

Linked-Based Implementation

Tiziana Ligorio

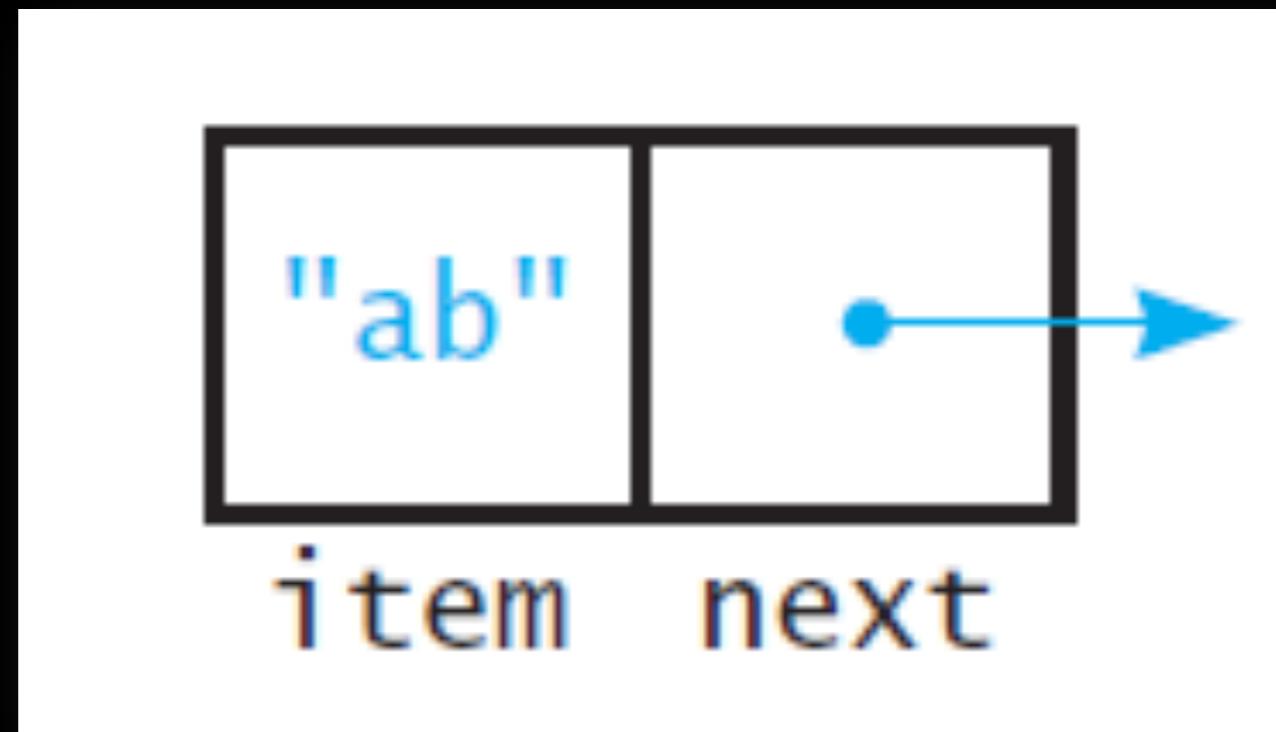
Data Organization

Place data within a **Node** object

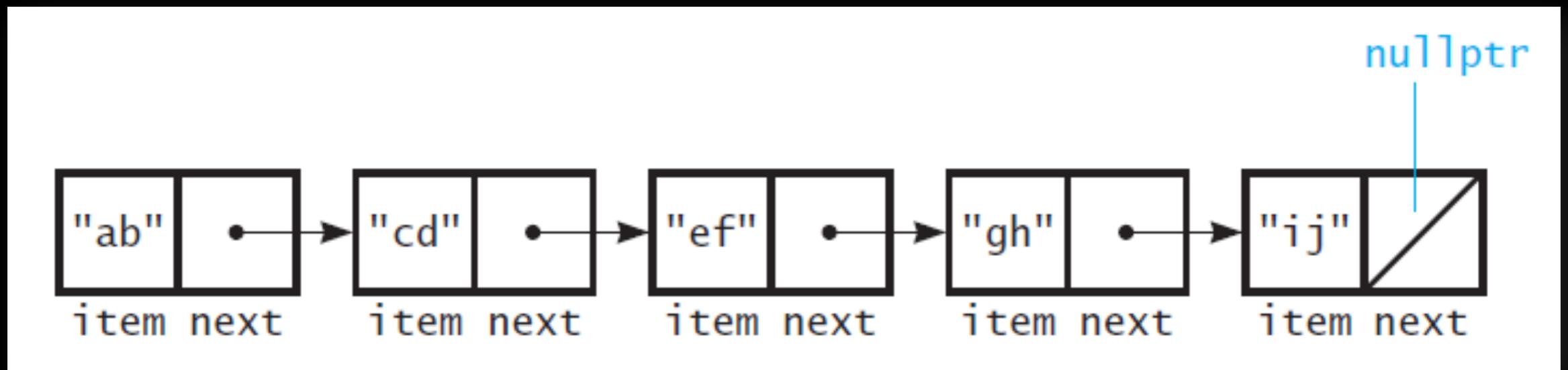
Link nodes into a **chain**



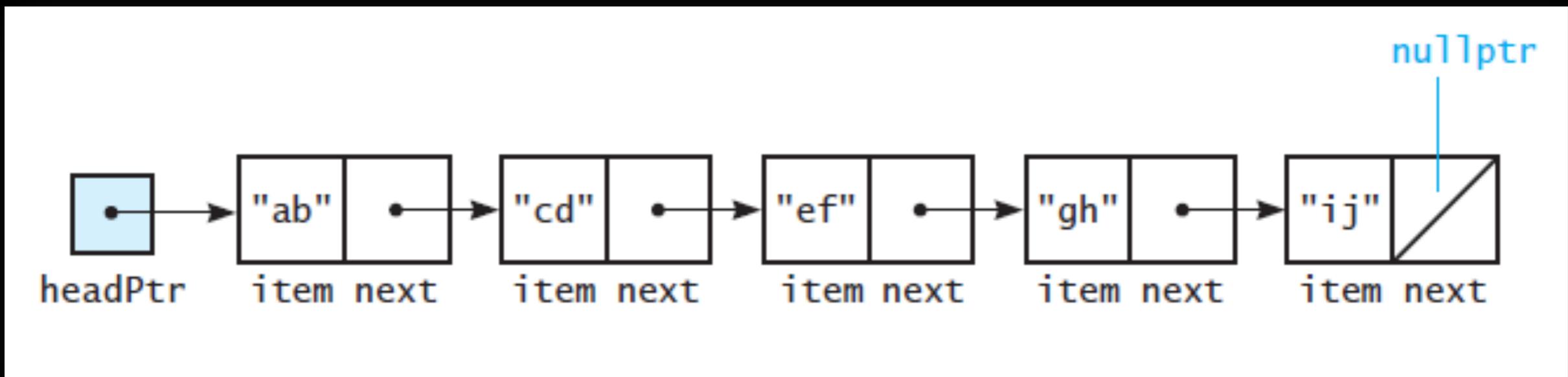
Node



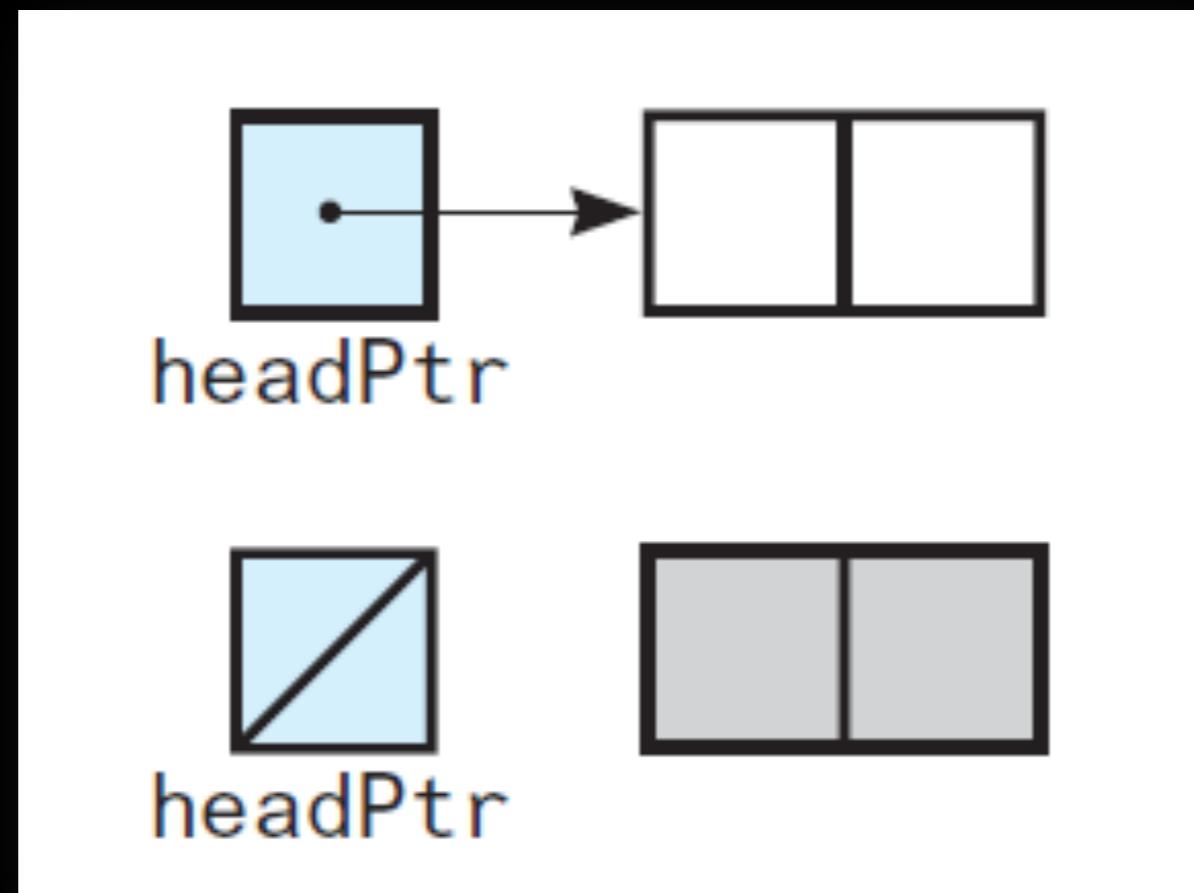
Chain



Entering the Chain



The Empty Chain



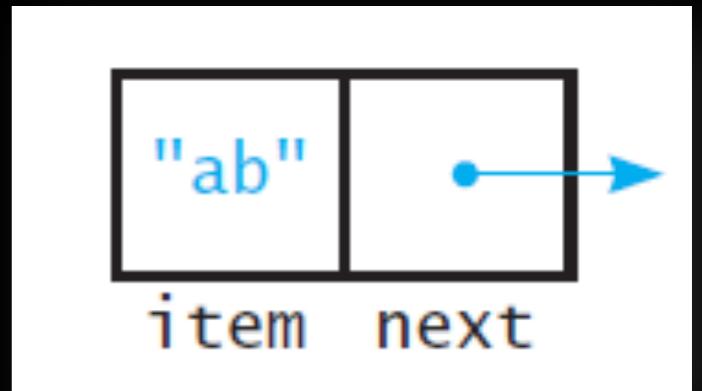
The Class Node

```
#ifndef NODE_H_
#define NODE_H_

template<class ItemType>
class Node
{
public:
    Node();
    Node(const ItemType& an_item);
    Node(const ItemType& an_item, Node<ItemType>* next_node_ptr);
    void setItem(const ItemType& an_item);
    void setNext(Node<ItemType>* next_node_ptr);
    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_
```

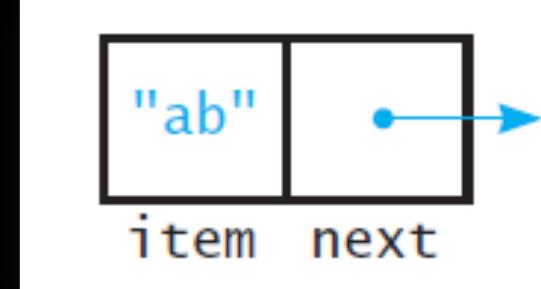


Node Implementation

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

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

The Constructors



```
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

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

