Data Structures and Advanced Programming

Fu-Yin Cherng

National Taiwan University

Array- and Link-based bags

Review

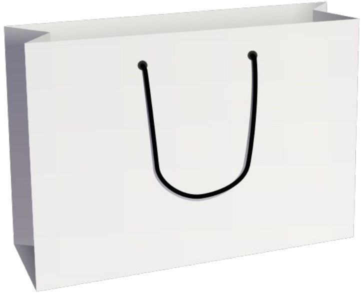
- Inheritance and Polymorphism
- Template and Exception handling

Outline

- Data Abstract Type “Bag”
- Two Ways to implement Bag
  - Array-based
  - Link-based (Linked List)

Data Abstract Type “Bag”

- paper bag, a reusable cloth bag
- container of a collection of objects.
- Bag can be an abstract data type
  - so let’s try to analyze, design, and implement it!



Data Abstract Type “Bag”

- object no particular order
- object may be duplicated
- all object in a bag is of the same type

Define Bag’s characteristics & behaviors

- Characteristic: member data
- Behaviors: member function

Identifying Behaviors - Accessor

- Get the number of items currently in the bag
- Detects if a bag is empty

Identifying Behaviors - Add/Remove

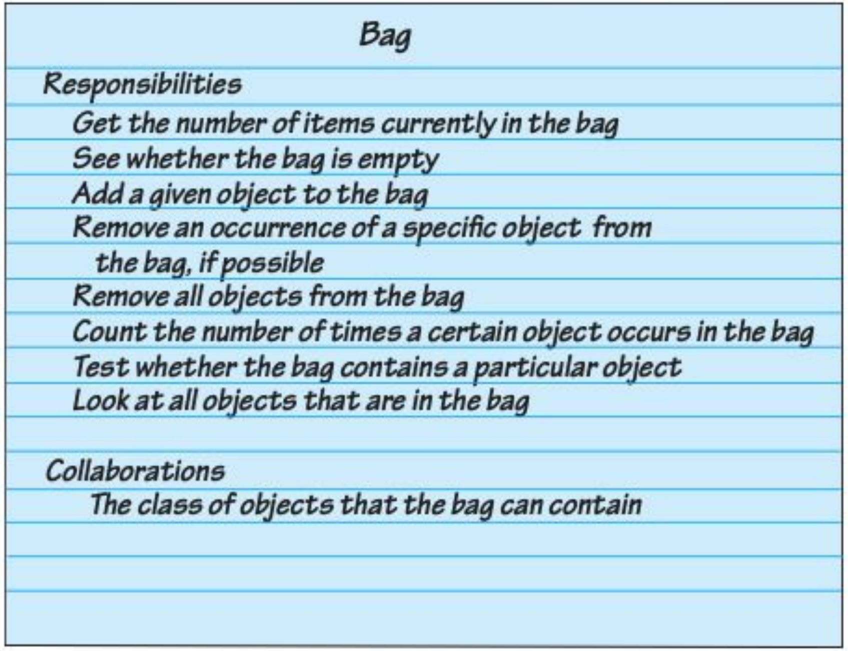
- Add a given object to the bag.
- Remove an occurrence of a specific object from the bag, if possible.
  - only remove the first one
- Remove all objects from the bag.

Identifying Behaviors - Count

- Count the number of times a certain object occurs in the bag.
- Test whether the bag contains a particular object.
- Look at all objects that are in the bag.

Record on a CRC card

- Class-responsibility-collaboration
- Good habit in programming design
- Help you have clear mind when defining function



Bag

Responsibilities

Get the number of items currently in the bag

See whether the bag is empty

Add a given object to the bag

Remove an occurrence of a specific object from the bag, if possible

Remove all objects from the bag

Count the number of times a certain object occurs in the bag

Test whether the bag contains a particular object

Look at all objects that are in the bag

Collaborations

The class of objects that the bag can contain

Specifying Data and Operations

- Before implement in C++, need to **describe** its data and specify in detail of behaviors
  - name the methods
  - choose their parameters
  - decide their return types
  - write comments to fully describe their effect on the bag’s data

Why do so many step??

- Process of object-oriented analysis and design
- After reading the problem specifications and going through the requisite amount of procrastination, most novice programmers simply begin to write code.
- Coding without a solution design increases debugging time

Pseudocode of Behaviors - getCurrentSize()

- Returns a count of the current number of entries in the bag
- No parameters and returns an integer

```
// Returns the current number of entries in the bag.
+getCurrentSize(): integer
```

Pseudocode of Behaviors - add()

- □ add and give bag a parameter to represent the new entry.
- □ why return boolean and use ItemType?

```
// Adds a new entry to the bag.
+add(newEntry: ItemType): boolean
```

Pseudocode of Behaviors - remove() & clear()

- remove particular entry
  - return boolean to indicate success or not
- remove all entries

```
+remove(anEntry: ItemType): boolean
+clear(): void
```

Pseudocode of Behaviors - Count related

- Counts the number of times a given object
- Test whether the bag contains a given object

```
// Counts number of times a given entry appears in the bag.
+getFrequencyOf(anEntry: ItemType): integer

// Tests whether the bag contains a given entry.
+contains(anEntry: ItemType): boolean
```

