr);
        delete node_to_delete_ptr;
        node_to_delete_ptr = nullptr;
        item_count--;
    } // end if

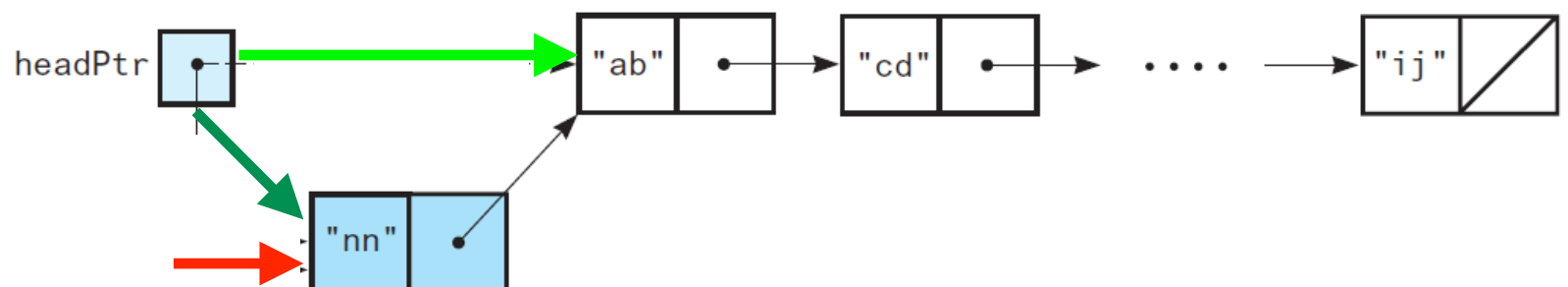
    return can_remove;
} // end remove
```

Find

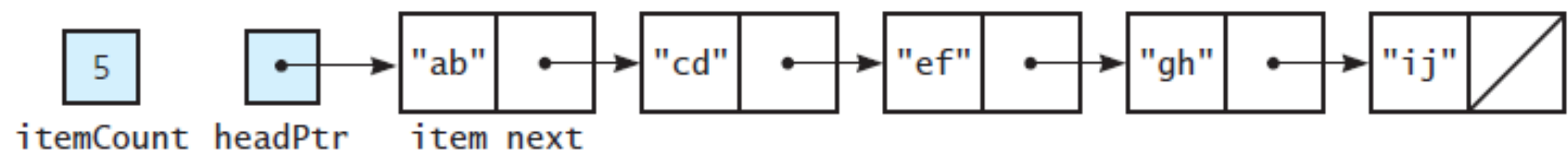
Deleting first node is easy

Copy data from first node
to node to delete
Delete first node

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



How do we clear the bag?



LinkedBag Implementation

```
#include "LinkedBag.hpp"
```

```
template<class T>
void LinkedBag<T>::clear()
{
    Node<T>* node_to_delete_ptr = head_ptr_;
    while (head_ptr_ != nullptr)
    {
        head_ptr_ = head_ptr_->getNext();

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

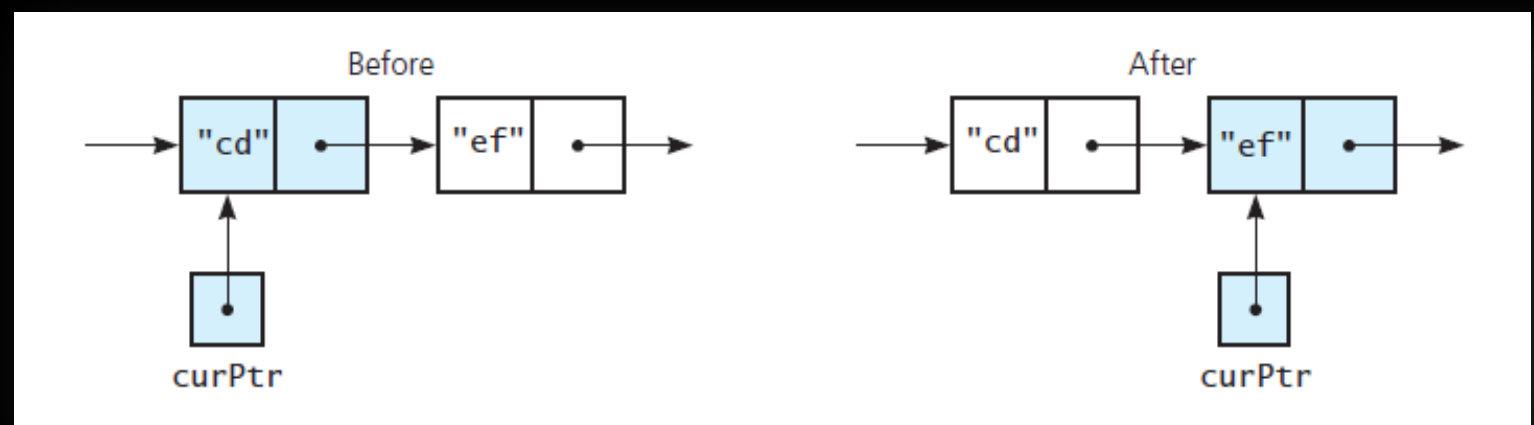
        node_to_delete_ptr = head_ptr_;
    } // end while
    // head_ptr_ is nullptr; node_to_delete_ptr is nullptr