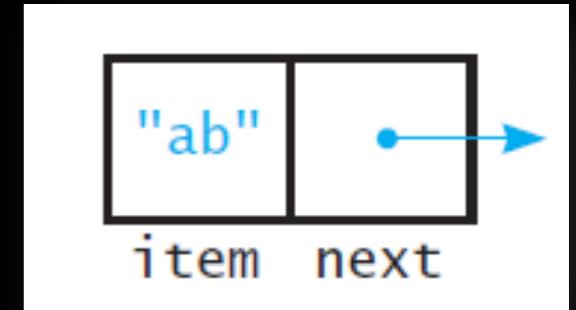
The “*setData*” members

```
template<typename 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

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

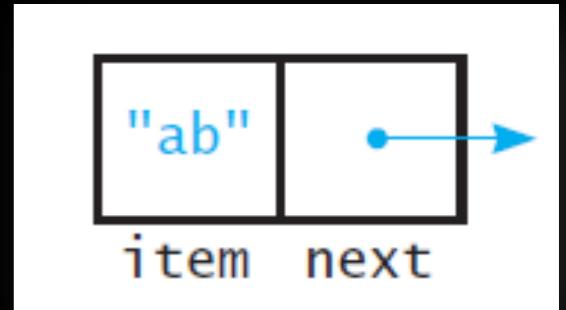
```
template<typename ItemType>
ItemType Node<ItemType>::getItem() const
{
    return item_;
}

} // end getItem
```

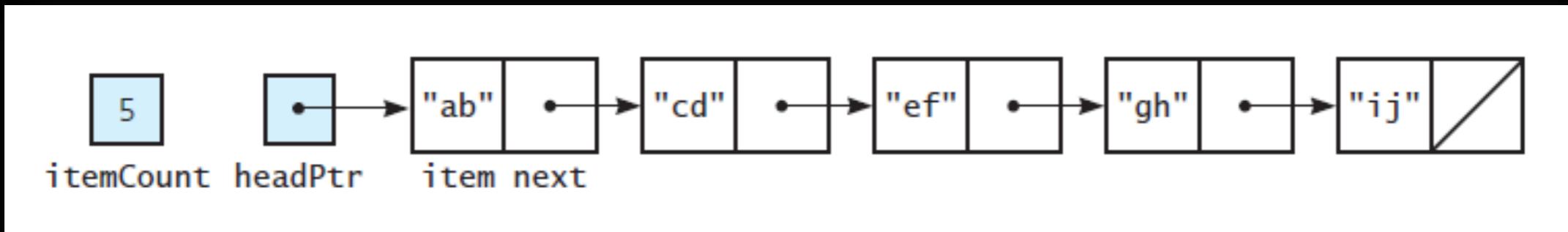
```
template<class ItemType>
Node<ItemType>* Node<ItemType>::getNext() const
{
    return next_;
}

} // end getNext
```

The “*getData*” 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
```

```
#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:
```

???

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

The Class LinkedBag

The Class LinkedBag

```
#ifndef LINKED_BAG_H_
#define LINKED_BAG_H_
#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_
```

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"

template<typename ItemType>
LinkedBag<ItemType>::LinkedBag() : head_ptr_(nullptr),
item_count_(0)
{
}

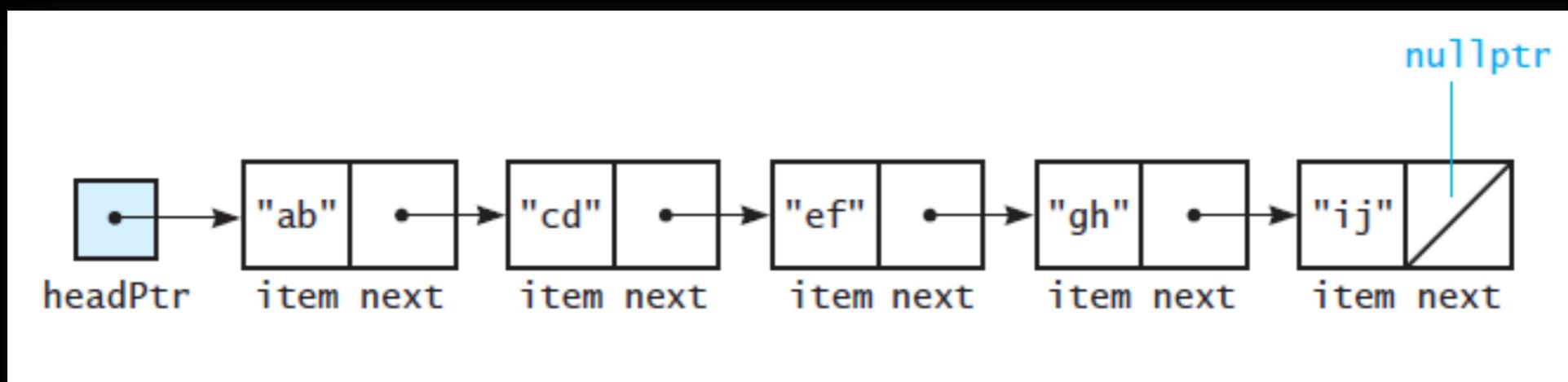
} // end default constructor
```