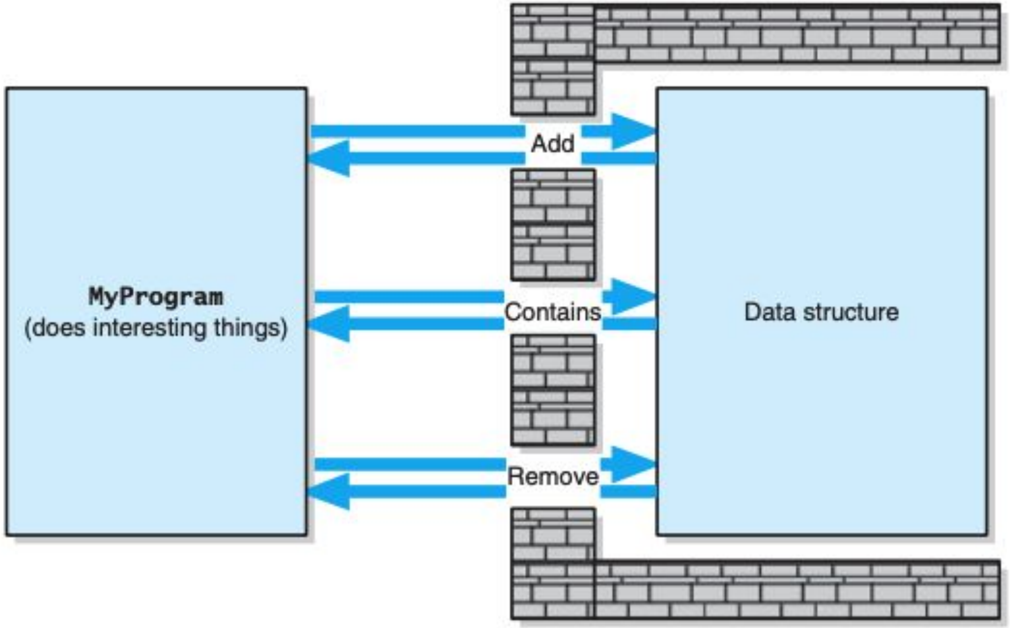
Summarize Pseudocode functions

- Table in p. 21 of Textbook
- Blueprint used during implementation
- Next, Design An Interface Template
  - write down these functions in C++ syntax, put them in a class `BagInterface`

Class BagInterface

```
template<class ItemType> class BagInterface
{
    public:
        virtual int getCurrentSize() const = 0;
        virtual bool isEmpty() const = 0;
        virtual bool add(const ItemType& newEntry) = 0;
        virtual bool remove(const ItemType& anEntry) = 0;
        virtual void clear() = 0;
        virtual int getFrequencyOf(const ItemType& anEntry)
    const = 0;
        virtual bool contains(const ItemType& anEntry) const =
    0;
};
```

ADT Interface As Walls



Using the ADT Bag

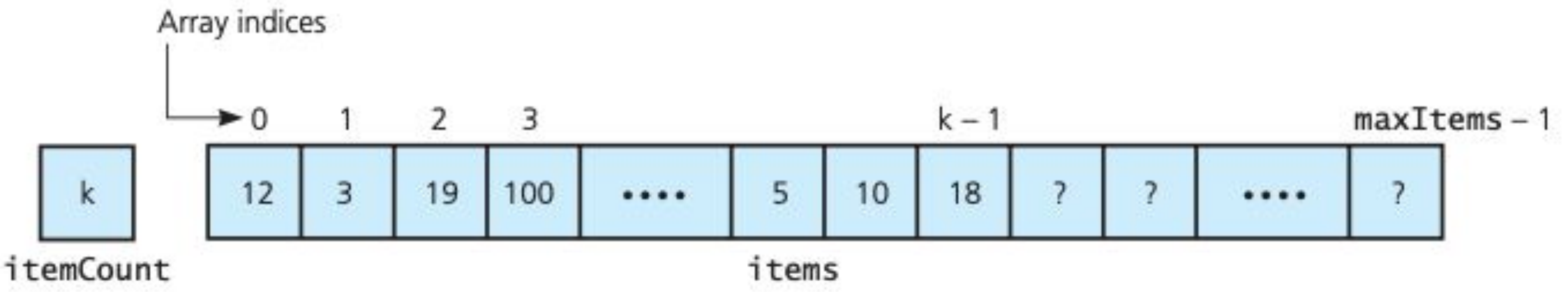
```
#include <iostream>
#include <string>
#include "Bag.h" // For ADT bag
using namespace std;
int main() {
        string clubs[] = { "Joker", "Ace", "Two", "Three",
    "Four", "Five", "Six", "Seven","Eight", "Nine", "Ten",
    "Jack", "Queen", "King" };

Bag<string> grabBag;
grabBag.add(clubs[1]);
grabBag.add(clubs[2]);
grabBag.add(clubs[4]);
…};
```

Bag Implementation

- **Array-based Implementation**
- Link-based Implementation

Fixed-size array Implementation

- each item occupies one entry of an array
- define maxItems for the array size
- use indices to access items



Class ArrayBag

```
template<class ItemType>
class ArrayBag : public BagInterface<ItemType> {
    private:
        static const int DEFAULT_CAPACITY = 6;
        ItemType items[DEFAULT_CAPACITY];
        int itemCount;
        int maxItems;
    public:
        //...
};
```