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

The clear method

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

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



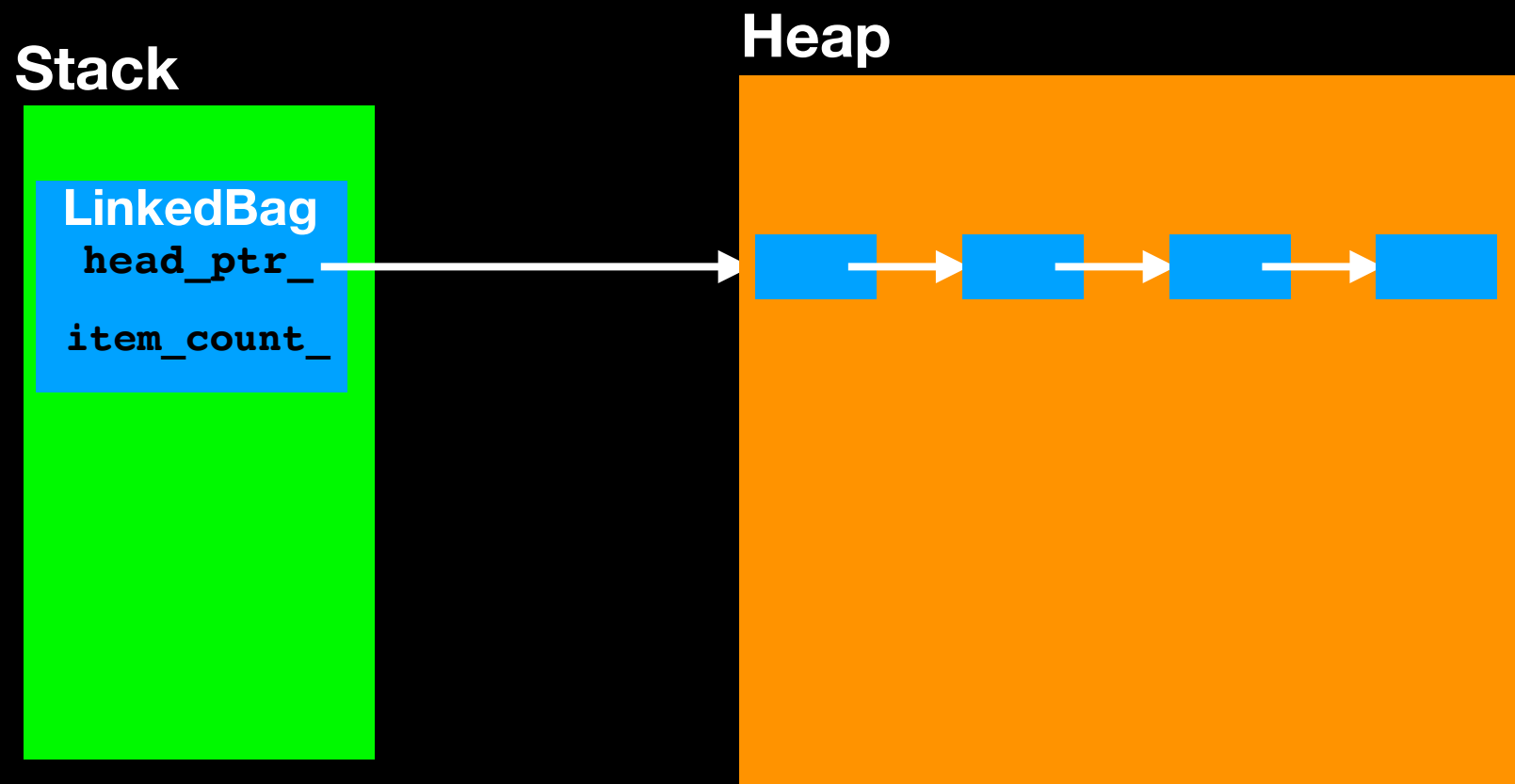
Dynamic Memory Considerations

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!

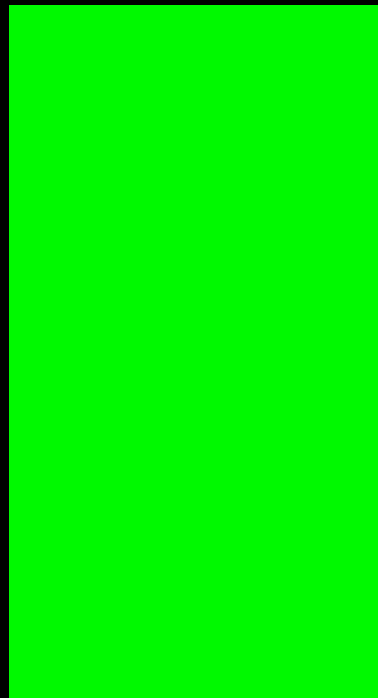
Avoid memory leaks!!!!

What happens when object goes out of scope?

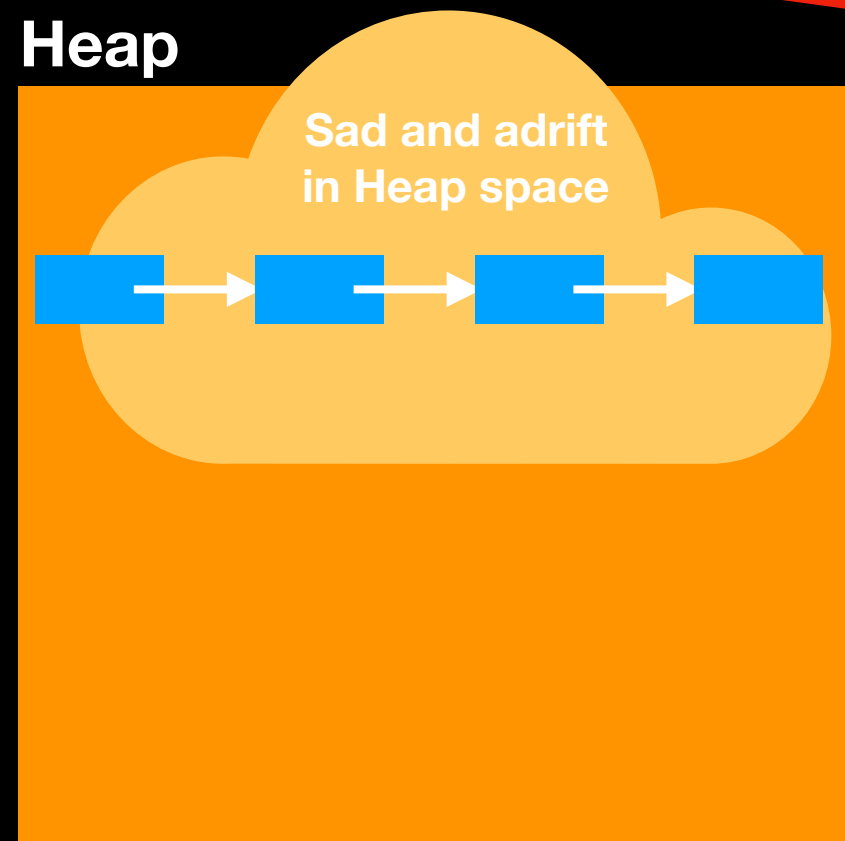


What happens when object goes out of scope?

Stack



Heap



Memory
leak!!!

LinkedBag Implementation

```
#include "LinkedBag.hpp"
```

The destructor

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

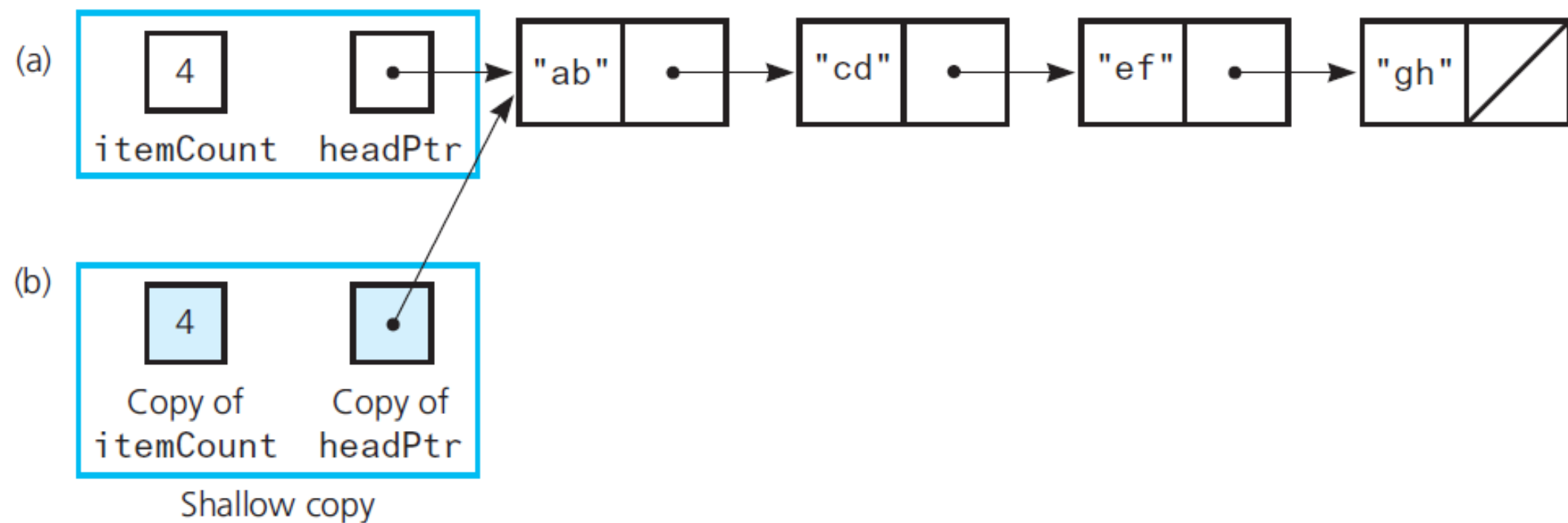
2. Copy an object to **pass by value** as an argument to a function

