

Linked-Based Implementation



Tiziana Ligorio
Hunter College of The City University of New York

Today's Plan



Announcements

Linked-Based
Implementation

Announcements

Project 3 opens today:

- Give yourself time to resolve unforeseen issues
- Give yourself time to seek help should you need it

STUBS for partial credit:

- All classes must be there, all functions must be there
- You want credit for what you implement correctly even if you don't correctly implement everything.
- Start with STUBS and implement incrementally

MUST HAVE APPOINTMENT FOR WEDNESDAY ONLINE TUTORING

Make appointments through Navigate

No lecture next Tuesday

What does a compiler do?

Get Help!

If you are still struggling with:

- Running, testing, compiling, debugging
- Basic OOP concepts (135 material)
- Git, GitHub, GitHub Classroom, Gradescope
- Keeping up with communication and reading project specifications

**YOU NEED TO GET HELP IMMEDIATELY,
NOT ON THE DUE DATE!!!**

Starting next week you **spend at least 3 hours EVERY DAY** in **tutoring AND come to office hours every week.**

Request

4 different Gradescope accounts for Chen

Please include your first name so we can look at your code when you ask for help

Recap

- **Pointers**
 - Variable that holds address of same type
 - Must be nullptr if not pointing to something
 - Can change what it points to
 - Can access values of what it points to
- **Dynamic memory allocation**
 - Can dynamically allocate memory on **Heap** through pointers
 - Use keyword **new** to allocate
 - Use keyword **delete** to deallocate and **MUST** set pointer to some other value
 - Beware of memory leaks
 - Beware of dangling pointers

Let's try a different
implementation for Bag

Link-Based Implementation



The Header File

```
#ifndef LINKED_BAG_H_
#define LINKED_BAG_H_

template<class T>
class LinkedBag
{
public:
    LinkedBag();
    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

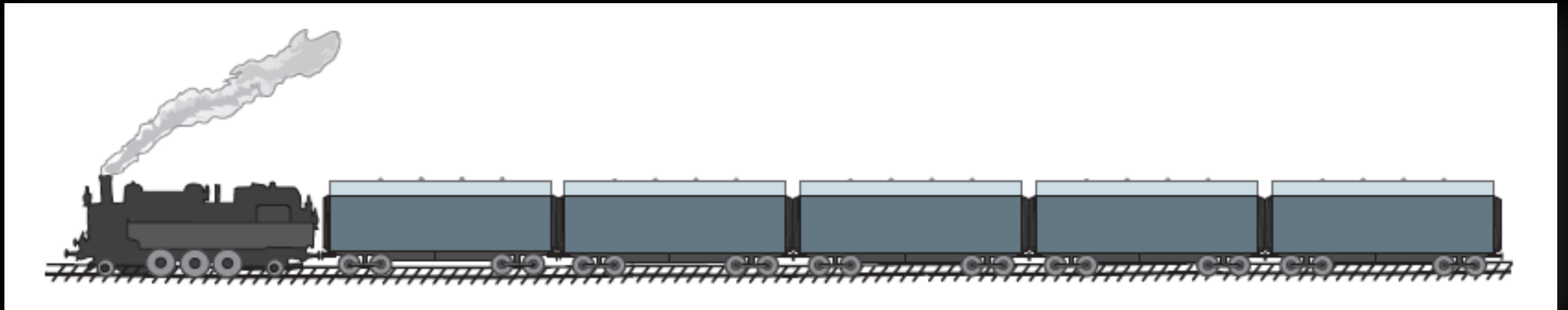
#include "LinkedBag.cpp"
#endif
```