Class ArrayBag - member functions

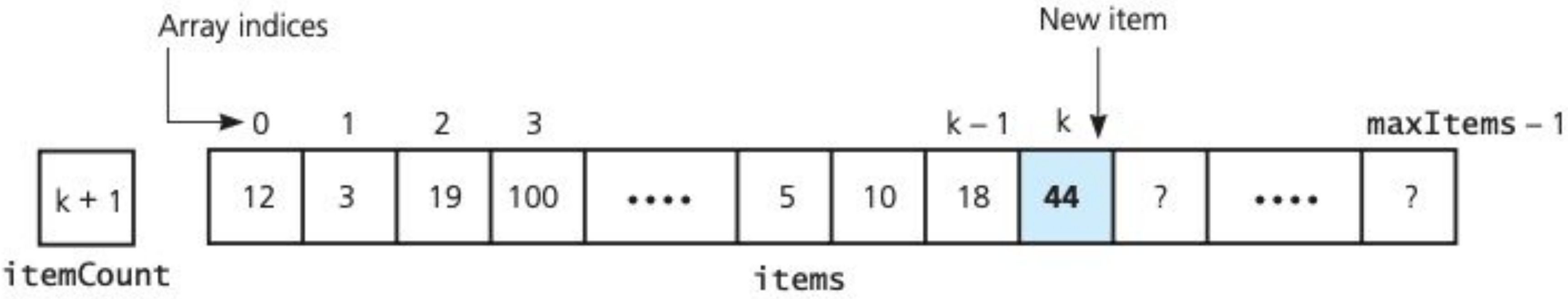
```
template<class ItemType>
class ArrayBag : public BagInterface<ItemType> {
    private:
        //...
    public:
        ArrayBag();
        int getCurrentSize() const;
        bool isEmpty() const;
        bool add(const ItemType& newEntry);
        bool remove(const ItemType& anEntry);
        void clear();
        bool contains(const ItemType& anEntry) const;
        int getFrequencyOf(const ItemType& anEntry) const;
};
```

Implement member functions

- do not cover all function implementations
- focus on those related to add() and remove()
  - change the content of Bag
  - important difference between array-based and link-based implementations

add()

- if there is room to store the item in the array
- return true if there is a room or false otherwise



add()

```
template<class ItemType>
bool ArrayBag<ItemType>::add(const ItemType& newEntry)
{
    bool hasRoomToAdd = (itemCount < maxItems);
    if (hasRoomToAdd){
        items[itemCount] = newEntry;
        itemCount++;
    } // end if
    return hasRoomToAdd;
} // end add
```

remove()

- remove a given **entry**
- return true if the item exists and removal was successful, or false otherwise
- keep the array “no gap”
- need to check if the entry **exists** & **where** is the entry
  - implement function getIndexOf() first

getIndexOf()

- Given an item, return the index of its first copy in the array or −1 otherwise.
- pseudocode:

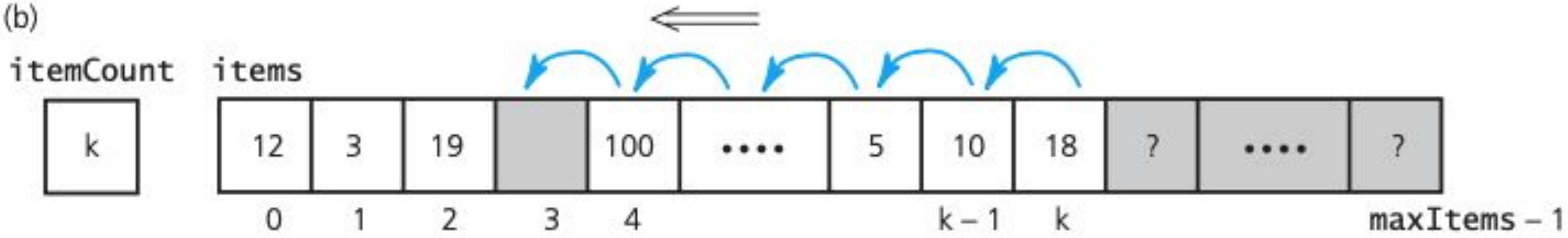
```
getIndexOf(anEntry:ItemType):integer
```

getIndexOf()