The default constructor

Private data member
initialization

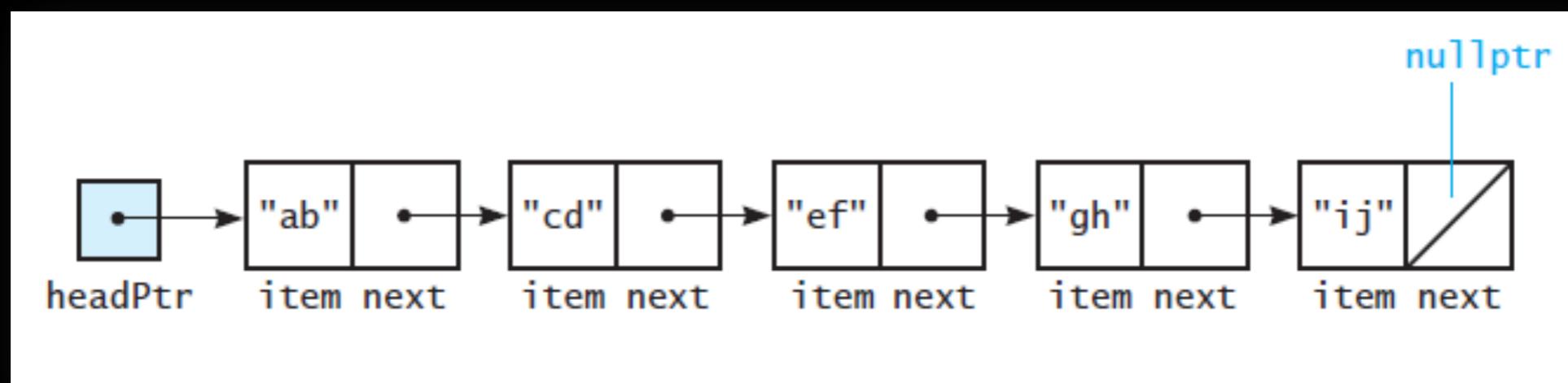
```
add(const ItemType& new_entry)
```

Where should we add?

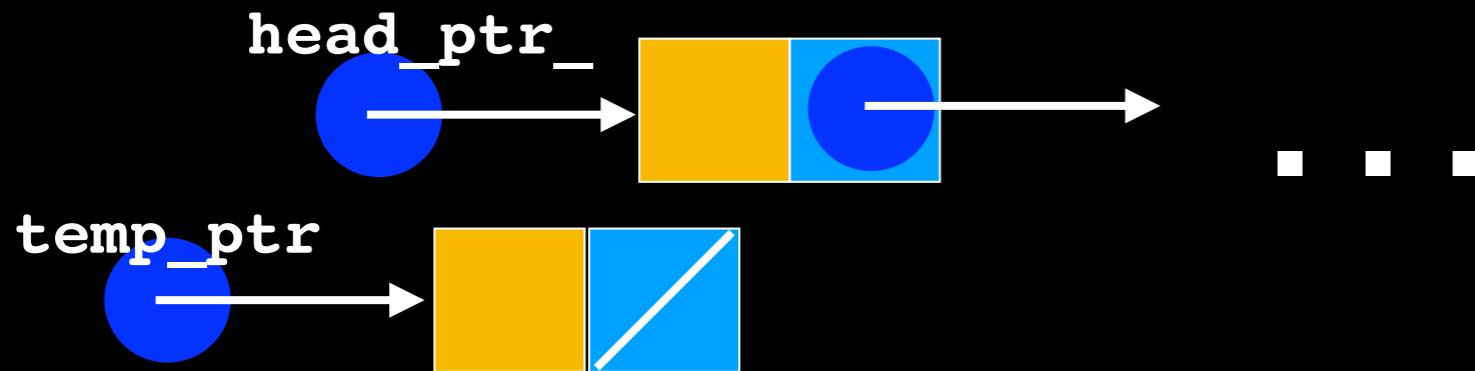


Lecture Activity

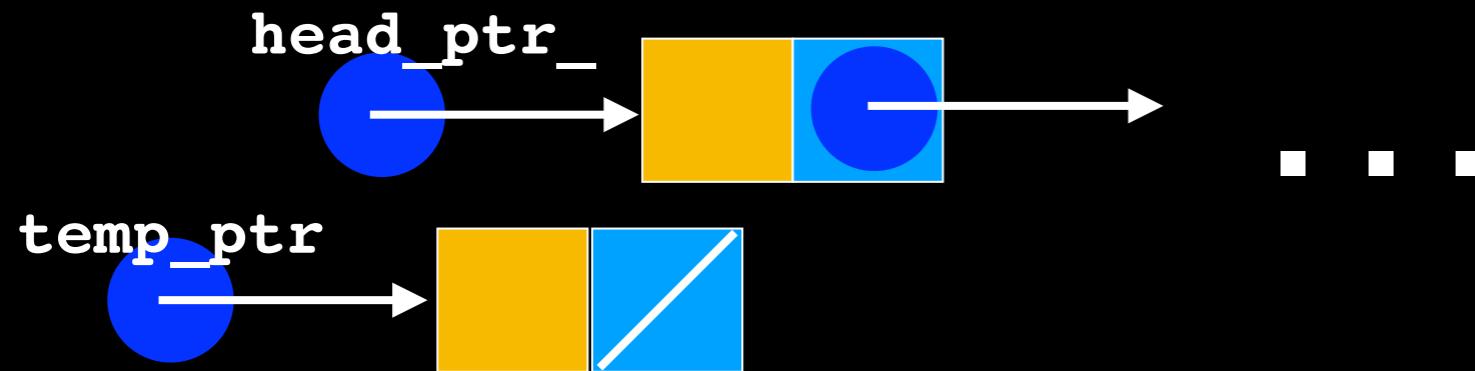
Write **pseudocode** for a sequence of steps to add to the **front** of the chain



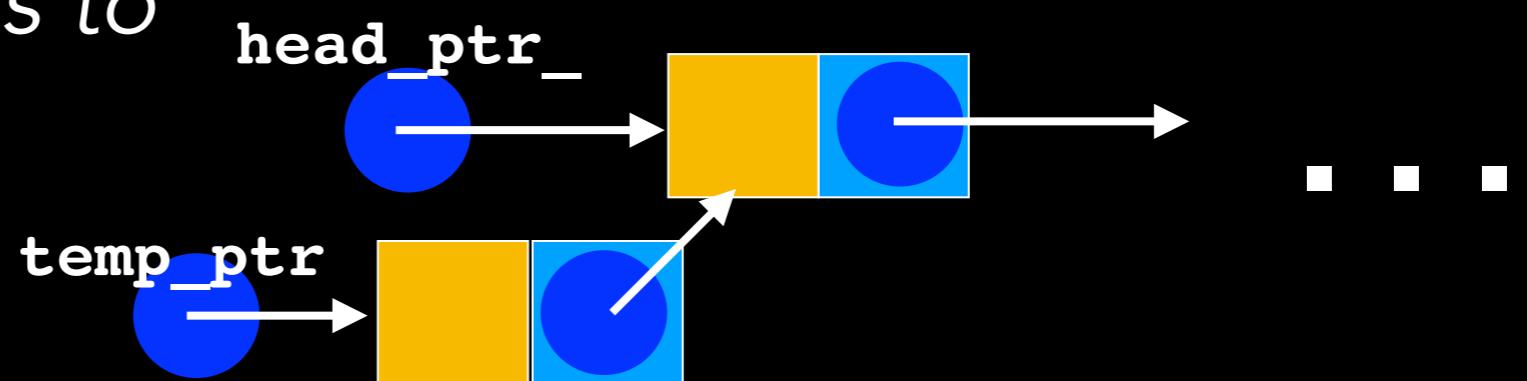
*Instantiate a **new** node and let a **temp pointer** point to it*