Same interface, different implementation

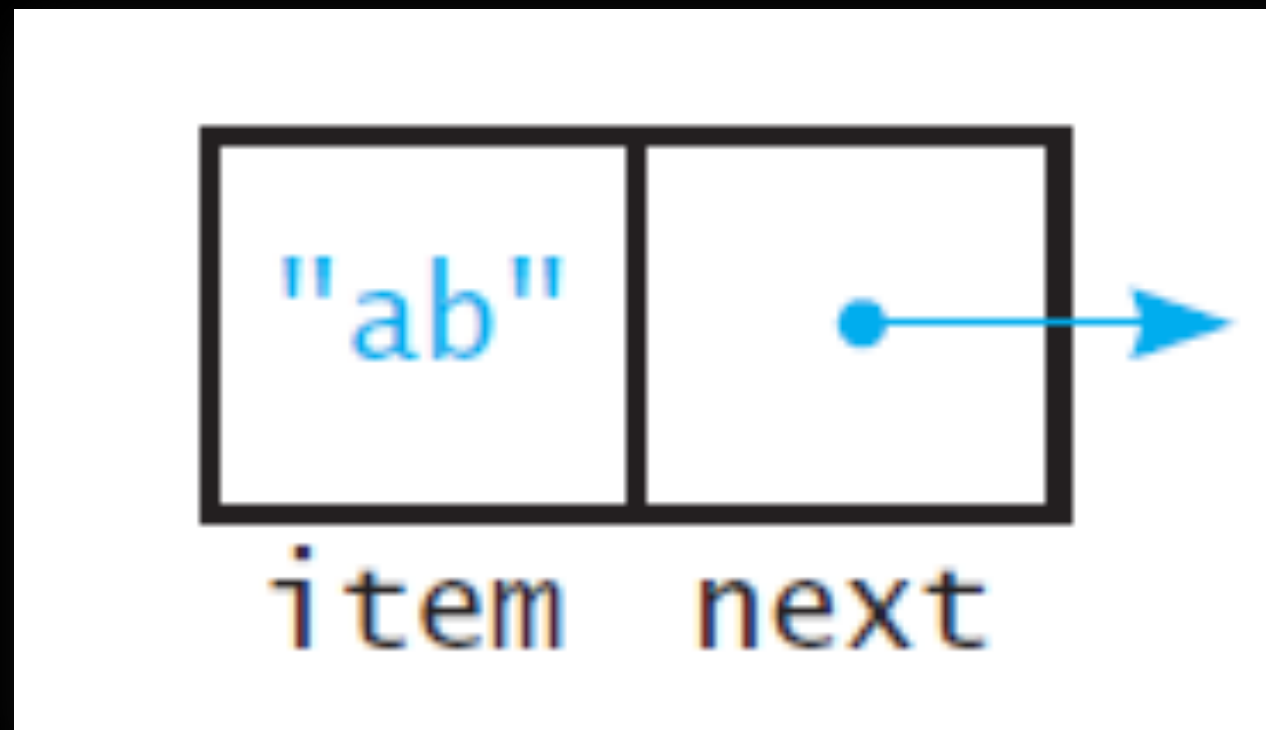
Data Organization

Place data within a **Node** object

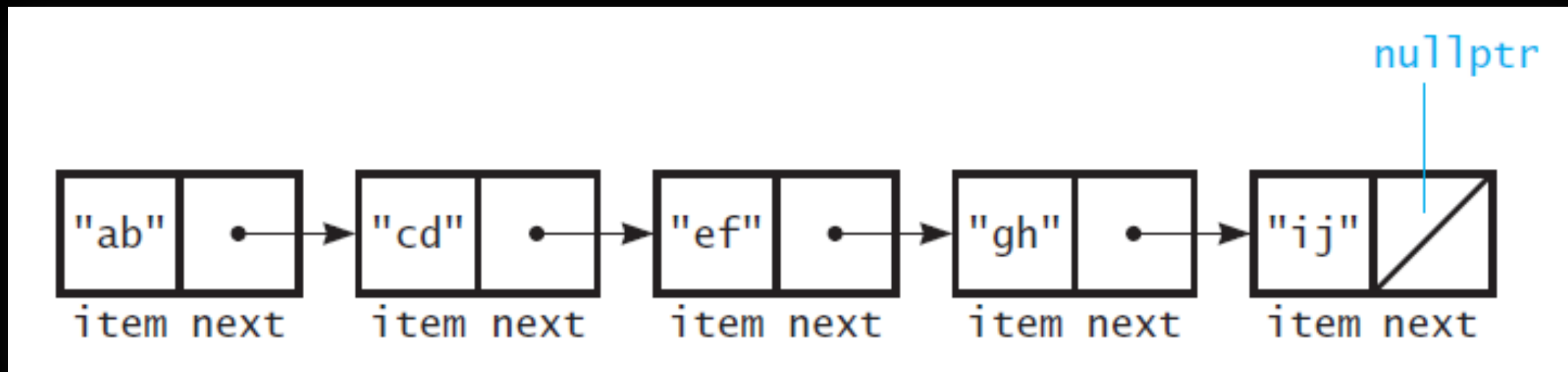
Link nodes into a **chain**



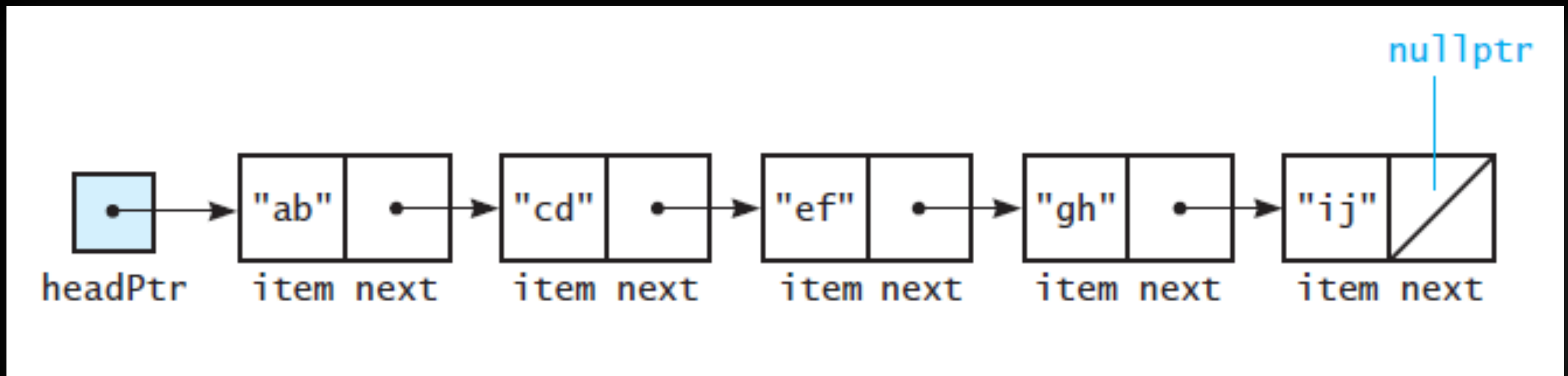
Node



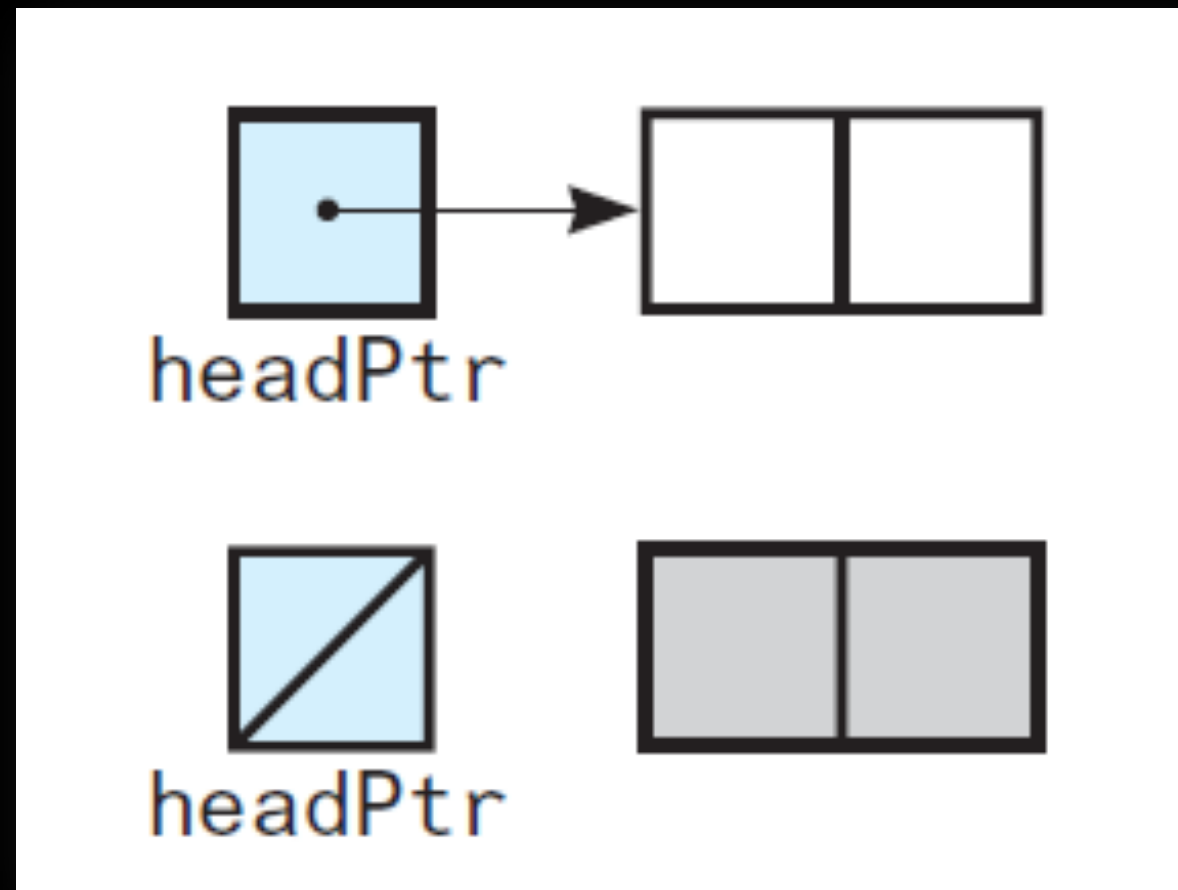
Chain



Entering the Chain



The Empty Chain



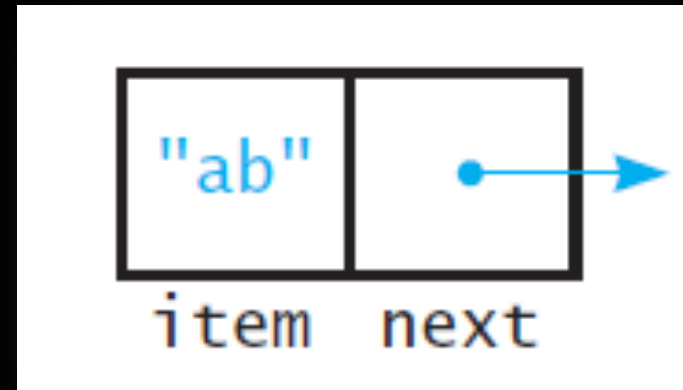
The Class Node

```
#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

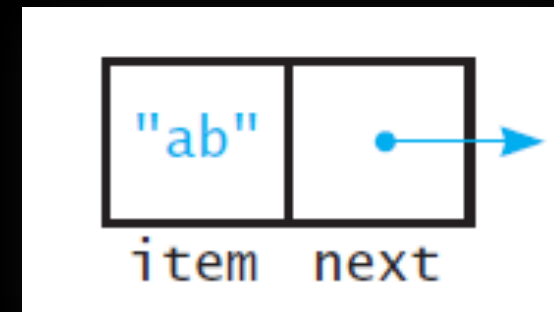
```
#include "Node.hpp"
```

```
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
```

The Constructors



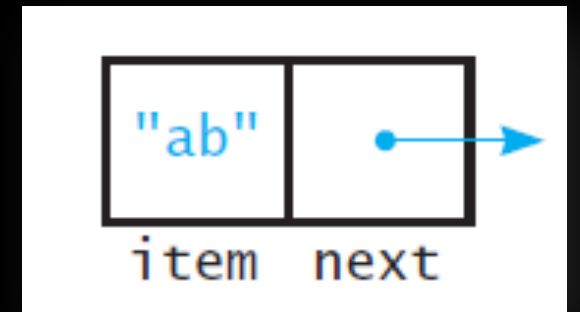
Node Implementation

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