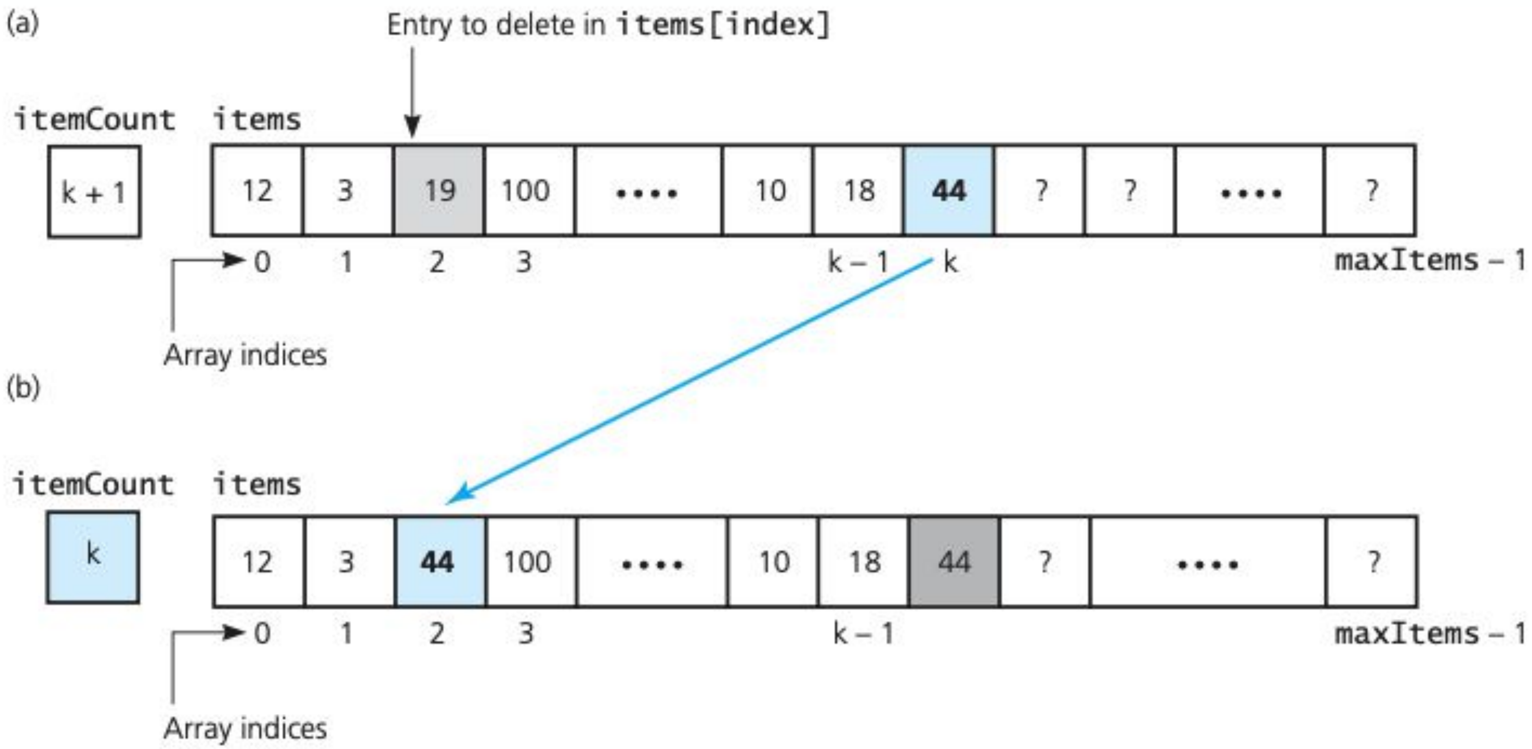
```
template<class ItemType>
int ArrayBag<ItemType>::getIndexOf(const ItemType& target) const {
    bool found = false;
    int result = -1;
    int searchIndex = 0;
    while (!found && (searchIndex < itemCount)) {
        if (items[searchIndex] == target) {
            found = true;
            result = searchIndex; }
        else{
            searchIndex++;} // end if
    } // end while
    return result;
}
```

remove() - “no gap” array

- □ keep the array “no gap”
- □ How?



remove() - “no gap” array



remove()

```
template<class ItemType>
bool ArrayBag<ItemType>::remove(const ItemType& anEntry) {

    int locatedIndex = getIndexOf(anEntry);
    bool canRemoveItem = !isEmpty() && (locatedIndex > -1);
    if (canRemoveItem){
        itemCount--;
        items[locatedIndex] = items[itemCount];
    } // end if

    return canRemoveItem;
} // end remove
```

getIndexOf() private or public?

- Useful to client
- Important reasons why should be **private**.
  - array to store entries items is private
  - client should access the entries in array only through "the wall" (ADT Interface we design)

Testing during implementations

- Do not wait until you complete the implementation of an ADT
- Stubs
  - An incomplete definition of a method is called a stub.

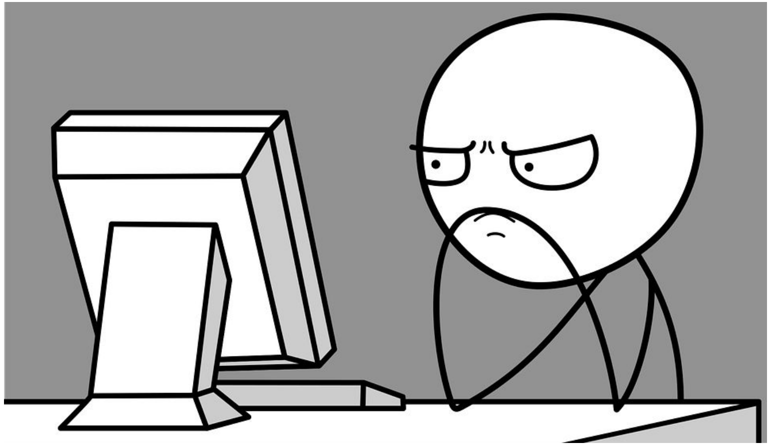
```
template<class ItemType>
bool ArrayBag<ItemType>::remove(const ItemType& anEntry) {
    return false; // STUB; return dummy value
} // end remove
```

Dynamic Array Bag

- from fixed-size to dynamic array
- how to modify to make the array dynamic?
- see the video of Prof. Ling-Chieh Kung
  - Dynamic-array bag