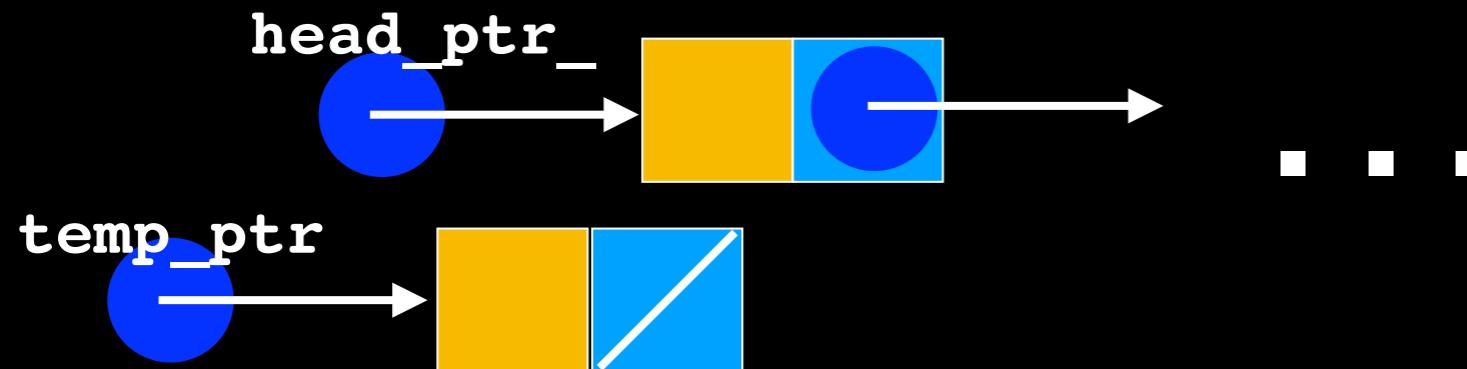
Instantiate a **new** node and let a **temp pointer** point to it



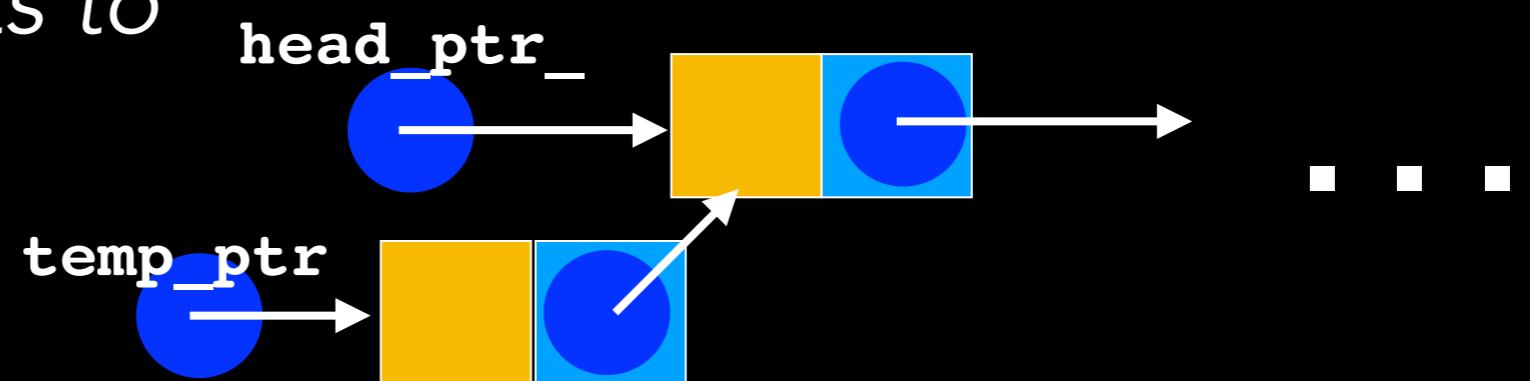
Let the **next pointer** of the **new node** point to the same node `head_ptr_` points to



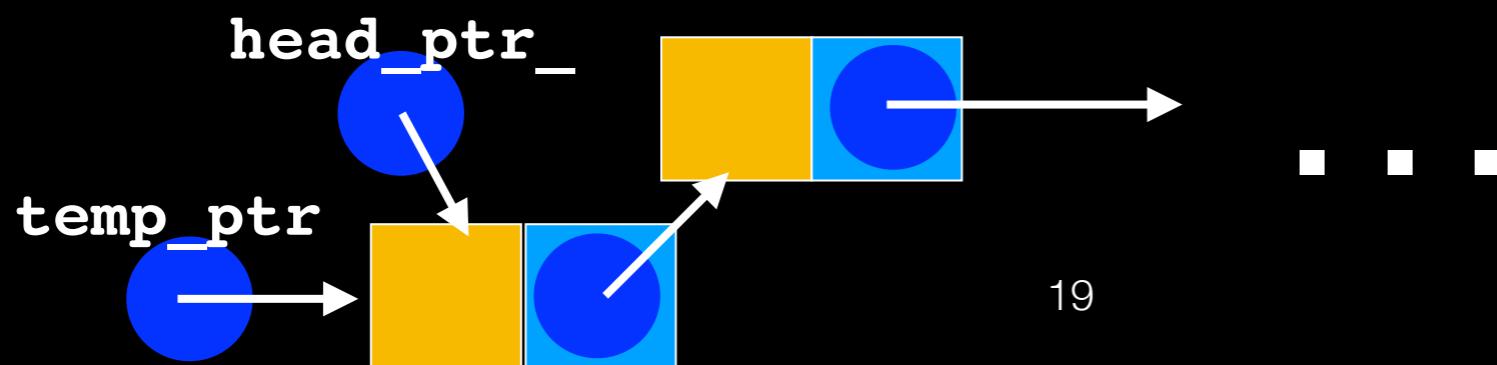
Instantiate a **new** node and let a **temp pointer** point to it



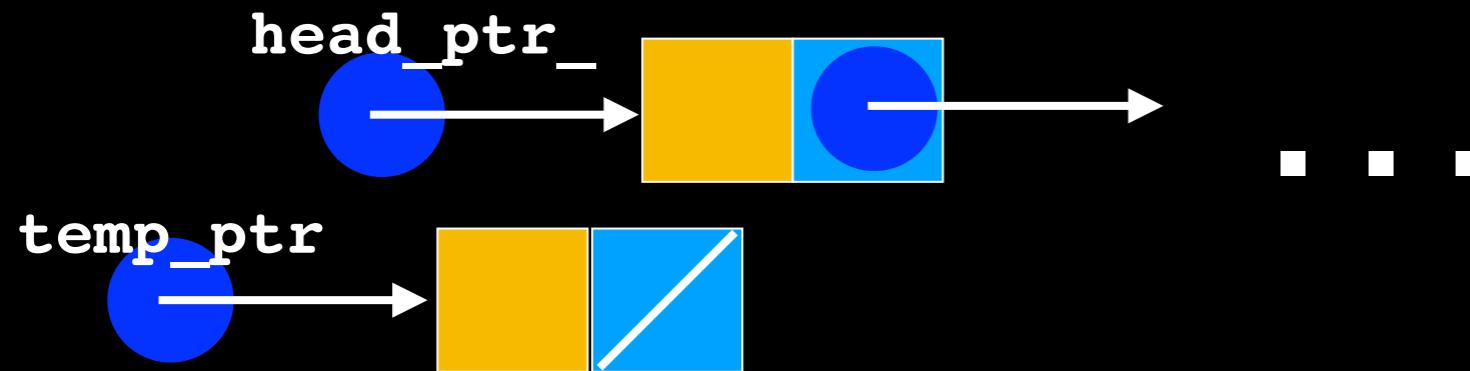
Let the **next pointer** of the **new node** point to the same node `head_ptr_` points to