```
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
```

The "setData" members



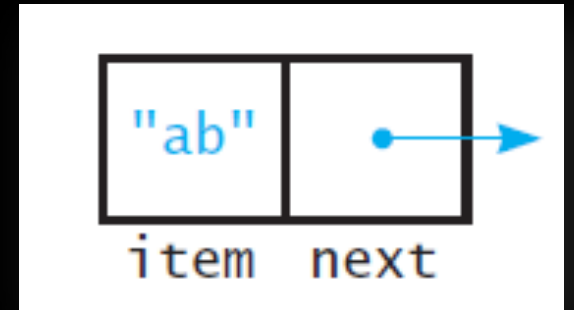
Node Implementation

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

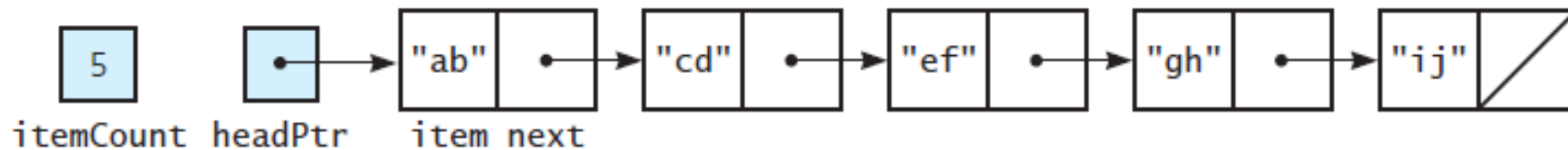
```
template<class T>  
T Node<T>::getItem() const  
{  
    return item_;  
} // end getItem
```

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

The "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
```

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_
```

Same interface, different implementation

The Class LinkedBag

```
#ifndef LINKED_BAG_H_
#define LINKED_BAG_H_

#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"
```

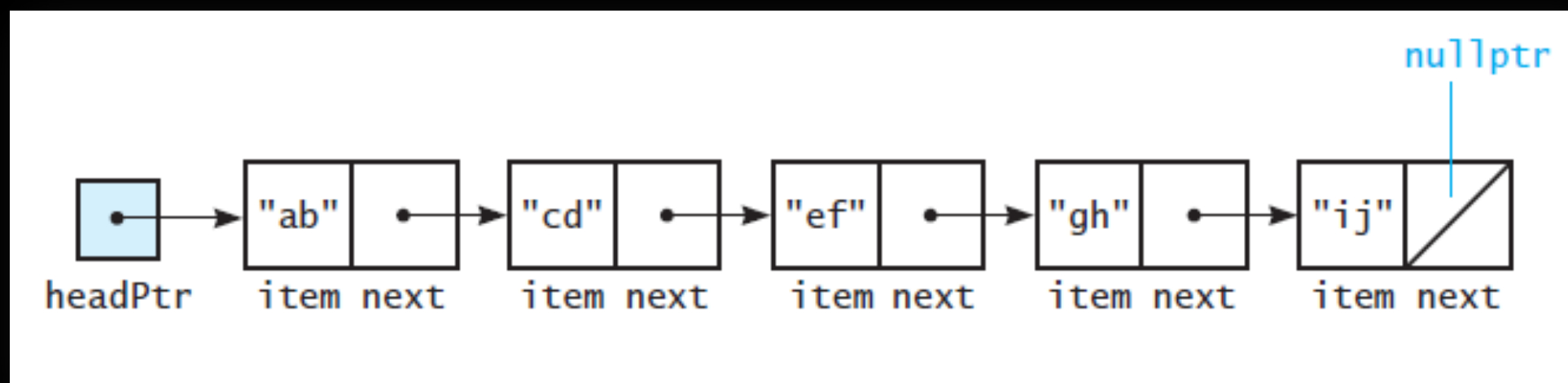
The default constructor