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

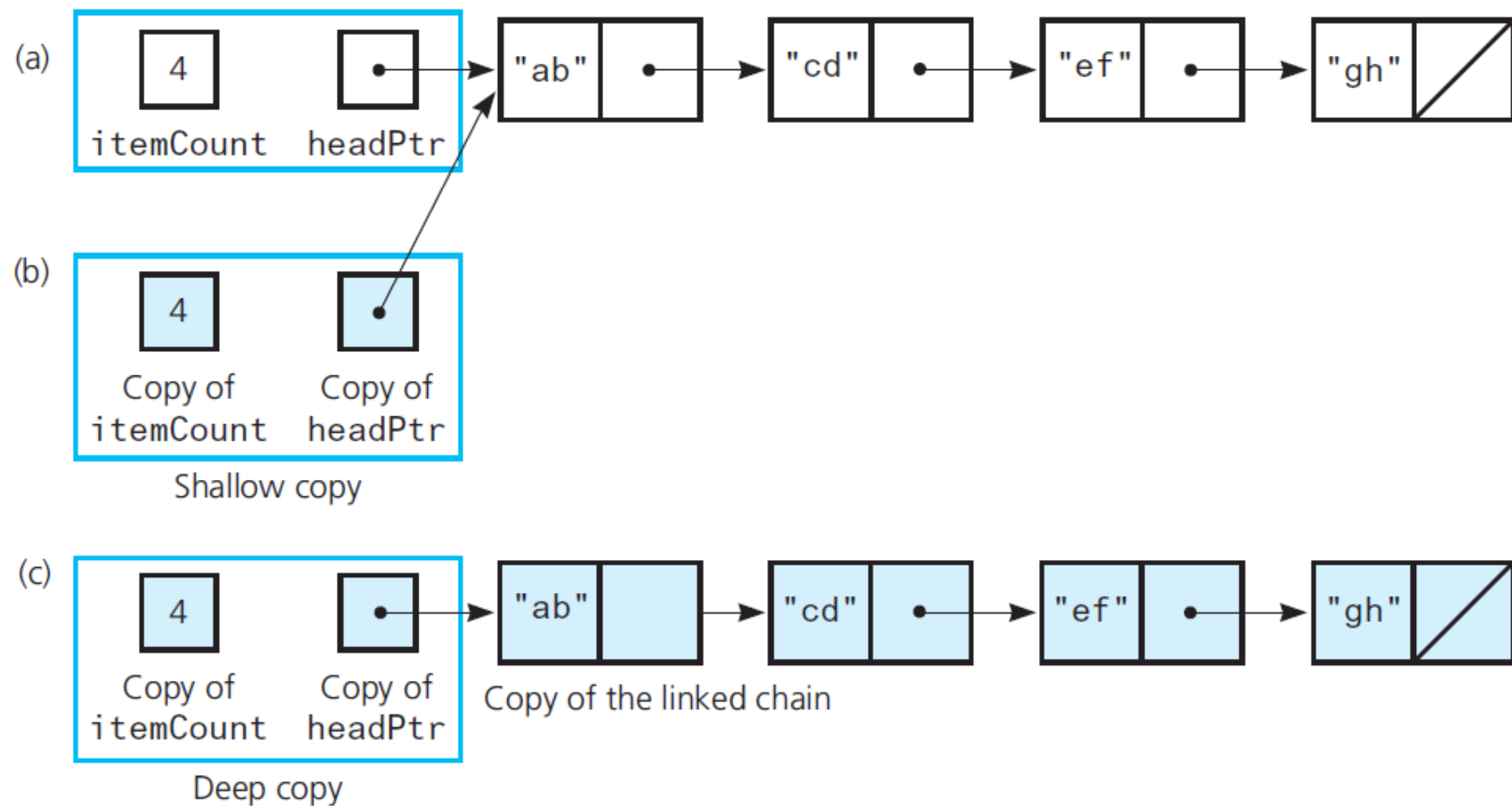
3. Copy an object to be **returned** by a function

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

Deep vs Shallow Copy



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

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

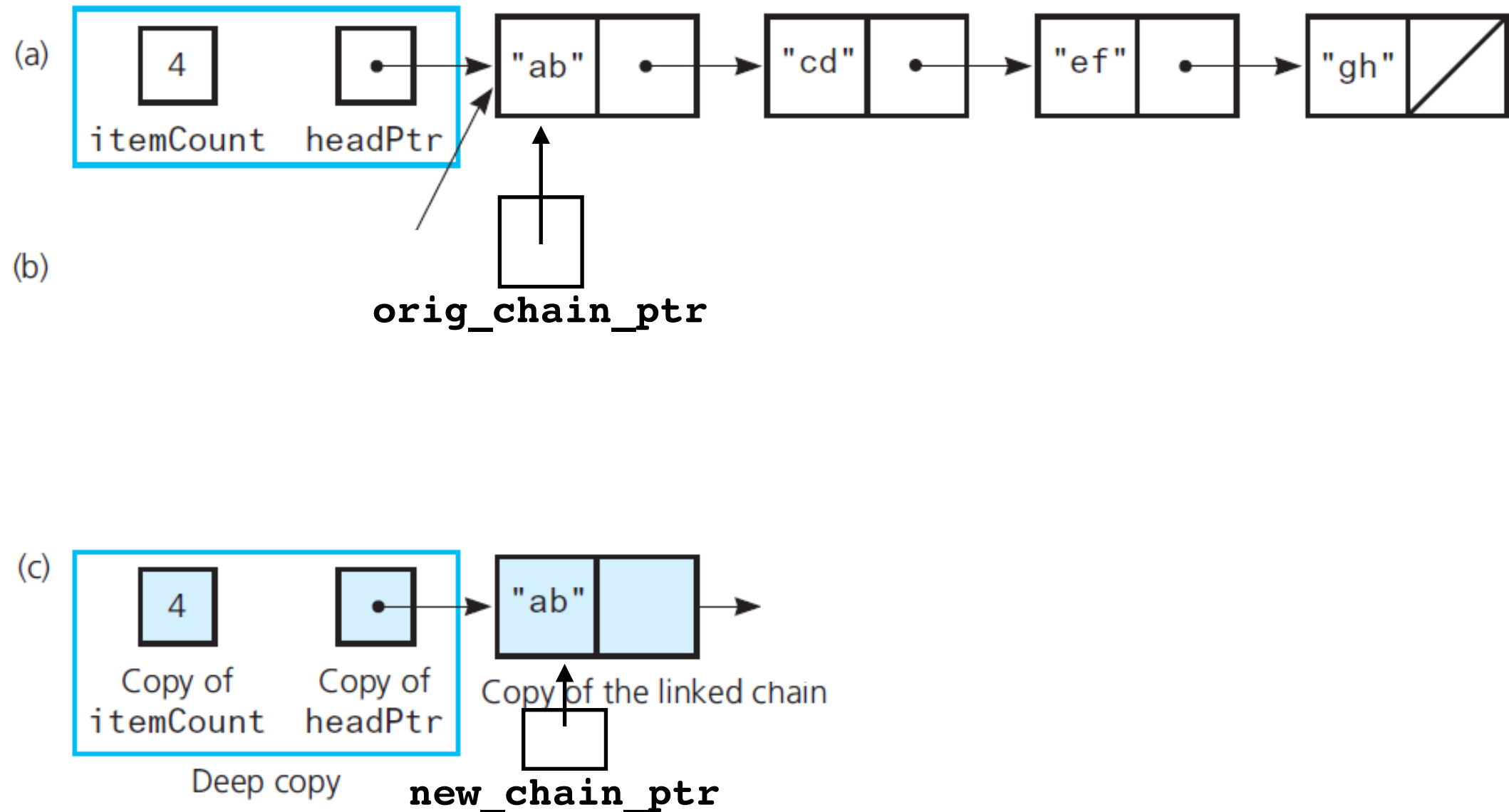
Advance pointer traversing original chain