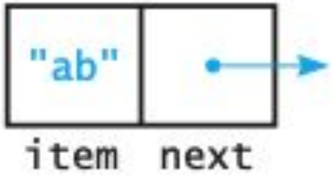
Bag Implementation

- Array-based Implementation
- **Link-based Implementation**

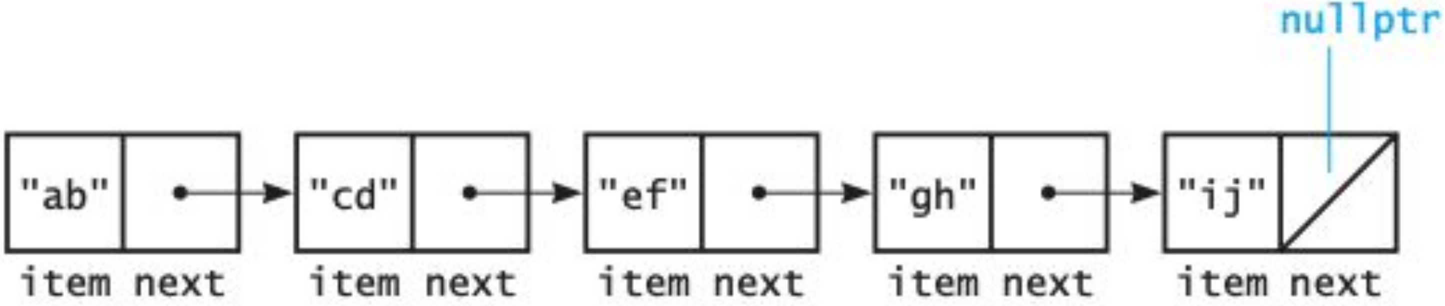


What’s link-based data structure

A Node:

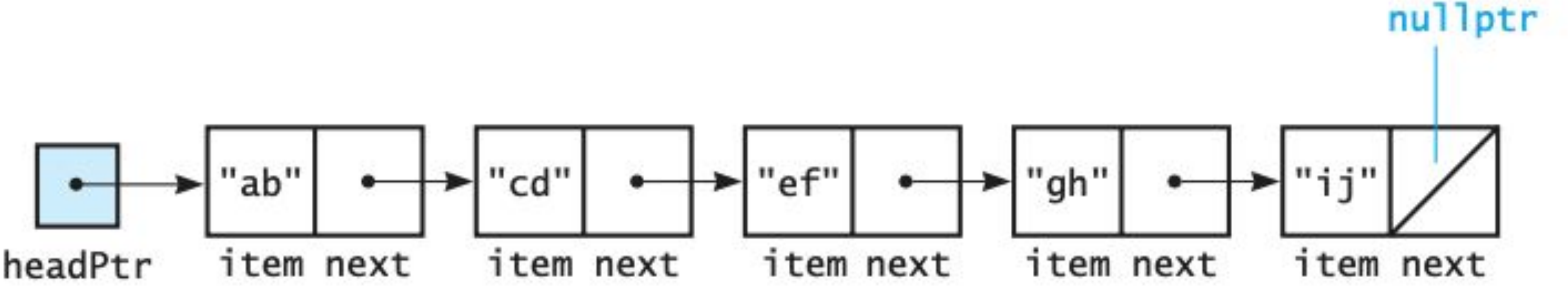


- contain two pieces of information
    - item and pointer (next)
- each **node** should be an object
- Linked List



Header pointer

- use headPtr to access the first node
- headPtr is not a node, is a simple pointer



Link-based Node

```
template<typename ItemType>
class Node{
    private:
        ItemType item; // A data item
        Node<ItemType>* next; // Pointer to next node
    public:
        Node();
        Node(const ItemType& anItem);
        Node(const ItemType& anItem, Node<ItemType>* nextNodePtr);
        void setItem(const ItemType& anItem);
        void setNext(Node<ItemType>* nextNodePtr);
        ItemType getItem() const ;
        Node<ItemType>* getNext() const ;
}; // end Node
```

Node Constructor

```
template<class ItemType> Node<ItemType>::Node() : next(nullptr) {
} // default constructor

template<class ItemType>
Node<ItemType>::Node(const ItemType& anItem) : item(anItem), next(nullptr)
{
} //initial Item constructor

template<class ItemType>
Node<ItemType>::Node(const ItemType& anItem, Node<ItemType>* nextNodePtr)
:
item(anItem), next(nextNodePtr) }
//initial Item and next pointer constructor
```

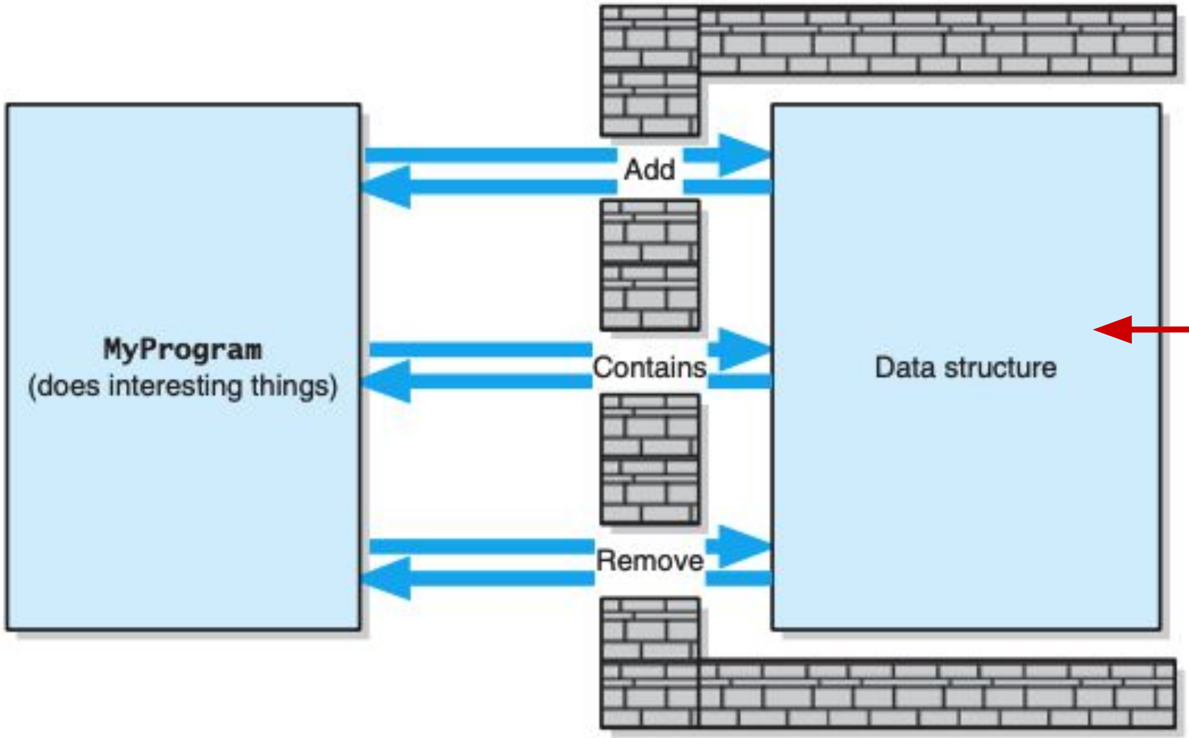
Set functions

```
template<class ItemType>
void Node<ItemType>::setItem(const ItemType& anItem) {
    item = anItem;
} // end setItem

template<class ItemType>
void Node<ItemType>::setNext(Node<ItemType>* nextNodePtr) {
    next = nextNodePtr;
} // end setNext
```