```
template<class T>
LinkedBag<T>::LinkedBag() : head_ptr_{nullptr},
item_count_{0}
{
} // end default constructor
```

Private data member
initialization

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

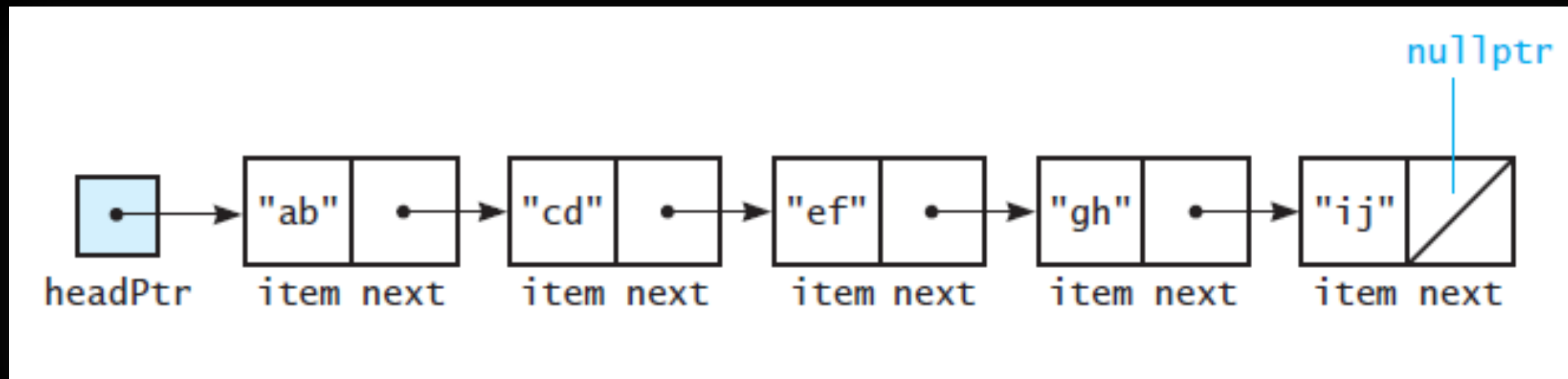
Where should we add?



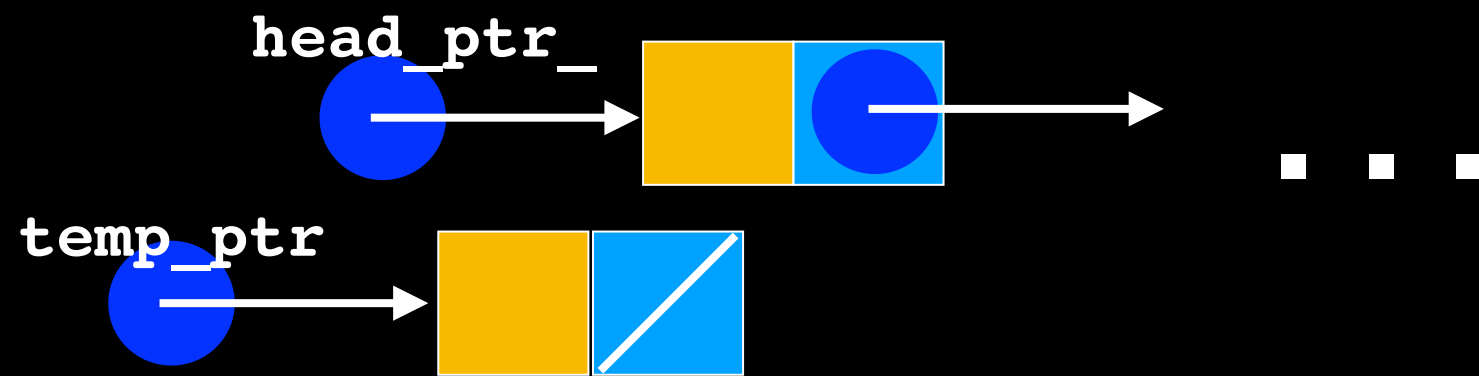
Lecture Activity

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

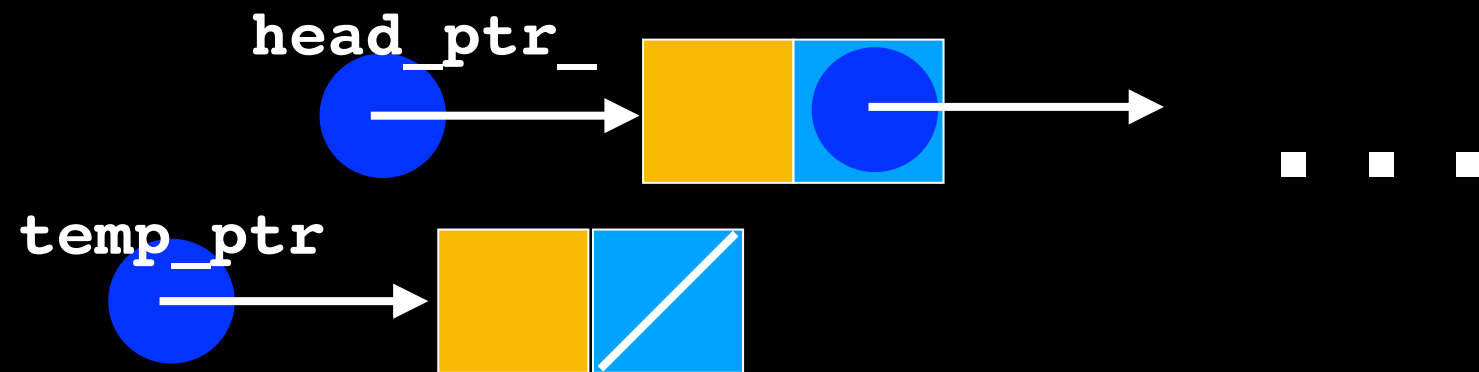
**Lecture 4 Assignment on Gradescope
Login and submit NOW!!!**



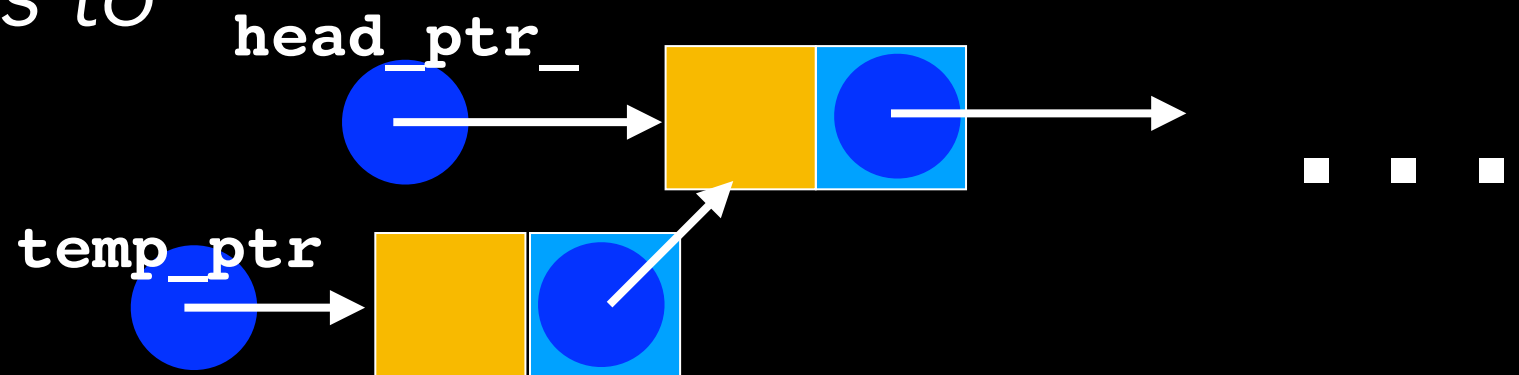
Instantiate a *new* node and let a *temp pointer* point to it



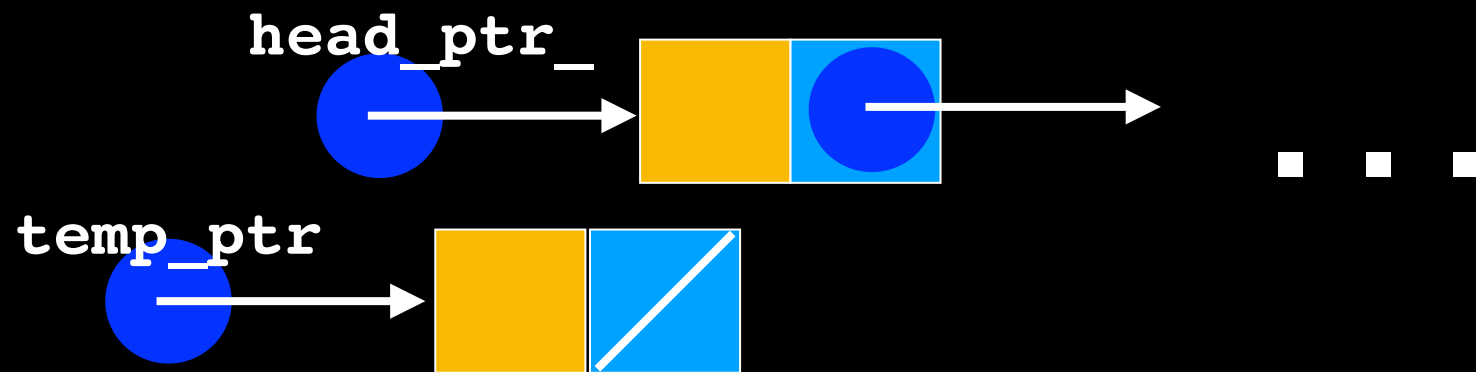
Instantiate a *new* node and let a *temp pointer* point to it