Signal last node

Deep vs Shallow Copy

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

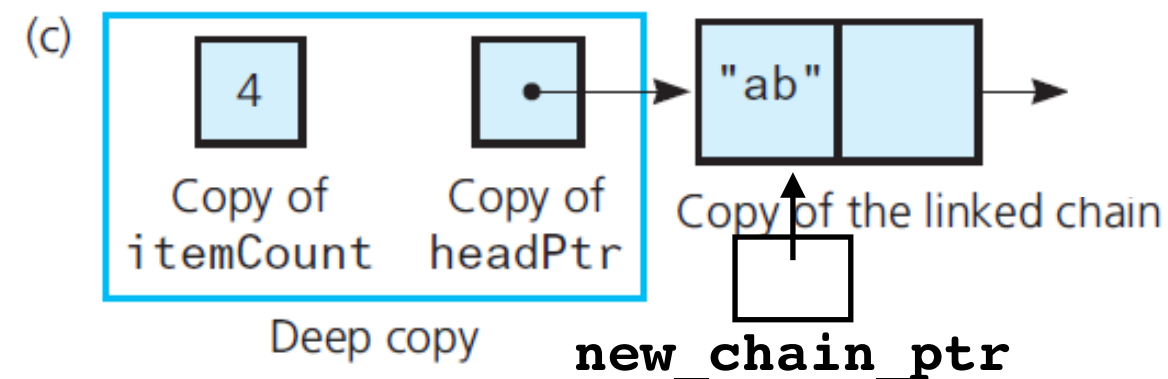
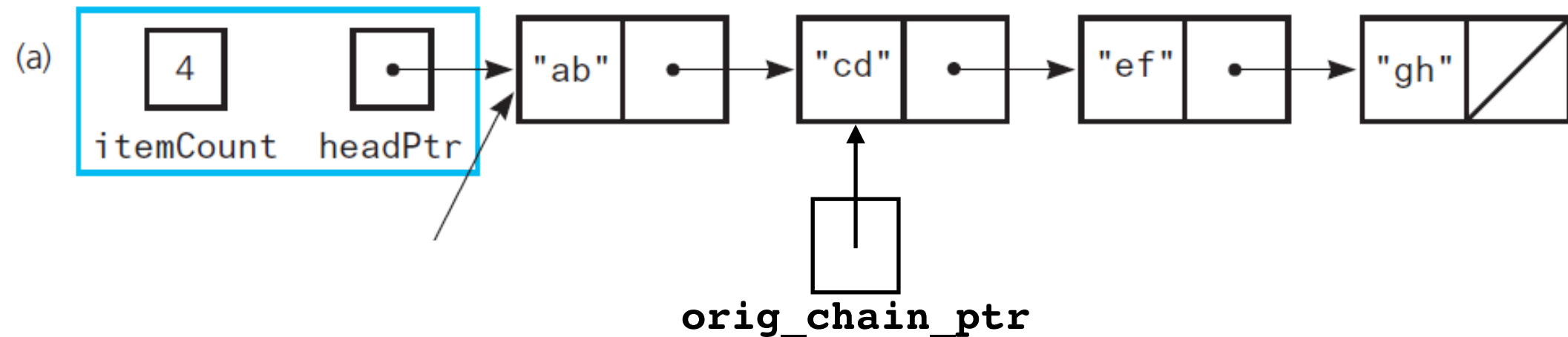
// Copy remaining nodes
Node<T>* new_chain_ptr = head_ptr_; // Points to last node in new chain
orig_chain_ptr = orig_chain_ptr->getNext();
```



Deep vs Shallow Copy

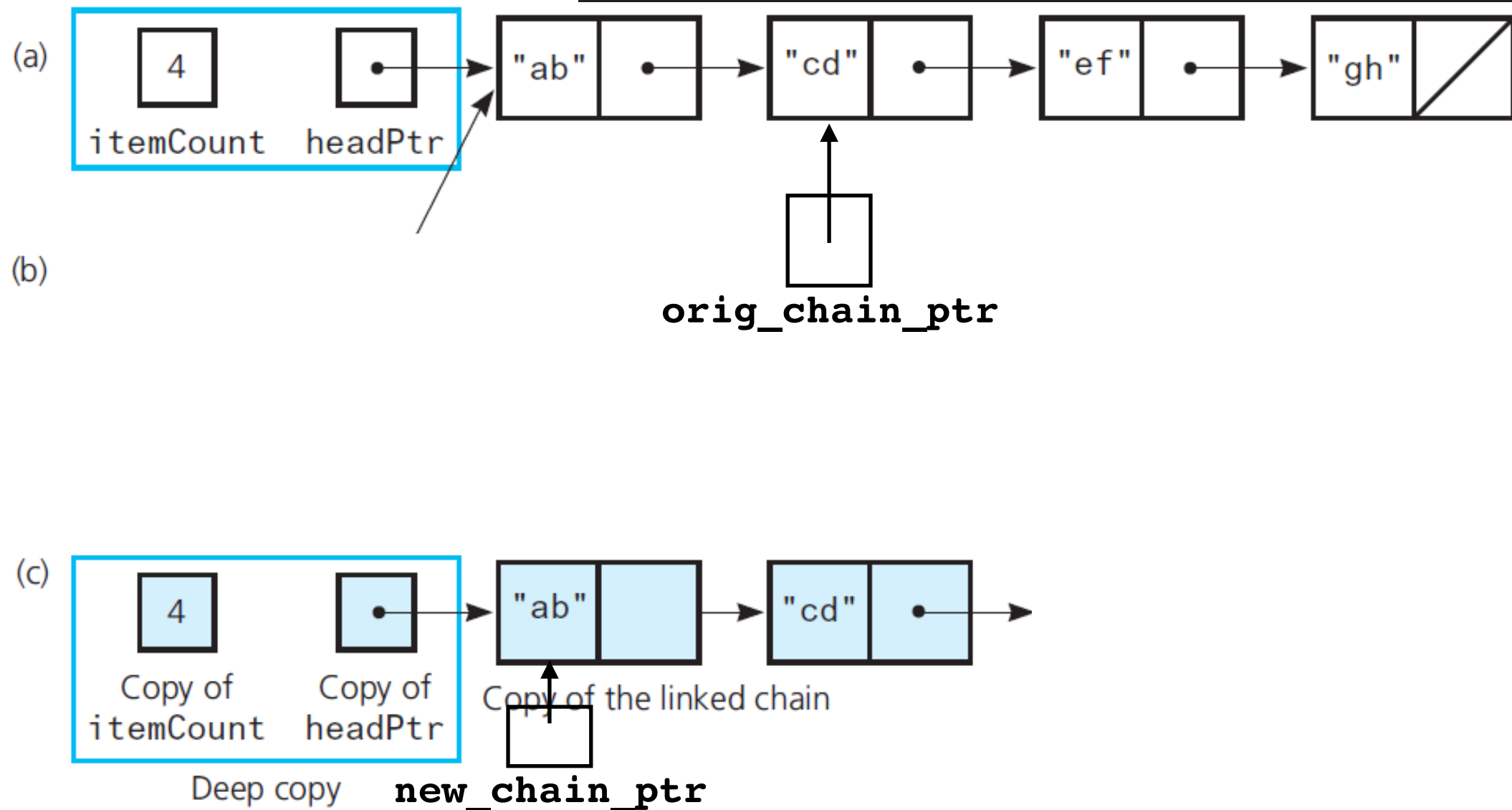
```
// Copy first node
head_ptr_ = new Node<T>();
head_ptr_->setItem(orig_chain_ptr->getItem());