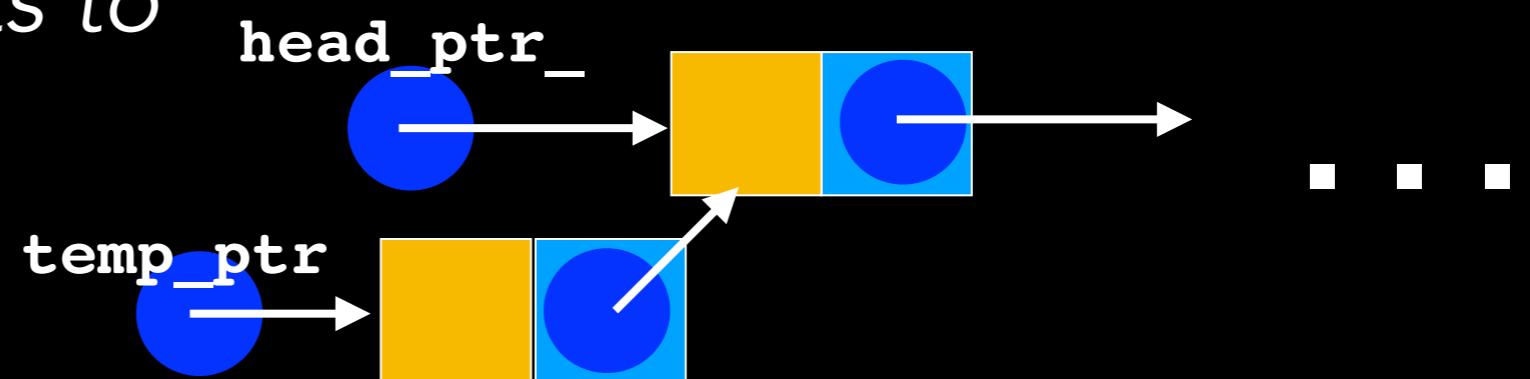
Let `head_ptr_` point to the **new node**



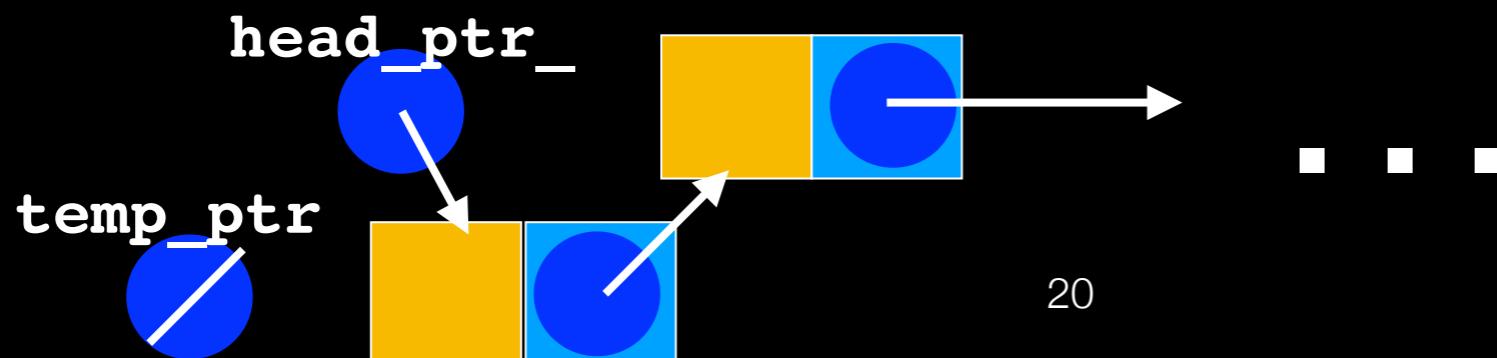
Instantiate a **new** node and let a **temp pointer** point to it



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
```

LinkedBag Implementation

```
#include "LinkedBag.hpp"

template<typename ItemType>
bool LinkedBag<ItemType>::add(const ItemType& new_entry)
{
    // Add to beginning of chain: new node references rest of chain;
    // (head_ptr_ is null if chain is empty)
    Node<ItemType>* new_node_ptr = new Node<ItemType>;
    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_++;

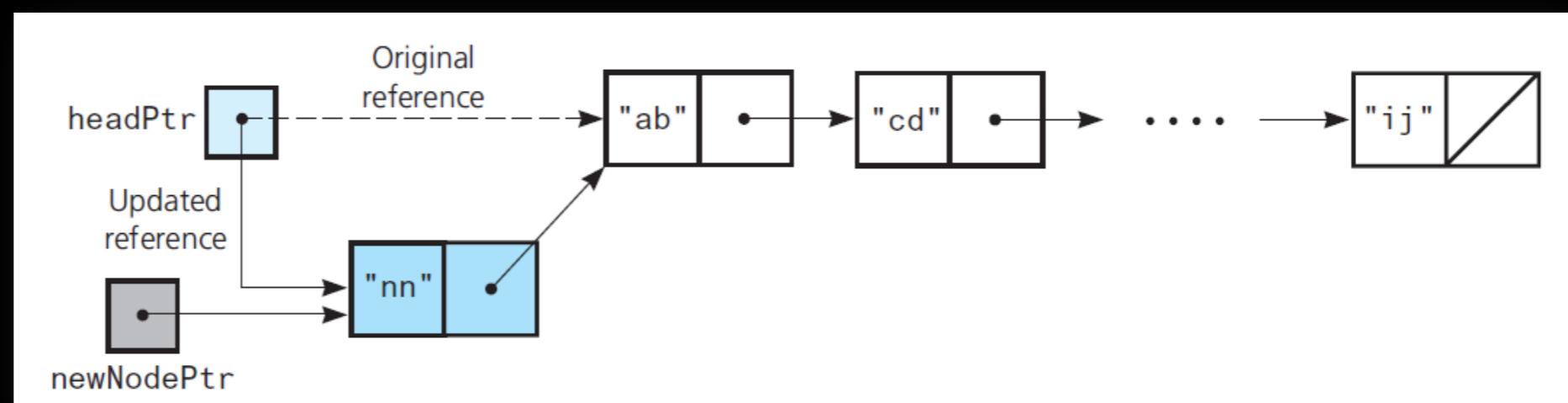
    return true;
} // end add
```

The add method

Add at beginning of chain is easy
because we have head_ptr_

Dynamic memory
allocation

Adding nodes to the heap!

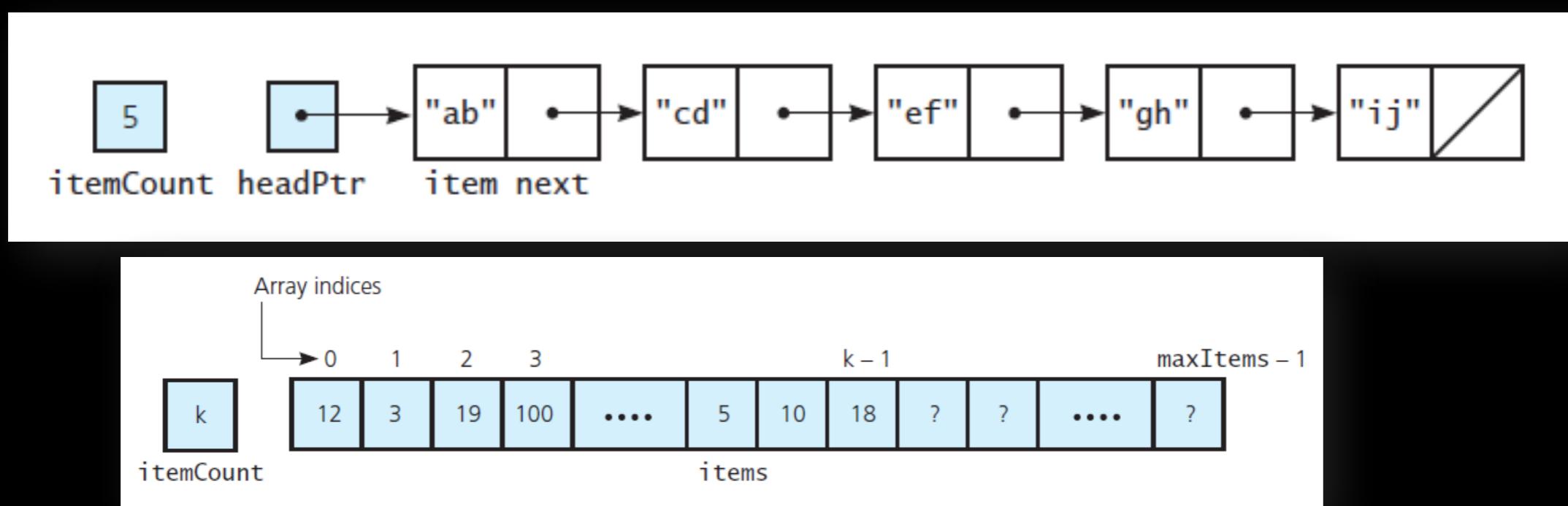


Efficiency

Create a new node and assign two pointers

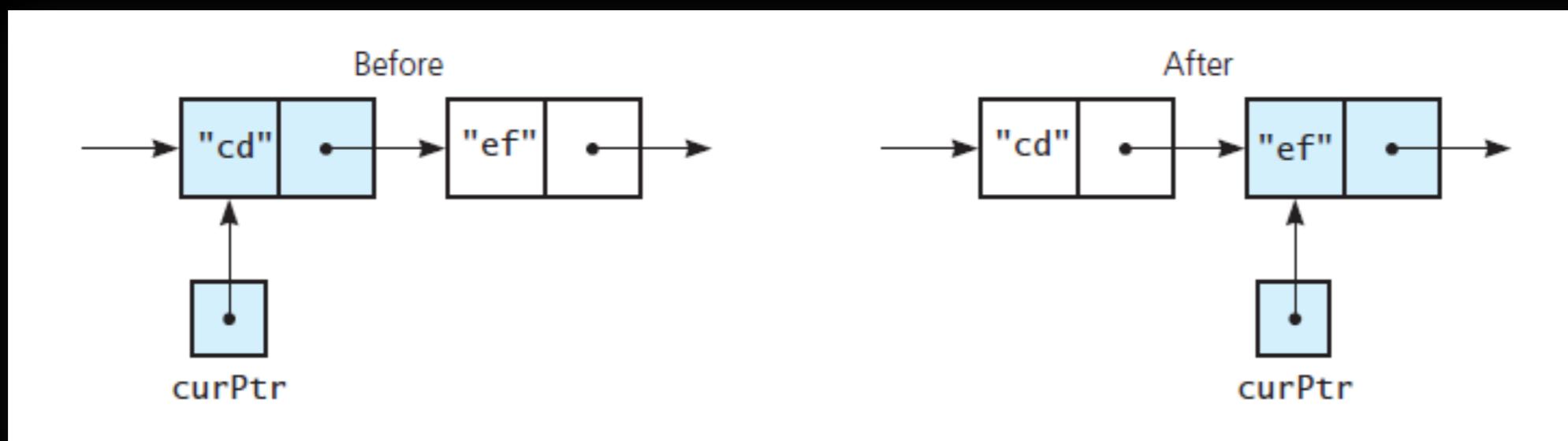
What about adding to end of chain?

What about adding to front of array?



Lecture Activity

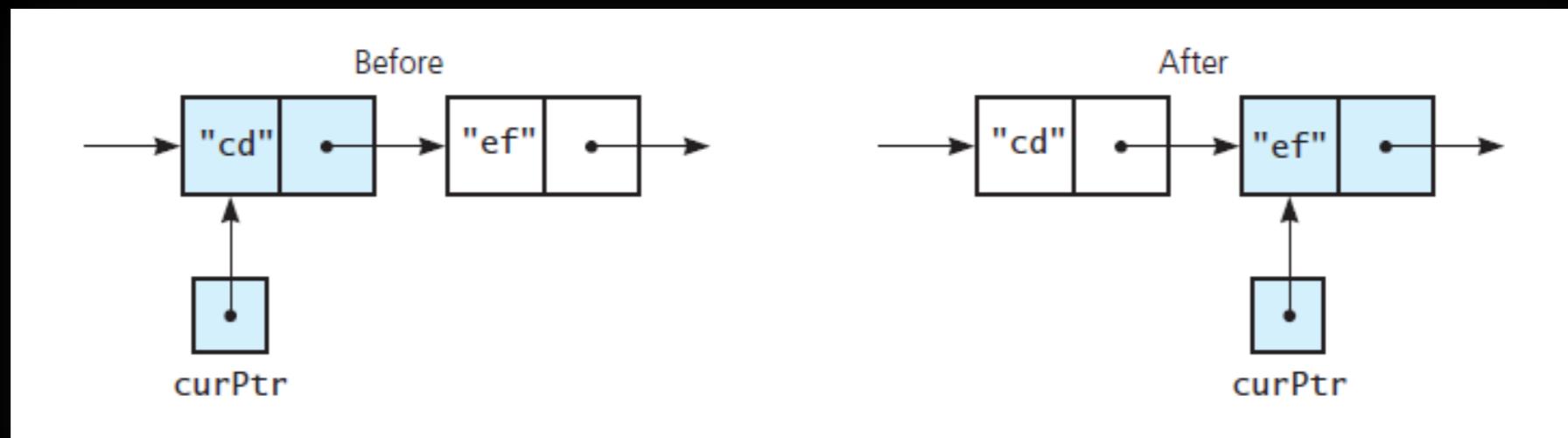
Write **Pseudocode** to traverse the chain from first node to last



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"
```