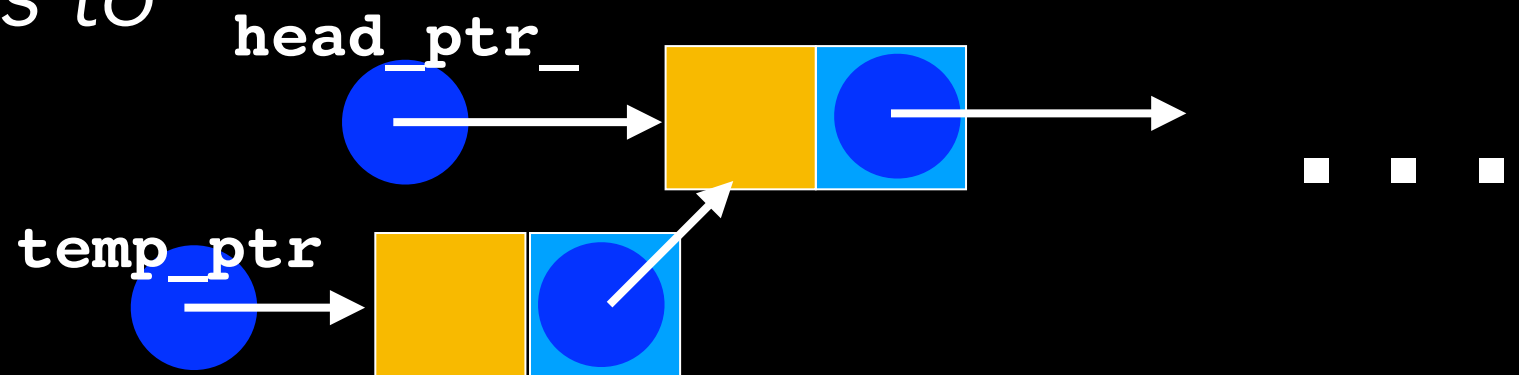
Let the *next pointer* of the *new node* point to the same node *head_ptr_* points to



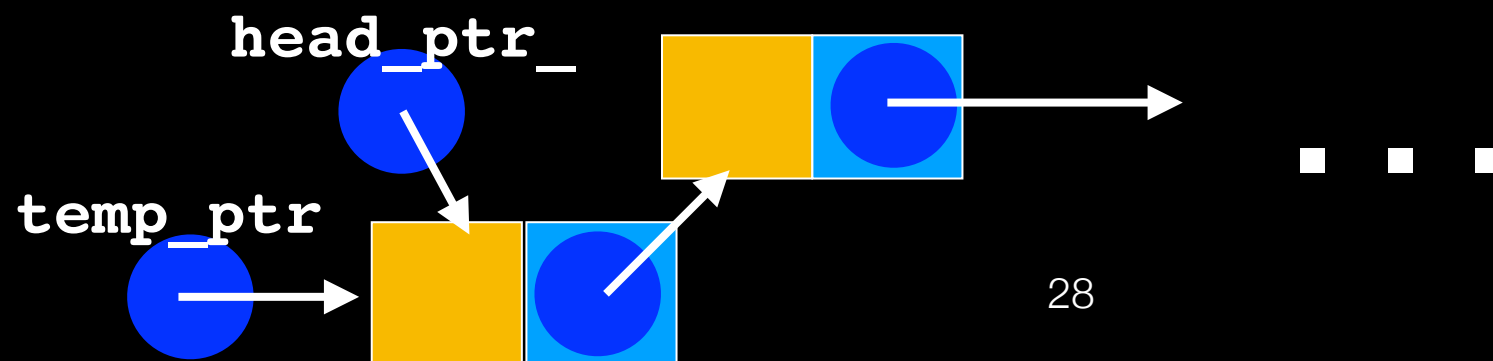
Instantiate a *new* node and let a *temp pointer* point to it



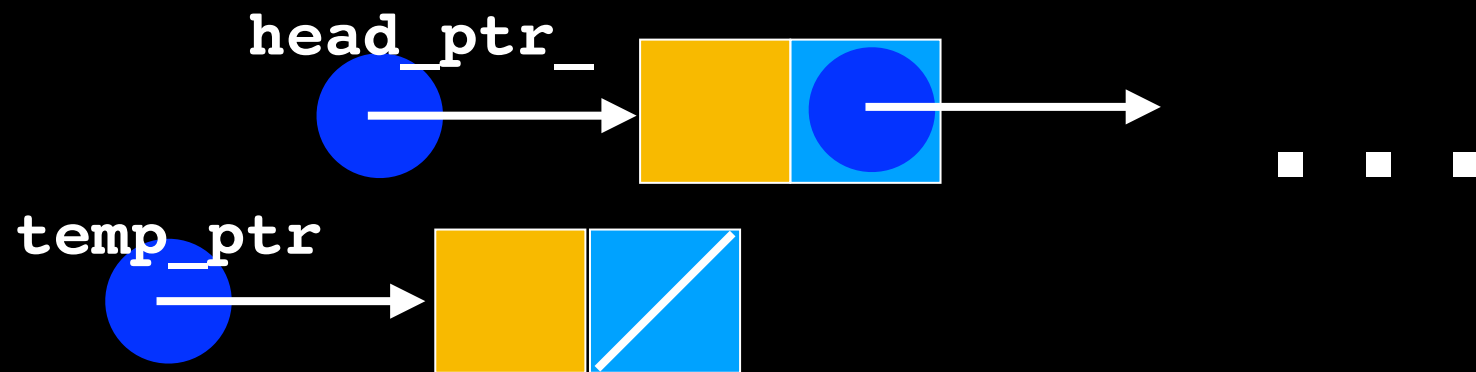
Let the *next pointer* of the *new node* point to the same node *head_ptr_* points to



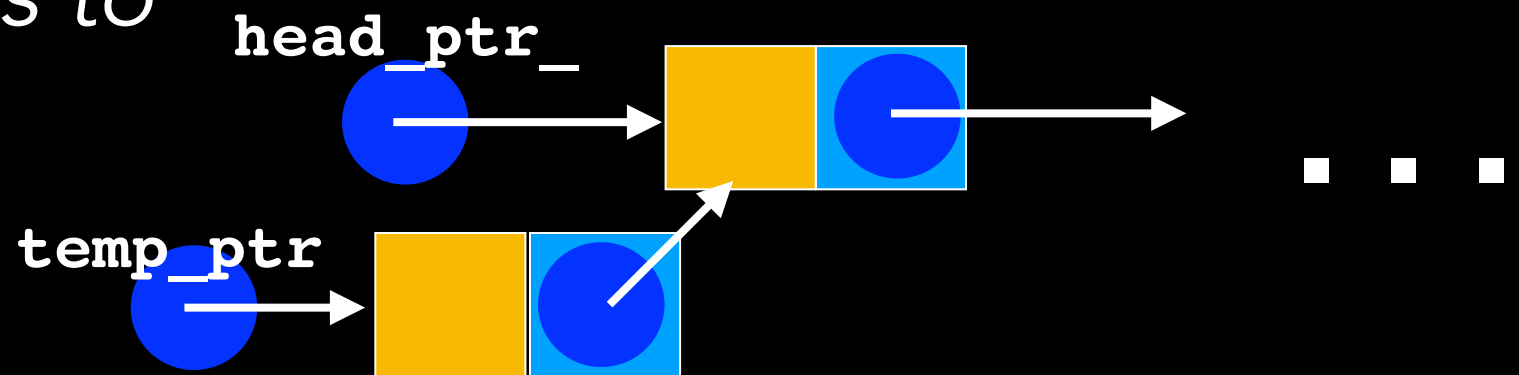
Let *head_ptr_* point to the *new node*



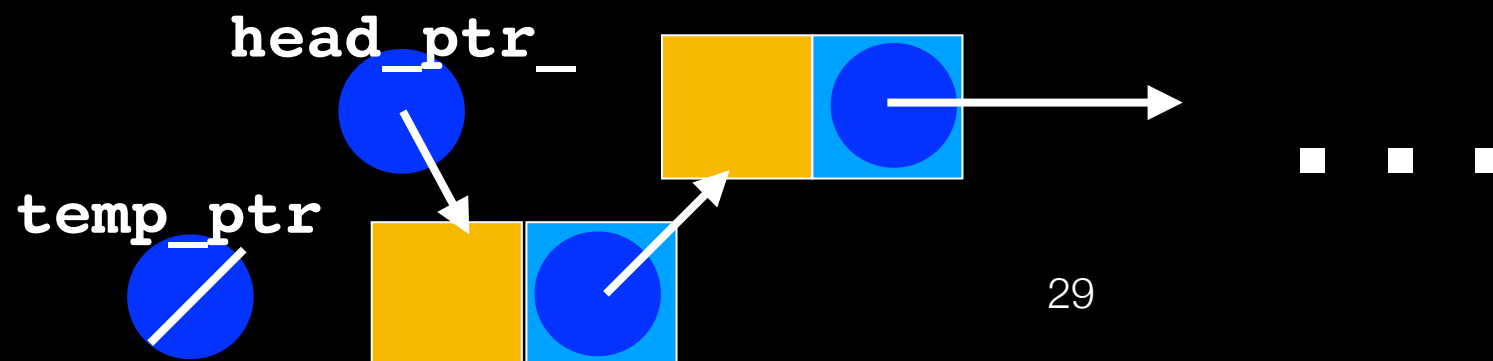
Instantiate a *new* node and let a *temp pointer* point to it



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"

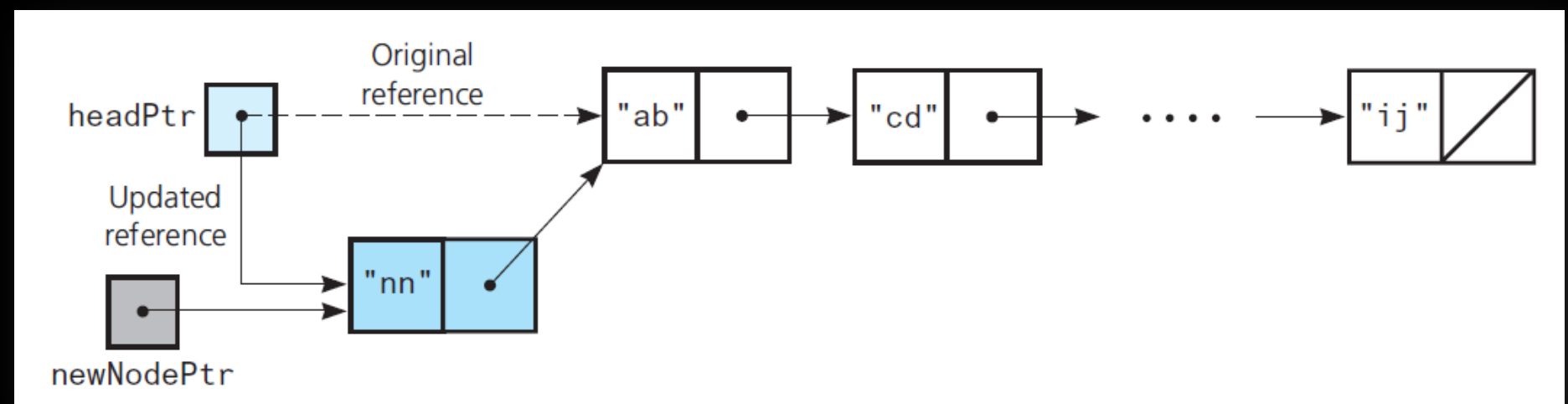
template<class T>
bool LinkedList<T>::add(const T& new_entry)
{