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



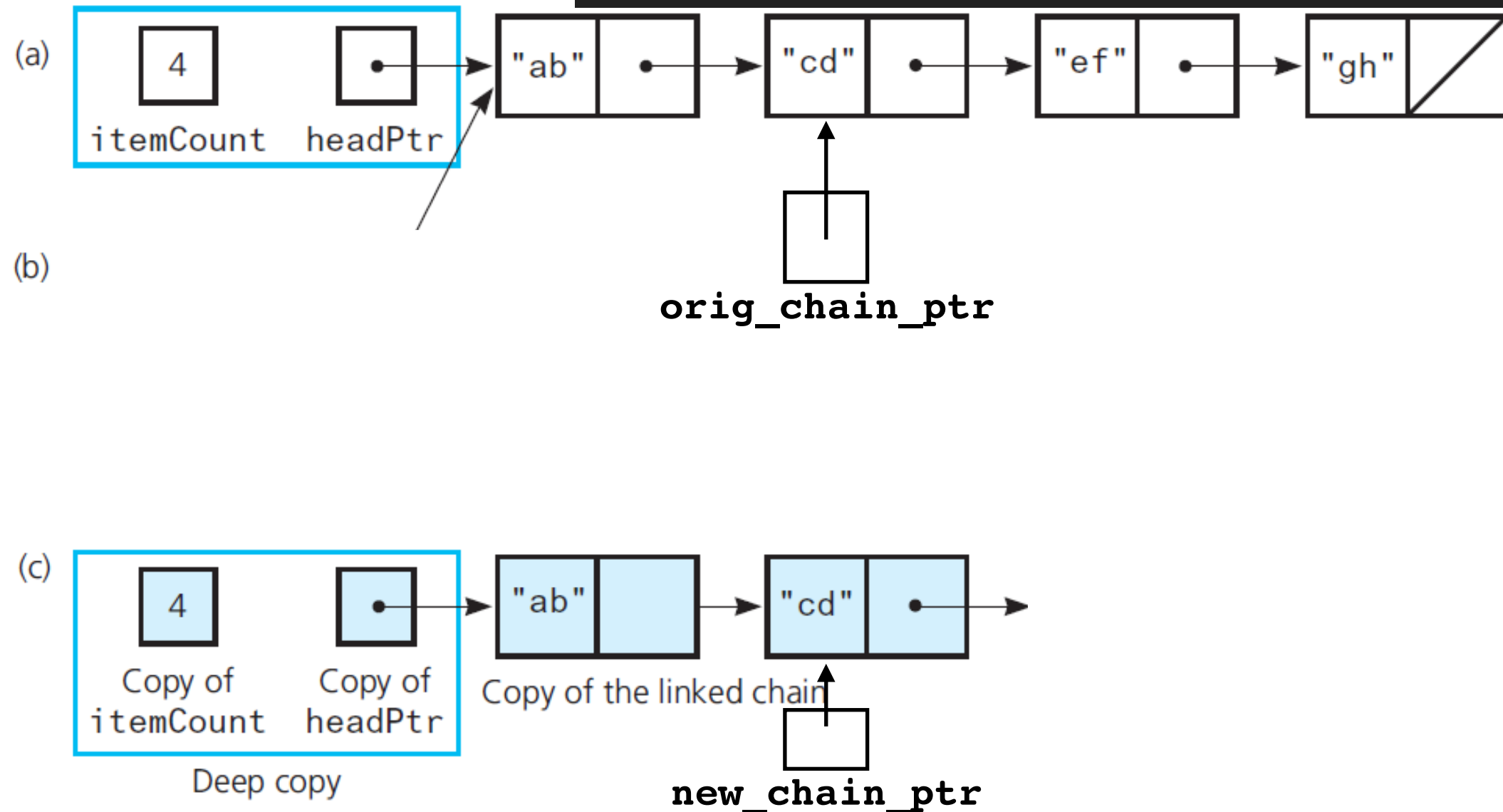
Deep vs Shallow Copy

```
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();
}
```