Get functions

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

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

Link-Based Bag Implementation



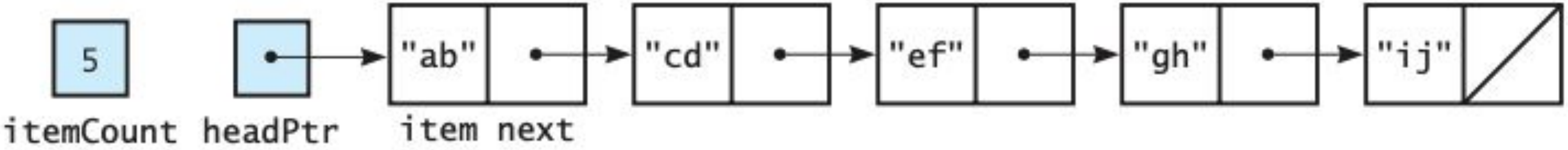
array

or

linked list

Link-Based Bag Implementation

- itemCount to track node number in a bag
- Node<string>



Class LinkedBag

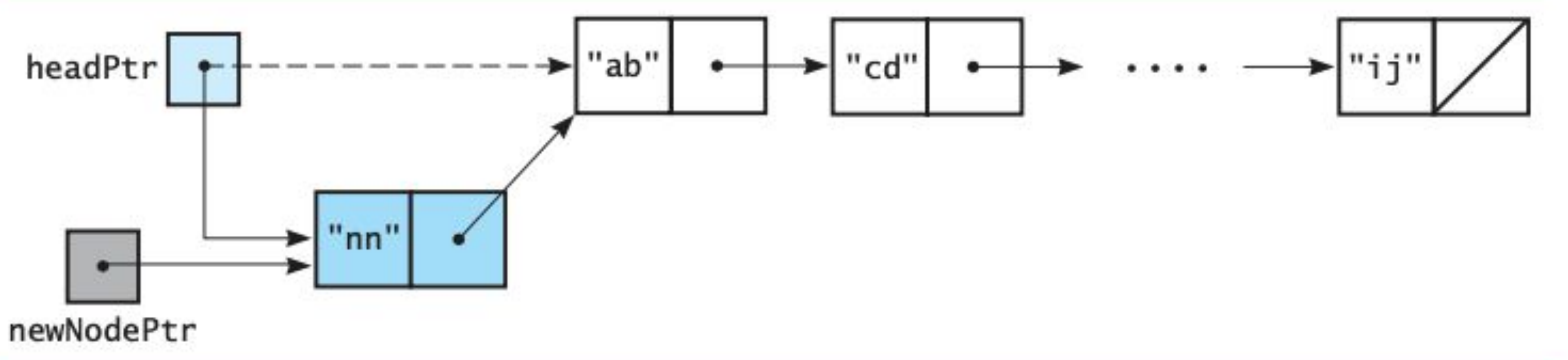
```
template<typename ItemType>
class LinkedBag : public BagInterface<ItemType>{
    private: // member data
        Node<ItemType>* headPtr; // Pointer to first node
        int itemCount; // Current count of bag items
    public:
        LinkedBag();
        LinkedBag(const LinkedBag<ItemType>& aBag); // Copy constructor
        virtual ~LinkedBag(); // Destructor should be virtual

        int getCurrentSize() const;
        bool isEmpty() const;
        //… the same member functions in BagInterface & ArrayBag
};
```

Default constructor

```
template<class ItemType>
LinkedBag<ItemType>::LinkedBag() : headPtr(nullptr),
itemCount(0) {
} // end default constructor
```

Add()

- insert a new item (node) at any **convenient** location in linked list
- insert into the beginning of the list is the most convenient location
- no need to pass node by node



Add()