    // Add to beginning of chain: new node references rest of chain;
    // (head_ptr_ is null if chain is empty)
    Node<T>* new_node_ptr = new Node<T>;
    new_node_ptr->setItem(new_entry);
    new_node_ptr->setNext(head_ptr_); // New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



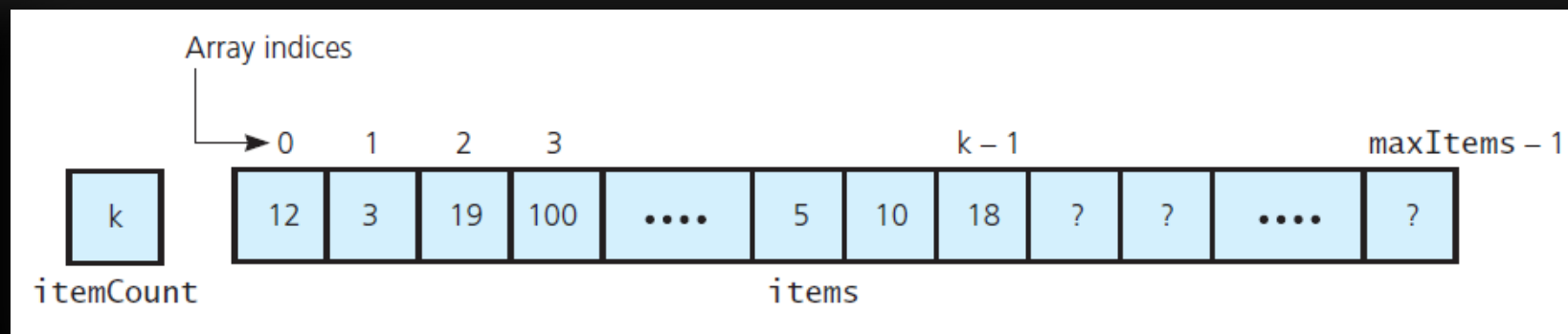
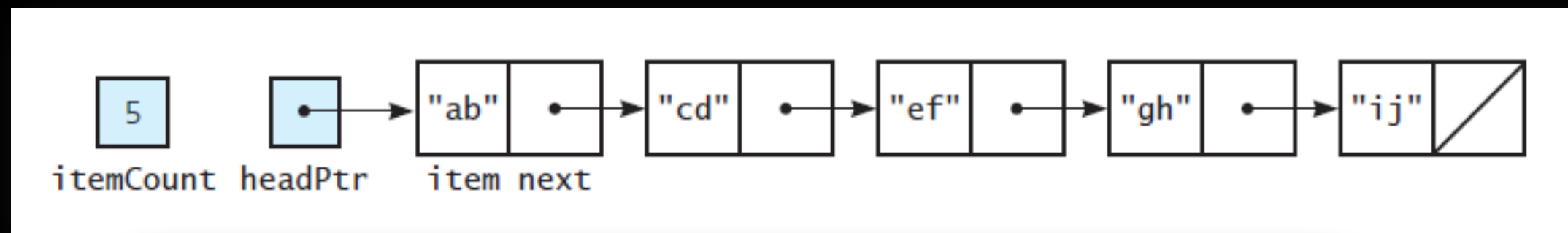
Efficiency

Create a new node and assign two pointers **$O(1)$**

What about adding to end of chain? **$O(n)$**

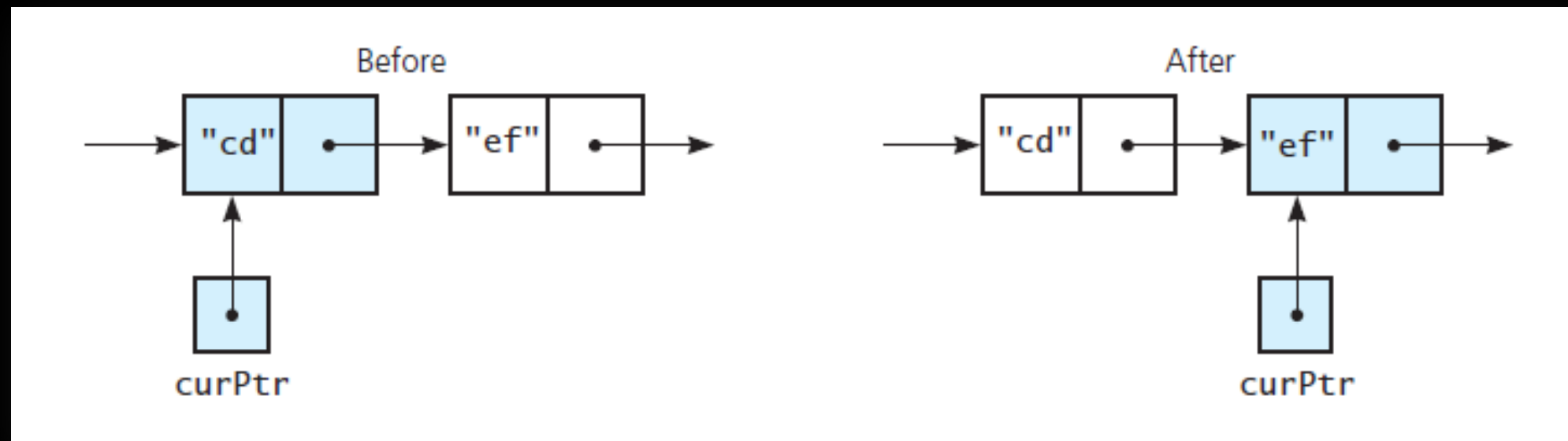
What about adding to front of array? **$O(1)$ or $O(n)$**

No order Order



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

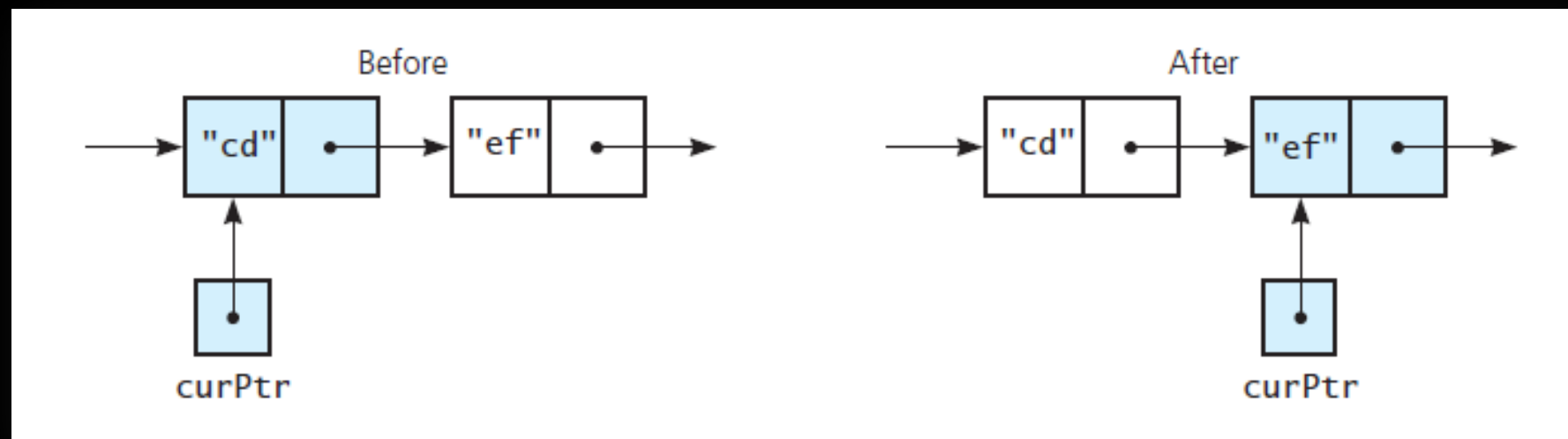
```
while(the current pointer is not the null pointer)
```