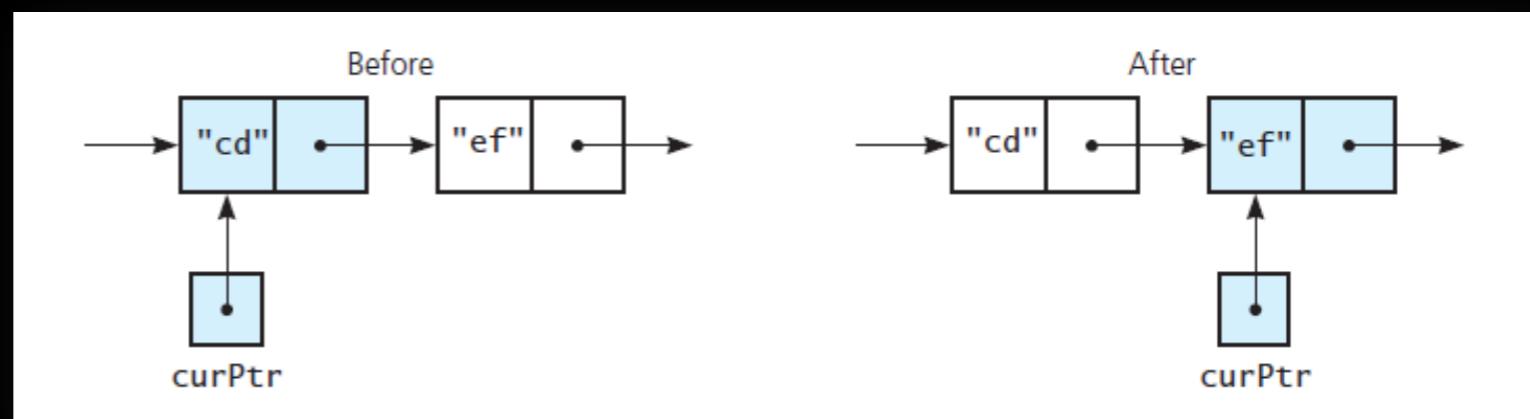
```
template<typename ItemType>
std::vector<ItemType> LinkedBag<ItemType>::toVector() const
{
    std::vector<ItemType> bag_contents;
    Node<ItemType>* cur_ptr = head_ptr_;

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

    return bag_contents;
} // end toVector
```

The toVector method

Traversing:
Visit each node
Copy it



LinkedBag Implementation

Similarly `getFrequencyOf` will:
traverse the chain and
count frequency of (count each) `an_entry`