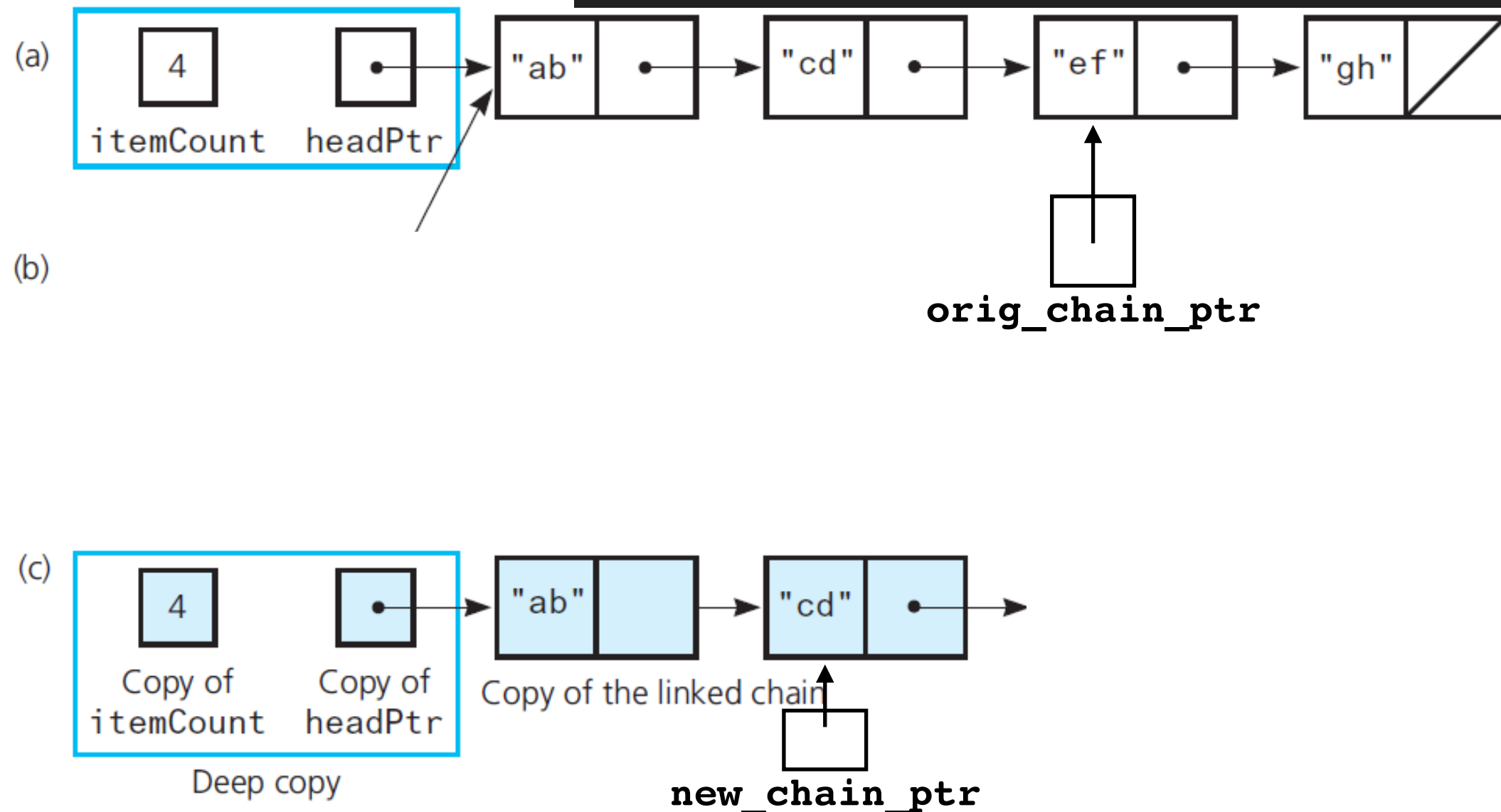
Deep vs Shallow Copy

```
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();
}
```



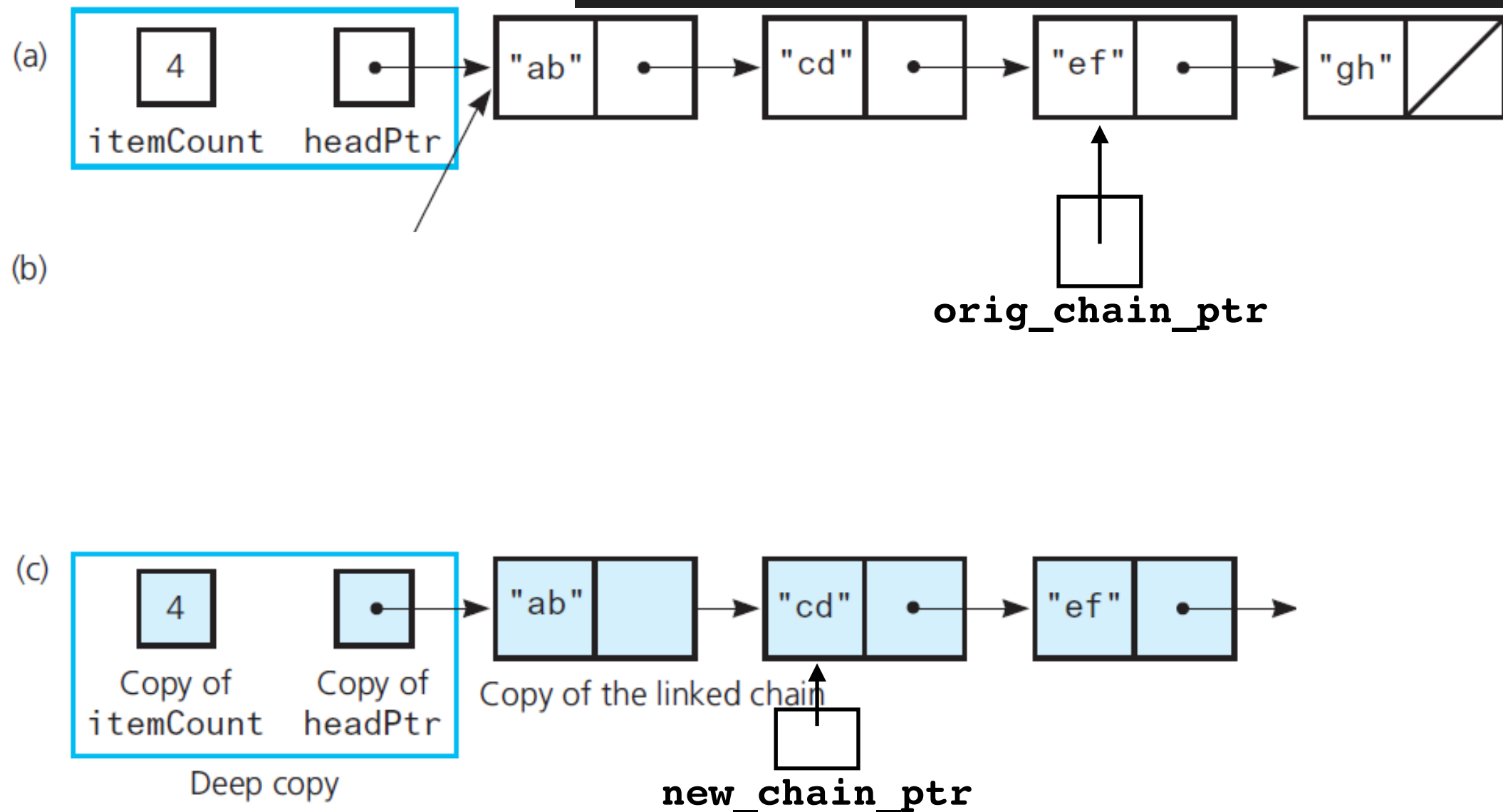
Deep vs Shallow Copy

```
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();
}
```