```
{
```

```
    "visit" the current node
```

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

```
}
```



LinkedBag Implementation

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

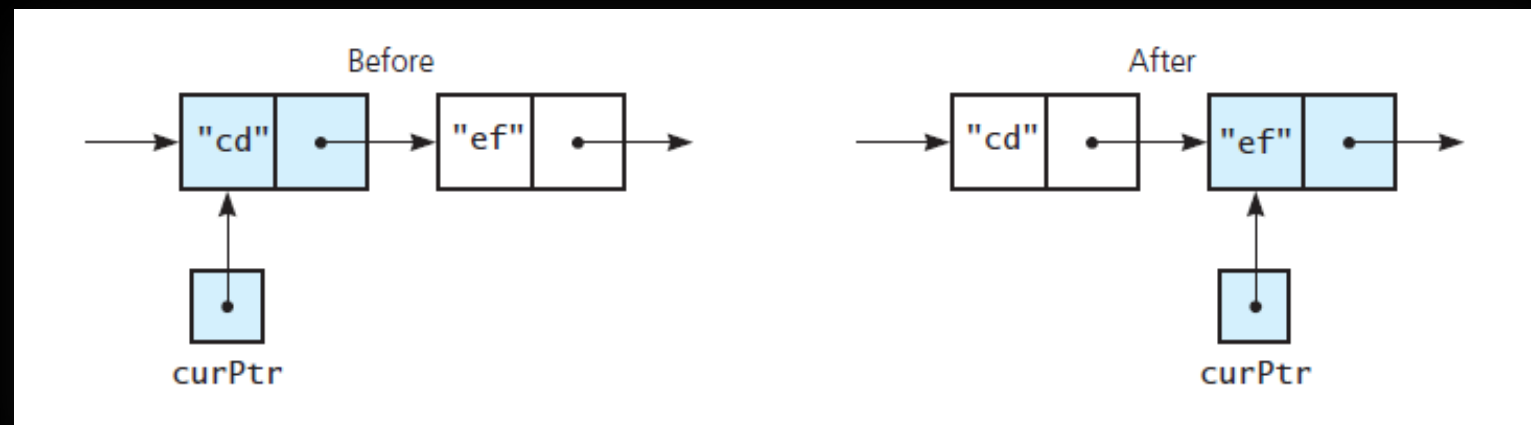
```
template<class T>
std::vector<T> LinkedBag<T>::toVector() const
{
    std::vector<T> bag_contents;
    Node<T>* cur_ptr = head_ptr_;

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

    return bag_contents;
} // end toVector
```

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<class T>
```

```
Node<T>* LinkedBag<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
```

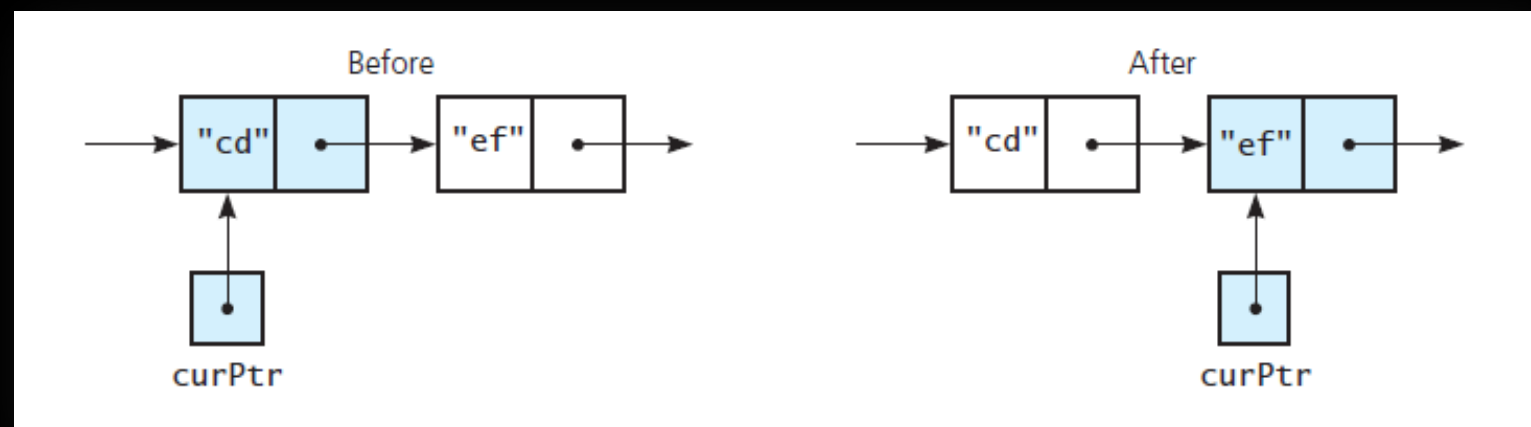
The getPointerTo
method

Traversing:

visit each node

if found what looking for

return



Efficiency

No fixed number of steps

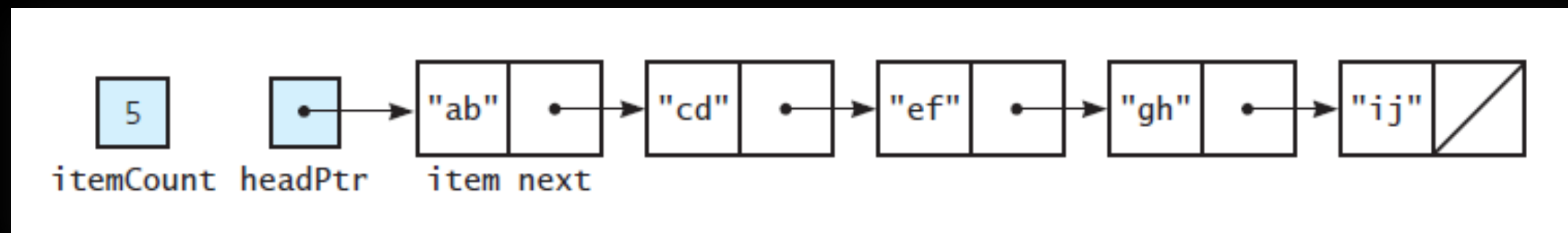
Depends on location of `an_entry`

- 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?

$O(n)$

Purposely vague
Some fixed amount of work

What should we do to remove?



LinkedList Implementation

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

O(1)

The remove method

```
template<class T>
bool LinkedList<T>::remove(const T& an_entry)
{
    Node<T>* entry_ptr = getPointerTo(an_entry);
    bool can_remove = (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

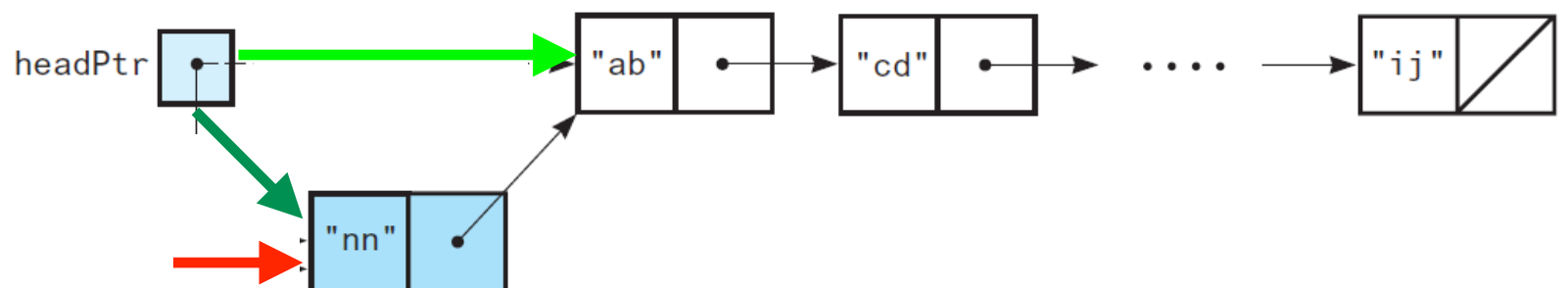
    return can_remove;
} // end remove
```

Find

Deleting first node is easy

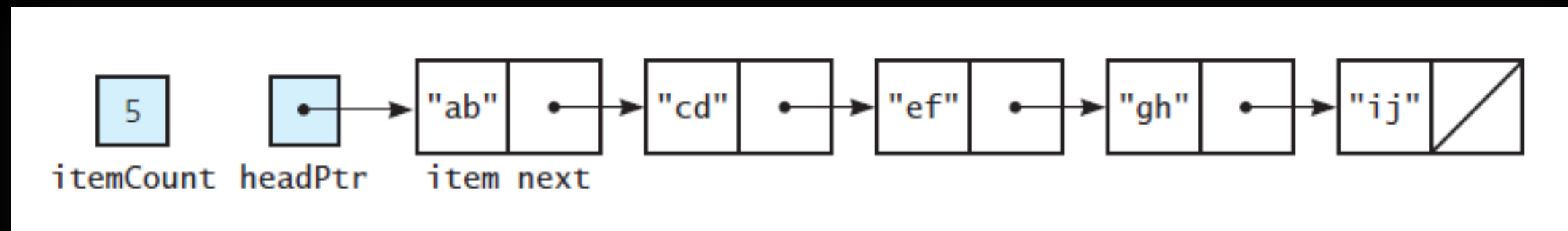
Copy data from first node to node to delete
Delete first node

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



How do we clear the bag?