LinkedBag Implementation

```
#include "LinkedBag.hpp"

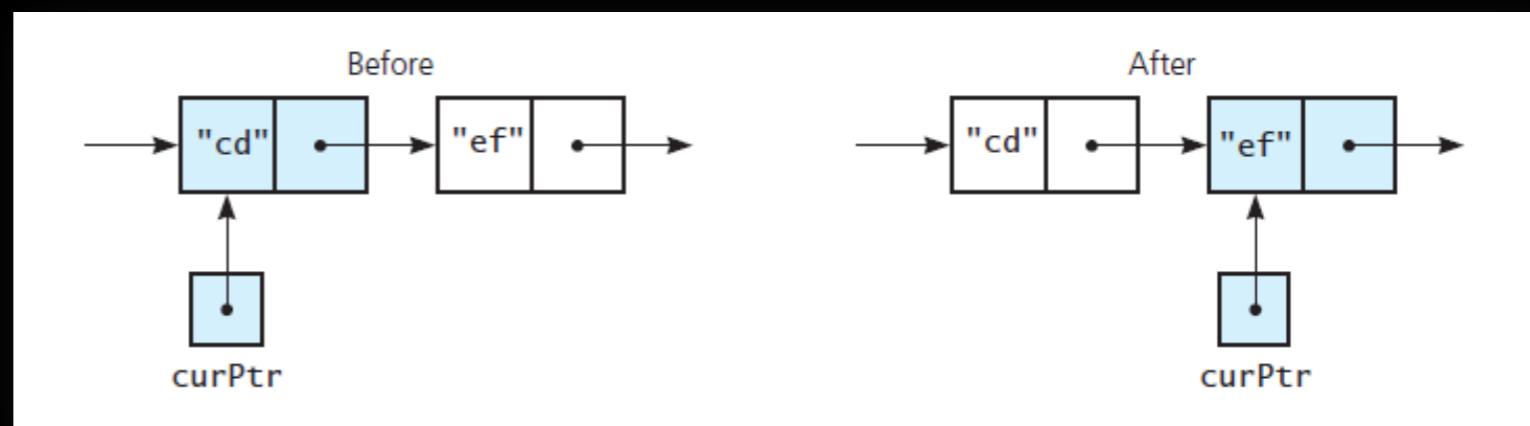
template<typename ItemType>
Node<ItemType>* LinkedBag<ItemType>::getPointerTo(const ItemType& an_entry) const
{
    bool found = false;
    Node<ItemType>* cur_ptr = head_ptr_;

    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

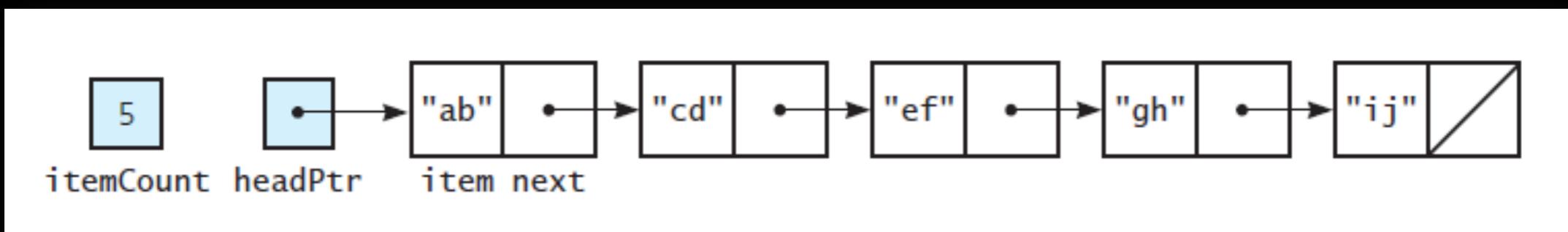
No fixed number of steps

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?



LinkedBag Implementation

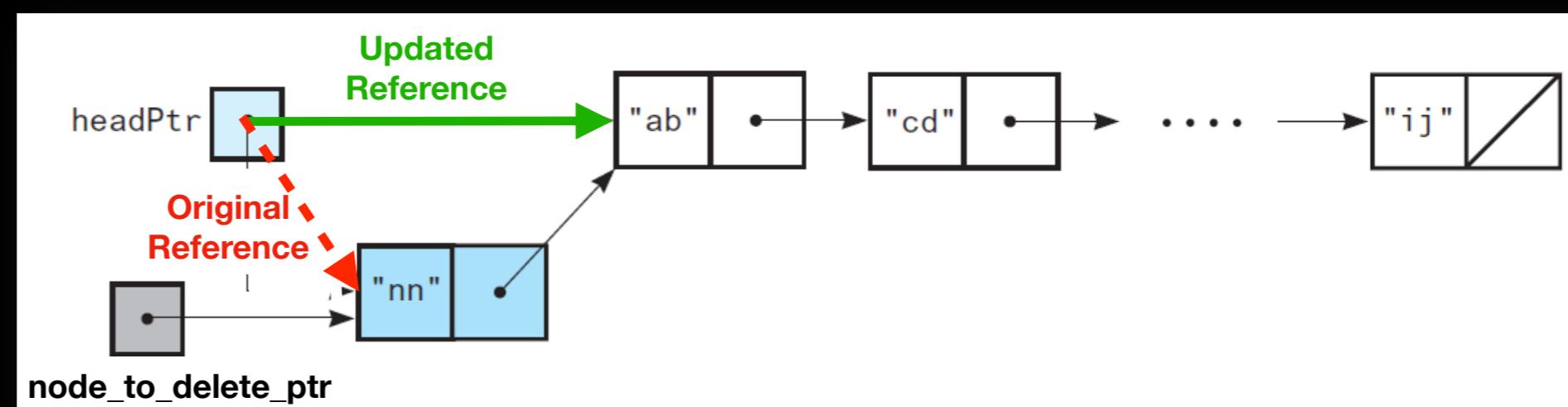
```
#include "LinkedBag.hpp"
template<typename ItemType> The remove method
bool LinkedBag<ItemType>::remove(const ItemType& an_entry)
{
    Node<ItemType>* entry_ptr = getPointerTo(an_entry);
    bool can_remove = (entry_ptr != nullptr);
    if (can_remove)
    {
        // Copy data from first node to found node
        entry_ptr->setItem(head_ptr_->getItem());
        // Delete first node
        Node<ItemType>* 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
an_entry

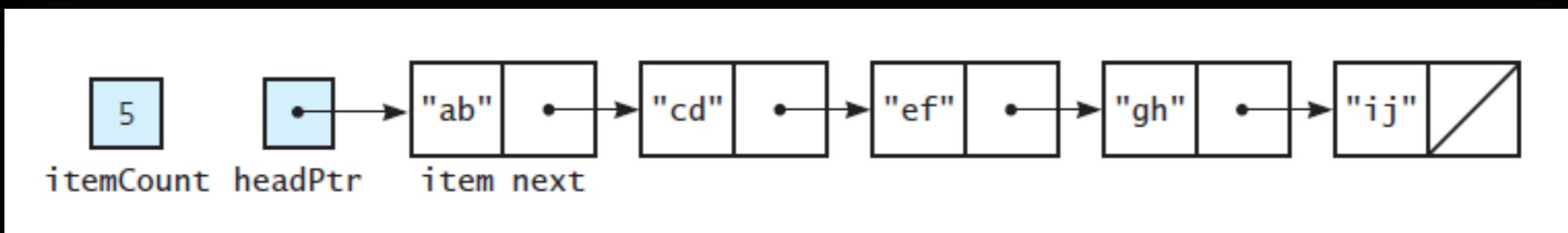
Deleting first
node is easy

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<typename ItemType>
void LinkedBag<ItemType>::clear()
{
    Node<ItemType>* 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;

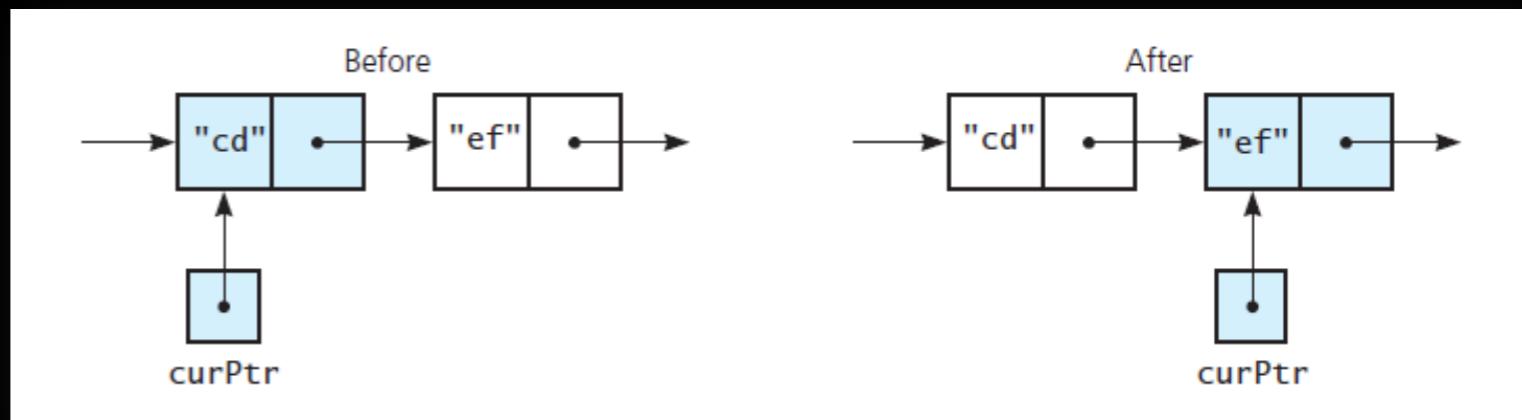
        node_to_delete_ptr = head_ptr_;
    } // end while
    // head_ptr_ is nullptr; node_to_delete_ptr is nullptr