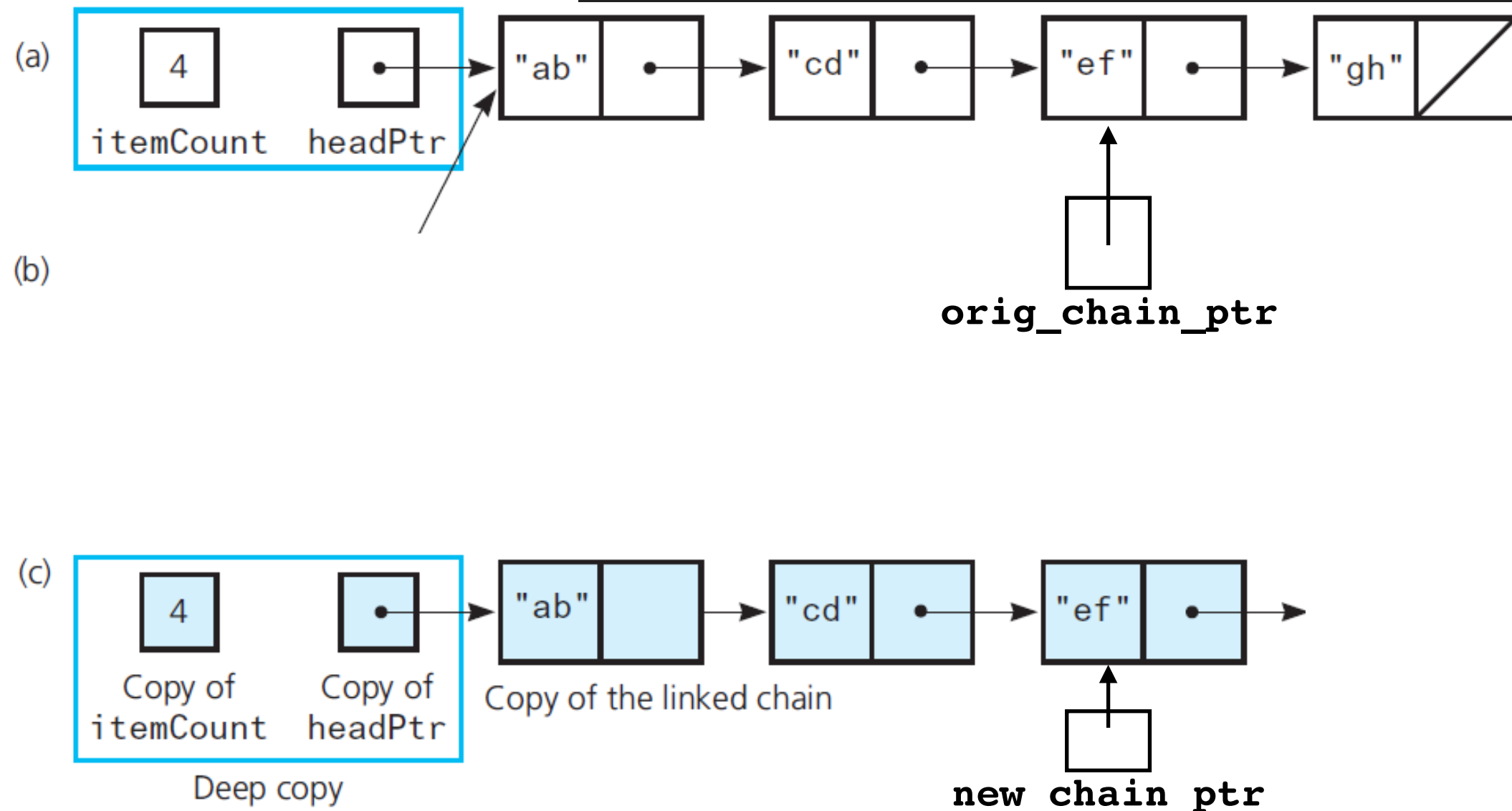
Deep vs Shallow Copy

```
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();
}
```



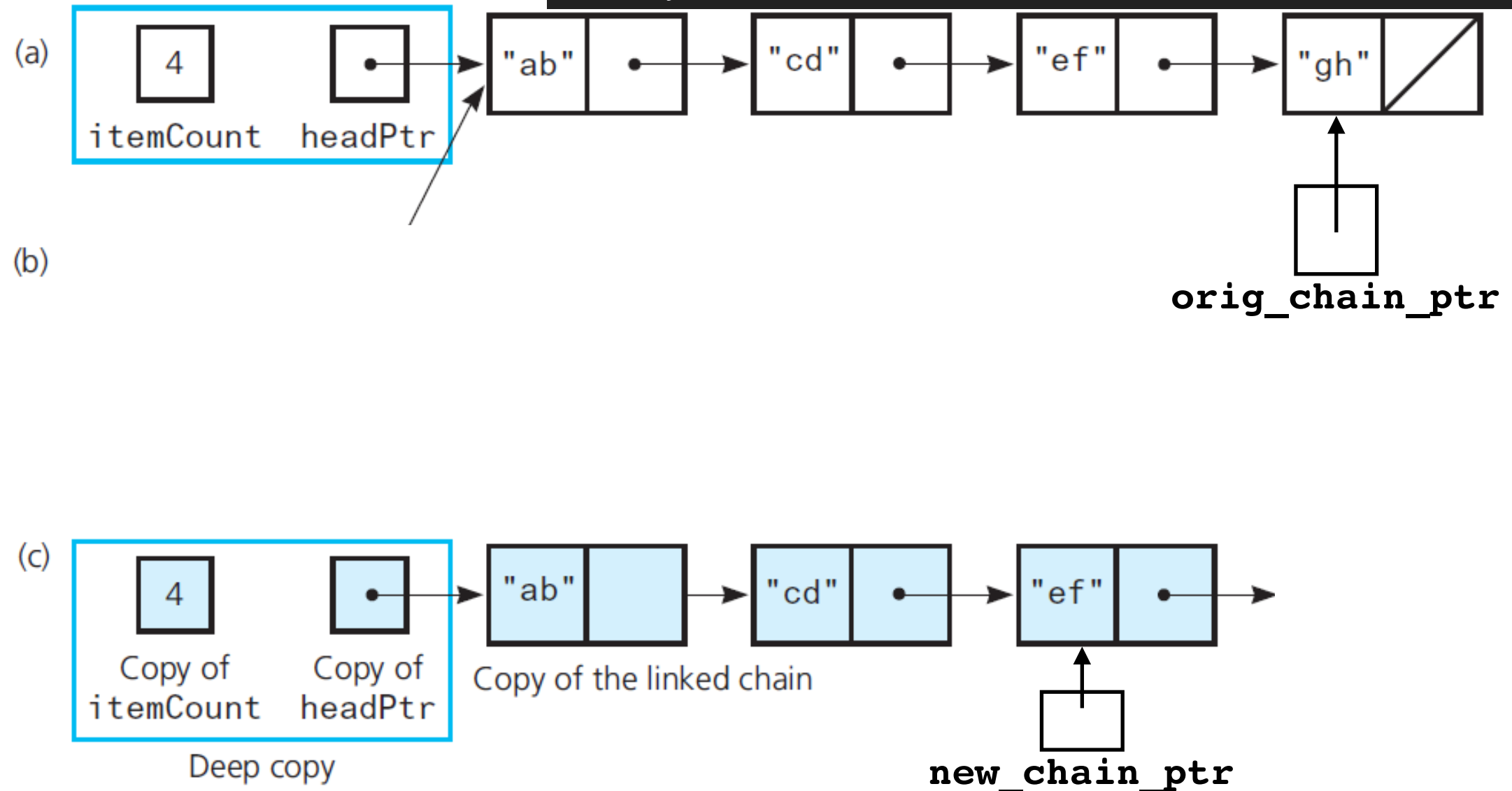
Deep vs Shallow Copy

```
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();
}
```