Can we do the same thing we did with array?



LinkedList Implementation

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

```
template<class T>
void LinkedList<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

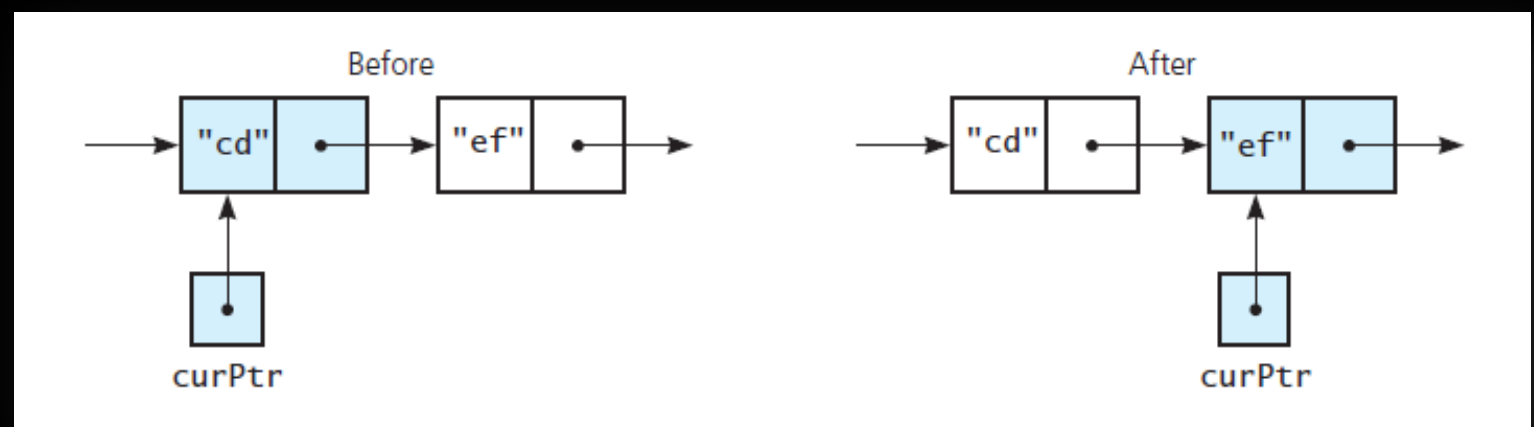
    item_count_ = 0;
} // end clear
```

O(n)

The clear method

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

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



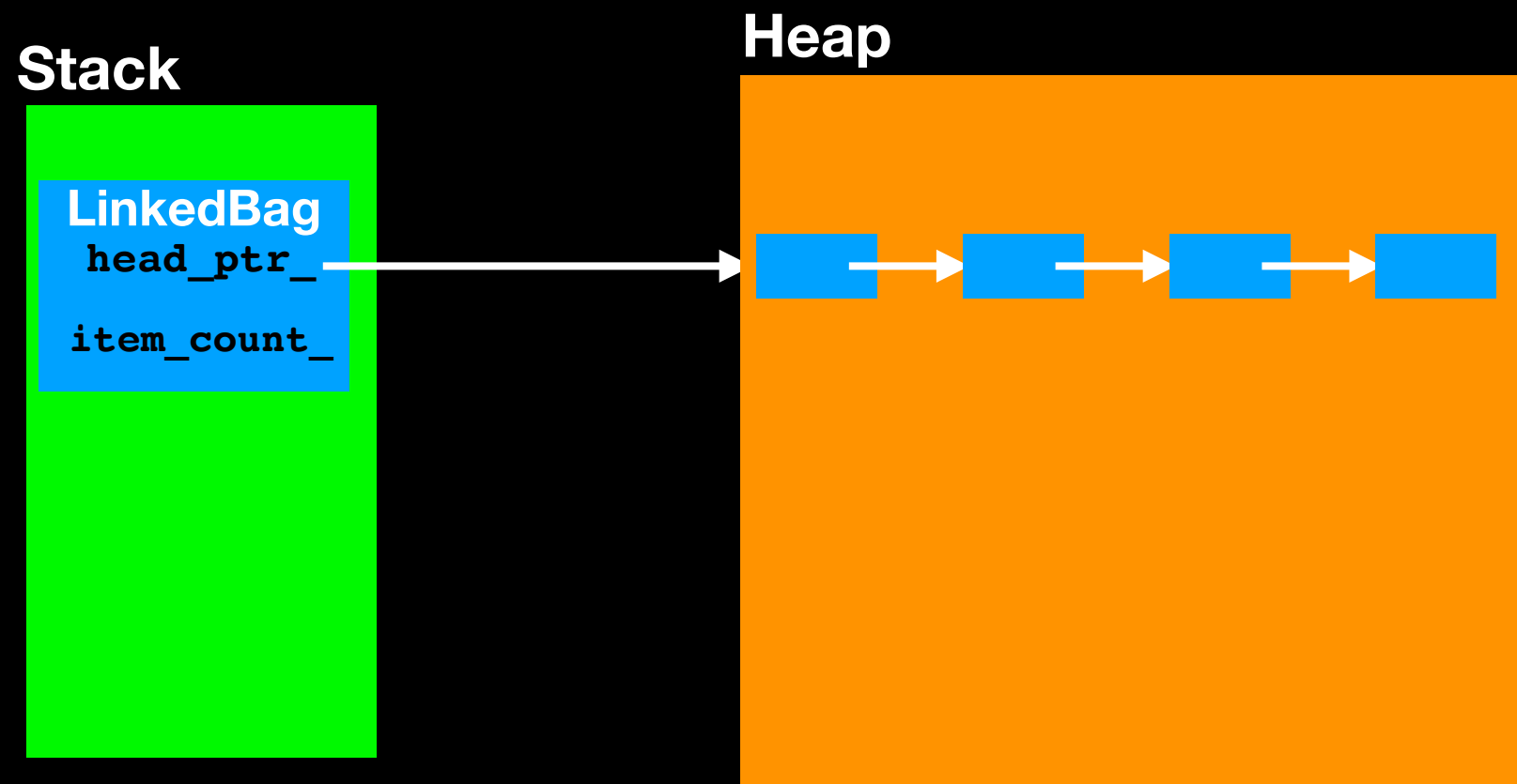
Dynamic Memory Considerations

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!

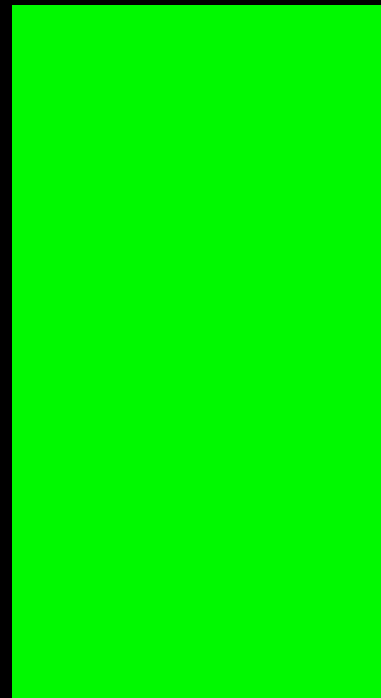
Avoid memory leaks!!!!

What happens when object goes out of scope?



What happens when object goes out of scope?

Stack



Heap



LinkedList Implementation

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

The destructor

```
template<class T>  
LinkedList<T>::~LinkedList()  
{  
  
    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<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_
```

$O(1)$



Efficient

$O(n)$



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_
```

O(1)



Efficient

O(n)



Expensive

Next time!