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

The `clear` method

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

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



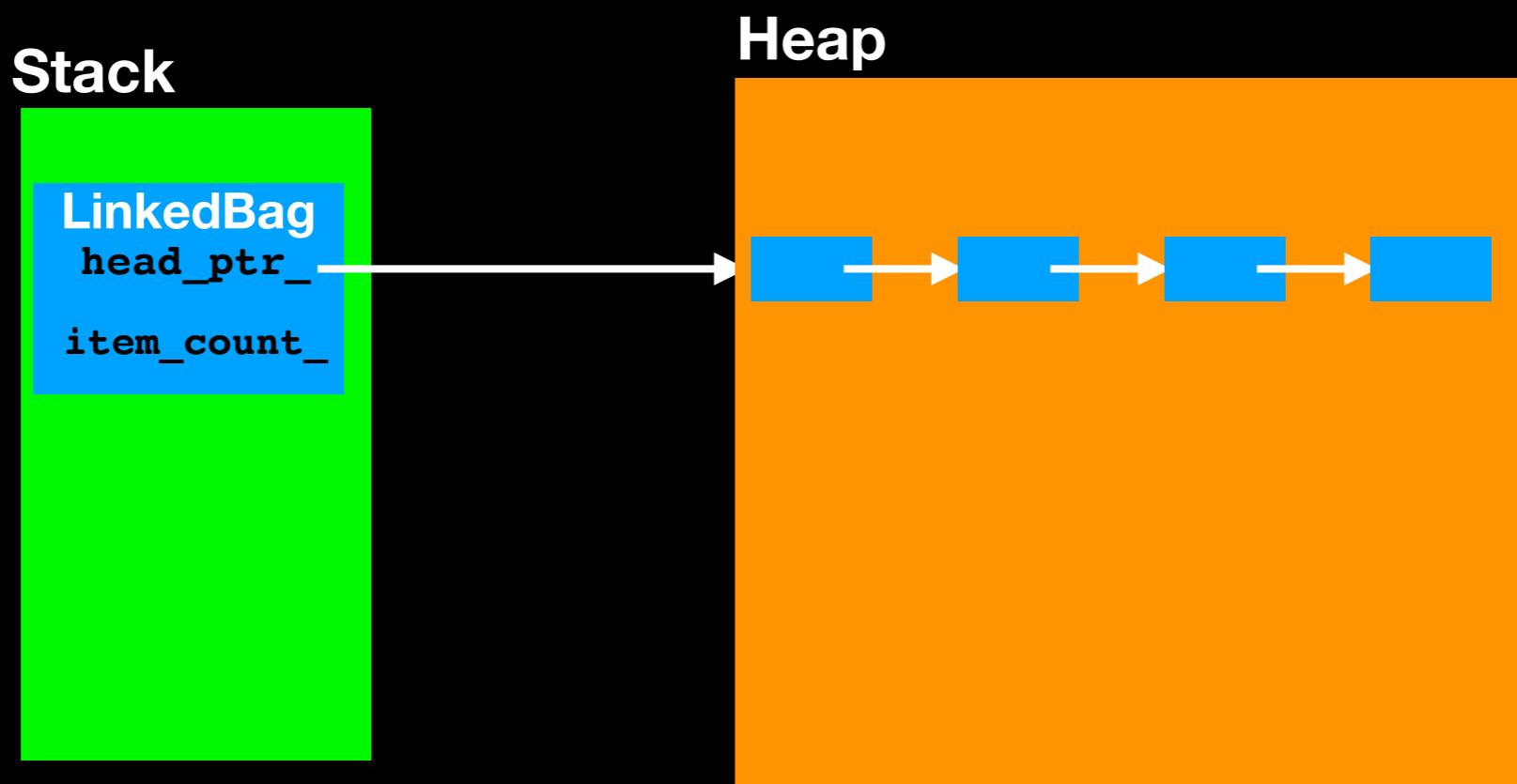
Dynamic Memory Considerations

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

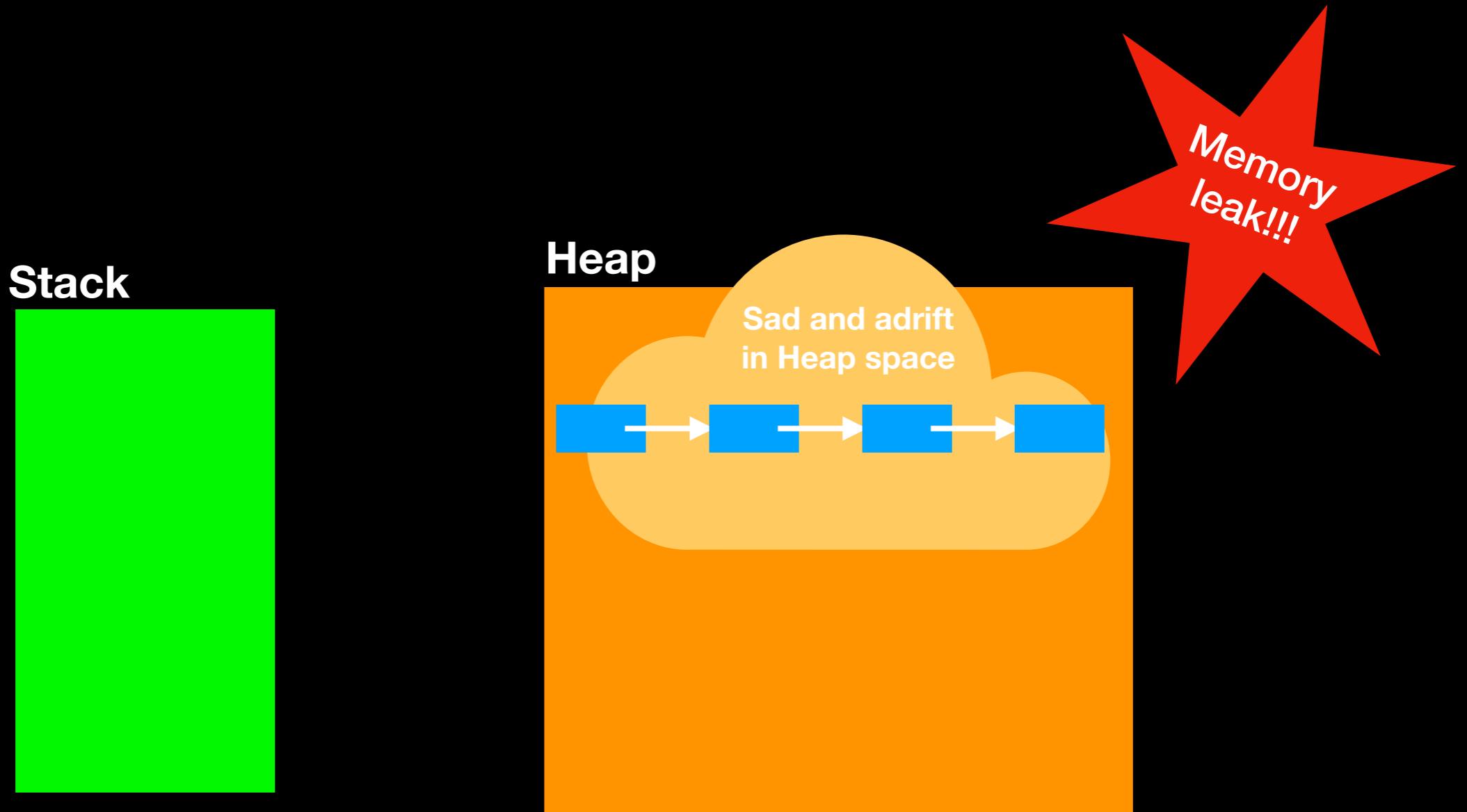
Programmer must ensure this memory is deallocated when object is destroyed!

Avoid memory leaks!!!!

What happens when object goes out of scope?



What happens when object goes out of scope?



LinkedBag Implementation

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

The destructor

```
template<typename ItemType>
LinkedBag<ItemType>::~LinkedBag()
{
    clear();
}

// end destructor
```

Ensure heap space is returned to the system

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

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