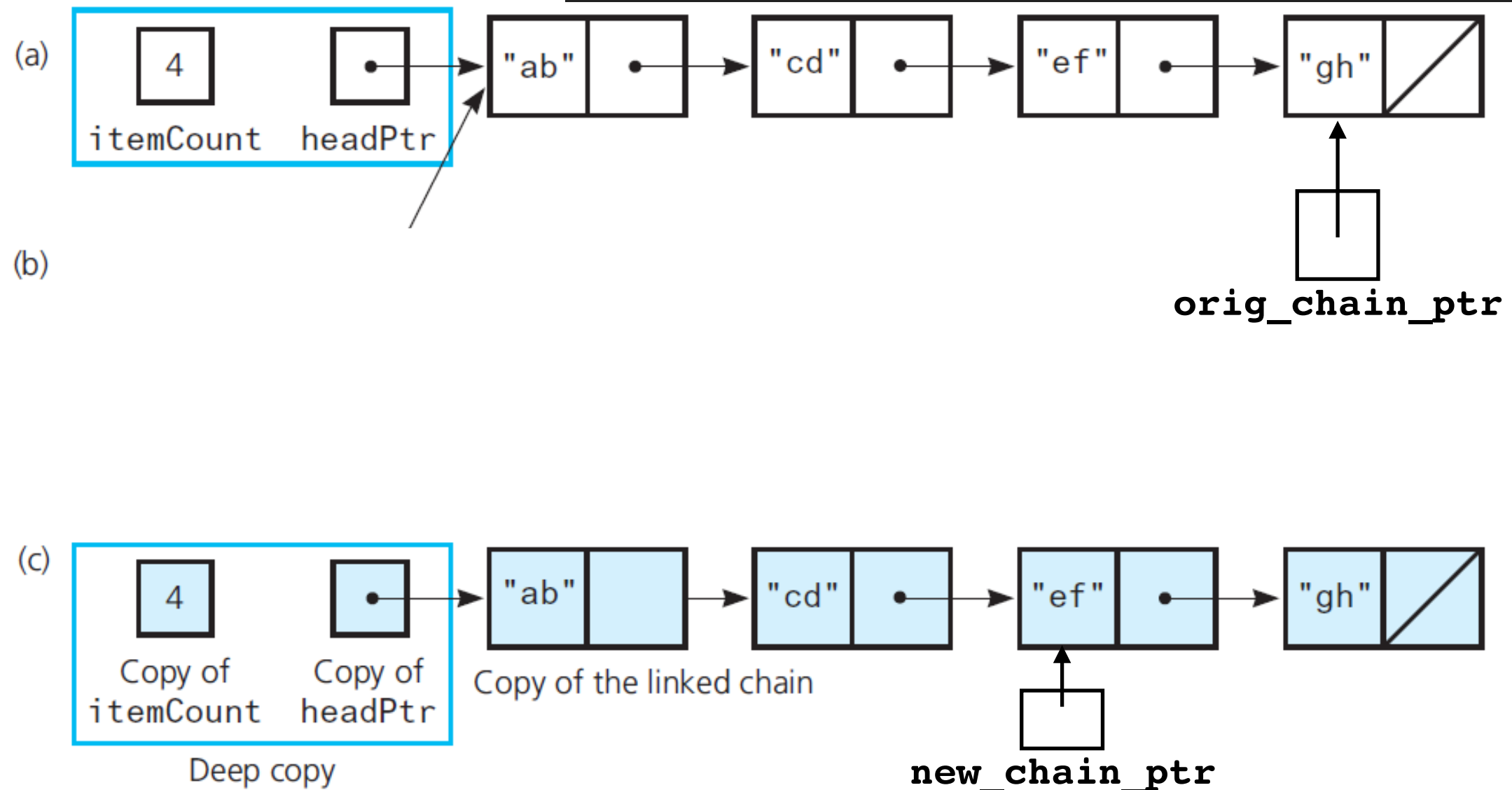
Deep vs Shallow Copy

```
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();
}
```



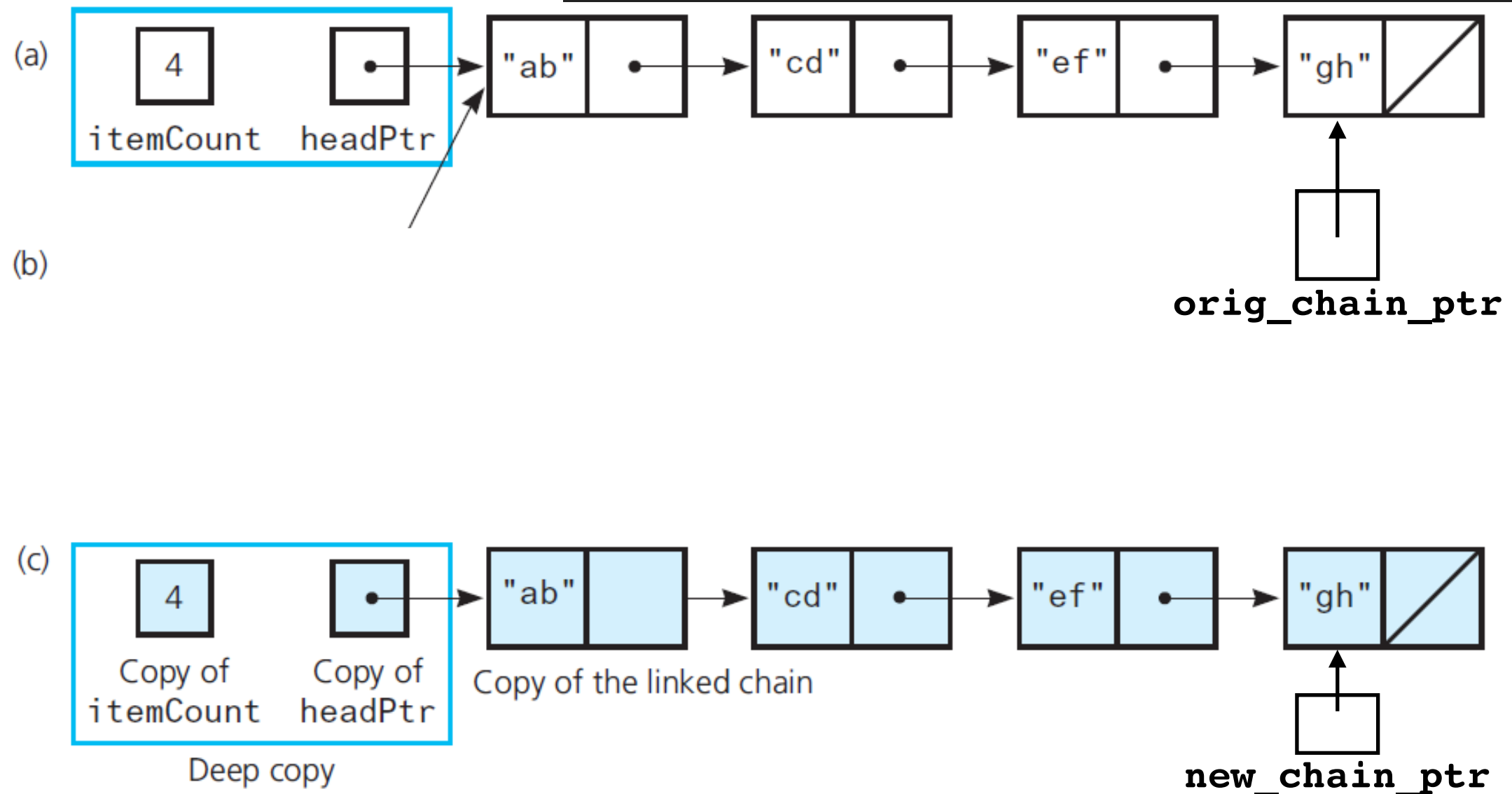
Deep vs Shallow Copy

```
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*")
- 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<class T>
class LinkedBag
{
public:
    LinkedBag();
    LinkedBag(const LinkedBag<T>& a_bag); // Copy 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_
```



Efficient



Expensive

THINK
WORST CASE

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(); // 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_
```



Efficient



Expensive

THINK
WORST CASE