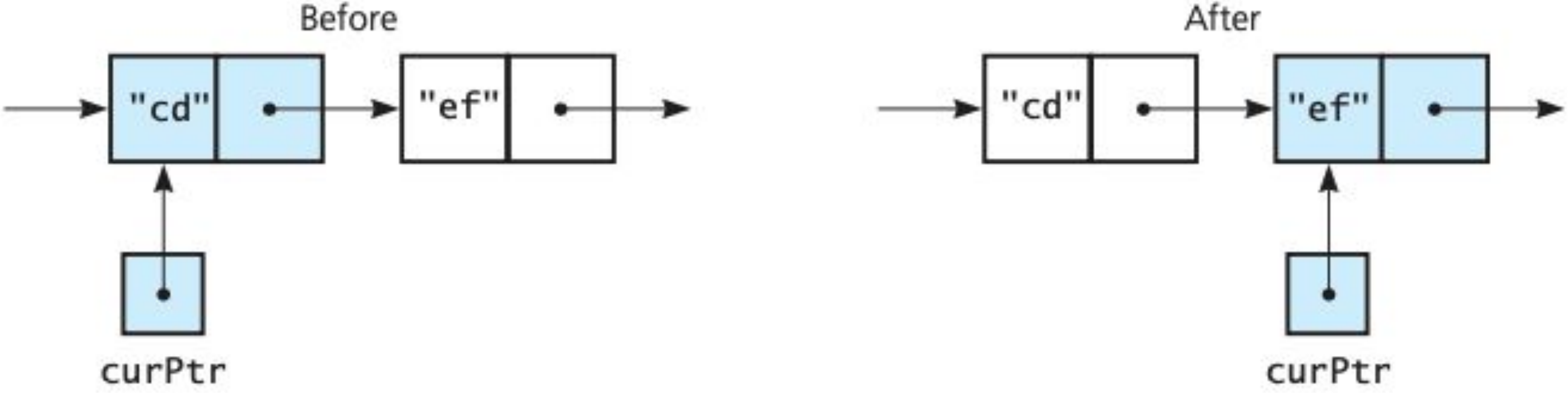
```
template<class ItemType>
bool LinkedBag<ItemType>::add(const ItemType& newEntry) {

    // Add to beginning of chain: new node references rest of chain;
    // (headPtr is nullptr if chain is empty)
    Node<ItemType>* newNodePtr = new Node<ItemType>();
    newNodePtr->setItem(newEntry);
    newNodePtr->setNext(headPtr); // New node points to chain
    headPtr = newNodePtr; // New node is now first node
    itemCount++;

    return true;
} // end add
```

toVector()

- retrieves the entries that are in a bag and returns them to the client within a vector.
- in array-based version,

```
template<class ItemType>
vector<ItemType> ArrayBag<ItemType>:: toVector() const
{
    vector<ItemType> bagContent;
    for (int i = 0; i < itemCount; i++){
        bagContents.push_back(items[i]); //add to vector
    }
    return bagContents;
}
```

toVector()



toVector()

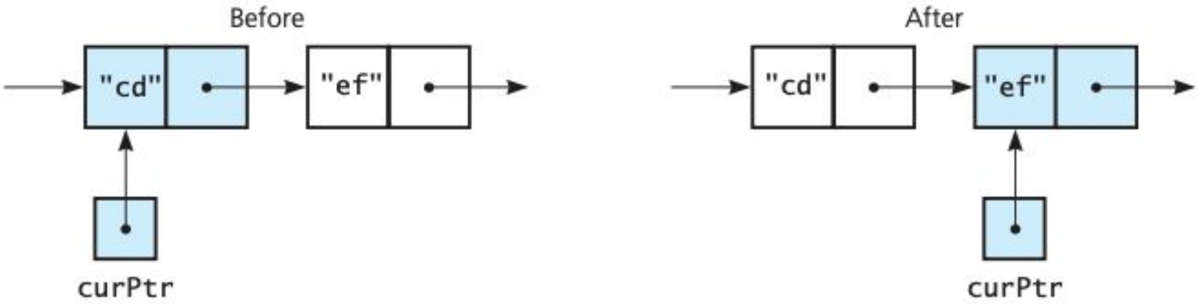
```
template<class ItemType>
vector<ItemType> LinkedBag<ItemType>::toVector() const {
    vector<ItemType> bagContents;
    Node<ItemType>* curPtr = headPtr;
    int counter = 0;
    while ((curPtr != nullptr) && (counter < itemCount)){
        bagContents.push_back(curPtr->getItem());
        curPtr = curPtr->getNext();
        counter++;
    } // end while
    return bagContents;
} // end toVector
```

remove()

- deletes one occurrence (the first copy) of a given entry
- returns either true or false to indicate whether the removal was successful
- **traverse** node to find the node with given entry
  - why we introduce toVector() first
- need to know where is to node: getPointerTo()

getPointerTo()

```
template<class ItemType>
Node<ItemType>* LinkedBag<ItemType>::getPointerTo(const ItemType& target) const{
     bool found = false;
     Node<ItemType>* curPtr = headPtr;
     while (!found && (curPtr != nullptr)) {
          if (target == curPtr->getItem()){ found = true;}
          else{curPtr = curPtr->getNext(); } // end while
     }
     return curPtr;
}
```



remove()

- pseudocode

```
remove(anEntry)
Find the node that contains anEntry // use getPointerTo() here
Replace anEntry with the entry that is in the first node
Delete the first node
```

remove()

```
template<class ItemType>
bool LinkedBag<ItemType>::remove(const ItemType& anEntry) {
    Node<ItemType>* entryNodePtr = getPointerTo(anEntry);
    bool canRemoveItem = !isEmpty() && (entryNodePtr != nullptr);
    if (canRemoveItem){
        entryNodePtr->setItem(headPtr->getItem());// Copy data from first node
        Node<ItemType>* nodeToDeletePtr = headPtr; // Delete first node
        headPtr = headPtr->getNext();
        nodeToDeletePtr->setNext(nullptr); // Return node to the system
        delete nodeToDeletePtr; //return memory
        nodeToDeletePtr = nullptr;
        itemCount--;
    }
    return canRemoveItem;
}
```

clear()

- linked list was allocated dynamically, so must `delete` them in clear()

```
template<class ItemType>
void LinkedBag<ItemType>::clear(){
    while (headPtr != nullptr) {

        Node<ItemType>* nodeToDeletePtr = headPtr;
        headPtr = headPtr->getNext();

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

    } // end while

    // headPtr is nullptr
    nodeToDeletePtr = nullptr;
    itemCount = 0;
}// end clear
```

Destructor

- if **statically** allocated memory (ArrayBag), can depend on **compiler-generated** destructor
- when uses **dynamically** allocated memory (LinkedBag), need to write a destructor that **deallocates** this memory by using delete.

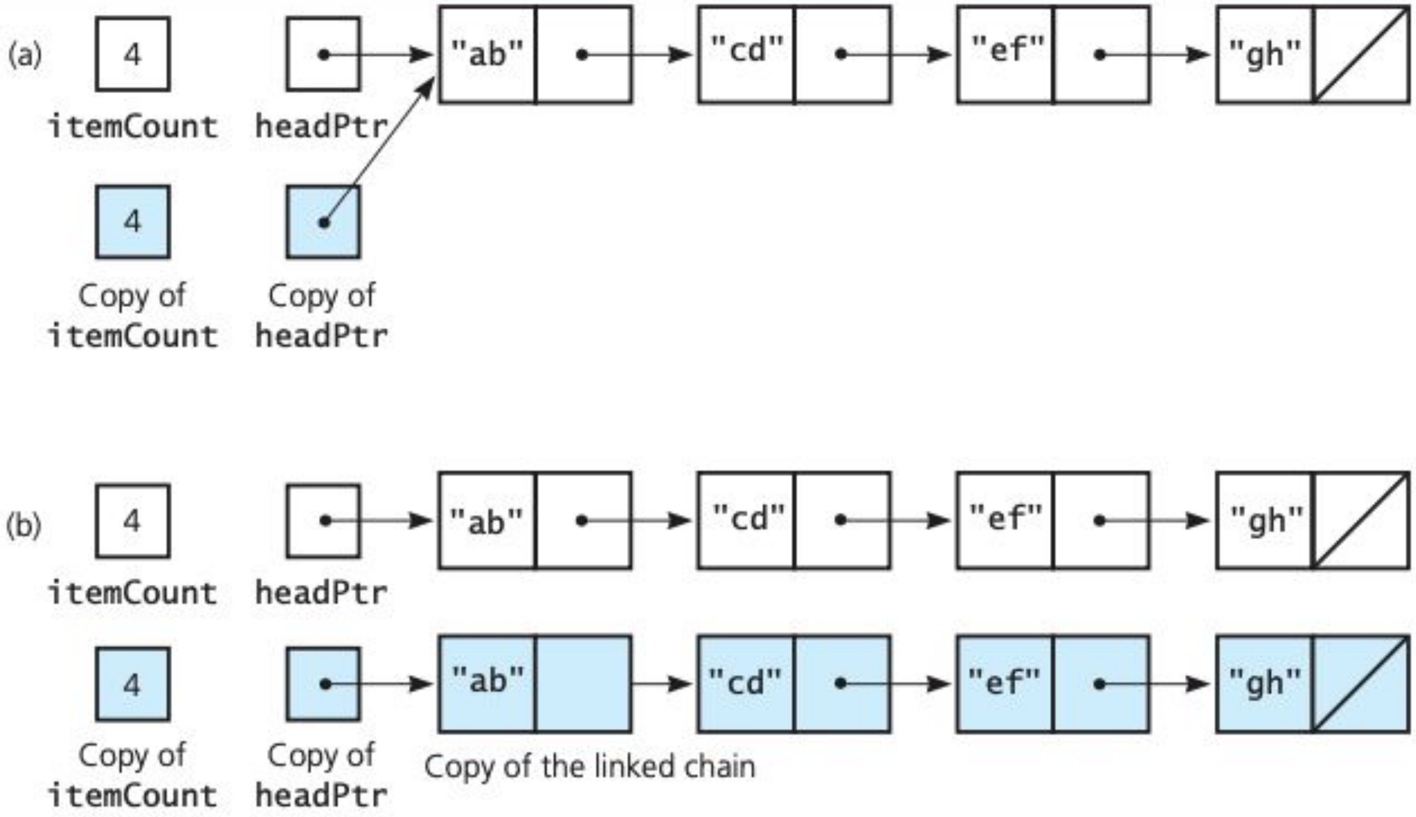
```
template<class ItemType> LinkedBag<ItemType>::~LinkedBag() {
    clear();
} // end destructor
```

Copy Constructor

- □ because of dynamically allocated memory, implement “deep” copy in copy constructor
- □ copy bag1 to bag2

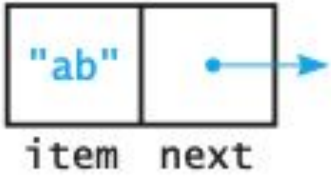
```
LinkedBag(const LinkedBag<ItemType>& aBag);
LinkedBag bag2(bag1);
```

Copy Constructor



Summary

- Introduce ADT “Bag”
  - BagInterface
- Array-based Implementation
  - add(), remove()…
- Link-based Implementation
  - Class node, add(), remove()…
- Next week
  - Compare Array-Based and Link-Based
  - Use two kinds of bag together
  - Recursion and Algorithm efficiency


"ab"
item
next