

# Data Structures

## Link-Based Implementations

Ling-Chieh Kung

Department of Information Management  
National Taiwan University

# Outline

- **Nodes.**
- Implementations of member functions.
- Constructors and destructor.
- Remarks.

# Link-based implementations

- Last time we implemented the ADT bag with arrays.
- There are some drawbacks:
  - Static arrays: The bag size is fixed.
  - Dynamic arrays: Doubling the bag size requires a lot of copy and paste.
- Here we will use pointers to do a **list-based implementation**.
  - Bag items will be put on a “**list**.”
  - On the list, each item is “**linked**” to the next item.

# Nodes

- The type of an item is `ItemType`.

```
template<typename ItemType>
class BagInterface
{
public:
    virtual bool
        add(const ItemType& newEntry) = 0;
    // all other functions
};
```

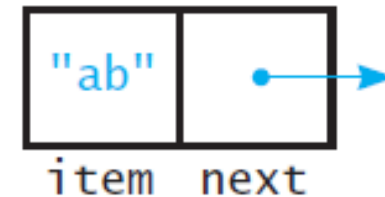


Figure 4-1 (Carrano and Henry, 2015)

- For each item, we will associate a **pointer** pointing to the **next** item-pointer pair.
- The combination of the item and pointer is often called a **node**.
  - The pointer **points to the next node**, not the next item!

# List of nodes

- In effect, a **list of nodes** will be created to store items.

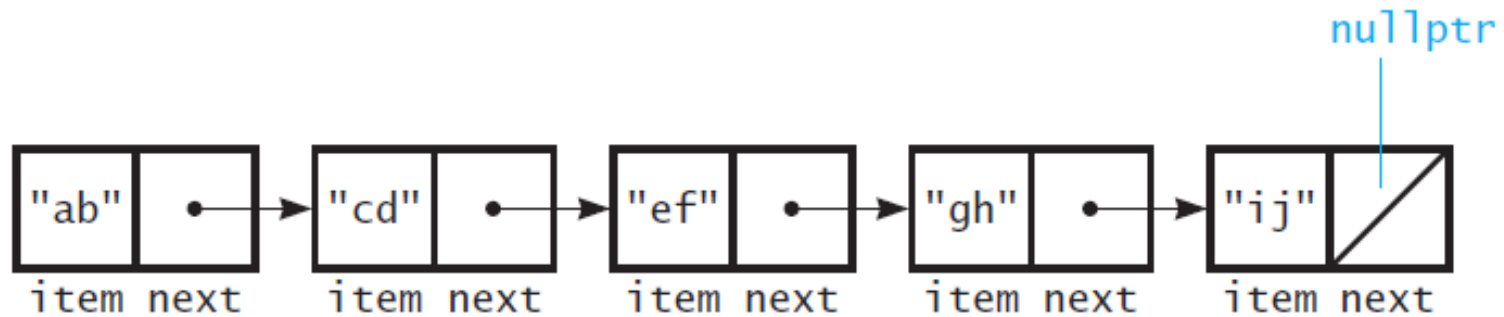


Figure 4-2 (Carrano and Henry, 2015)

- Each node is linked to the next node.
  - The pointer **next** points to the next node.

# The class Node

```
template<class ItemType>
class Node
{
private:
    ItemType item;
    Node<ItemType>* next;
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;
};
```

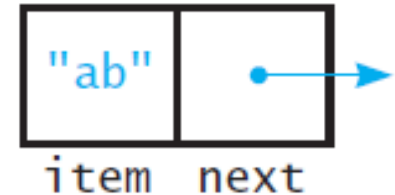


Figure 4-1  
(Carrano and  
Henry, 2015)

# The class Node

- Constructors of **Node**:

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

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

template<class ItemType>
Node<ItemType>::Node(const ItemType& anItem, Node<ItemType>* nextNodePtr)
    : item(anItem), next(nextNodePtr) {}
```

# The class Node

- Getters and setters of **Node**:

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

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

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

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



# The class **Node**

- The class **Node** has **getters** and **setters** for all instance variables.
  - It does **no data hiding** at all.
- Sometimes people implement **Node** as a **structure**.
  - Member variables of a structure are by default public.
  - Though they can be set to be private.
- Sometimes a better way is to set **LinkedList** a **friend** of **Node**.

# Outline

- Nodes.
- **Implementations of member functions.**
- Constructors and destructor.
- Remarks.

# The class `LinkedListBag`

- What should be contained in the class `LinkedListBag`?
- Let's look at `ArrayBag` again:
  - `itemCount` is still needed.
  - Capacity-related members are not needed.
- Where to store our nodes?
  - The 1<sup>st</sup> node links to the 2<sup>nd</sup>.
  - The  $i^{\text{th}}$  node links to the  $(i + 1)^{\text{th}}$ .
- All we need is to store the 1<sup>st</sup> node, i.e., the `head`.

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

# The head pointer headPtr

- An object of **LinkedList** will have an **item counter** `itemCount`.
  - Its type is **int**.
- An object of **LinkedList** will have a **head pointer** `headPtr`.
  - Its type is **Node<itemType>\***. It is a pointer, **not a node**.
  - **headPtr** is a **static member**; all nodes are “dynamic members.”

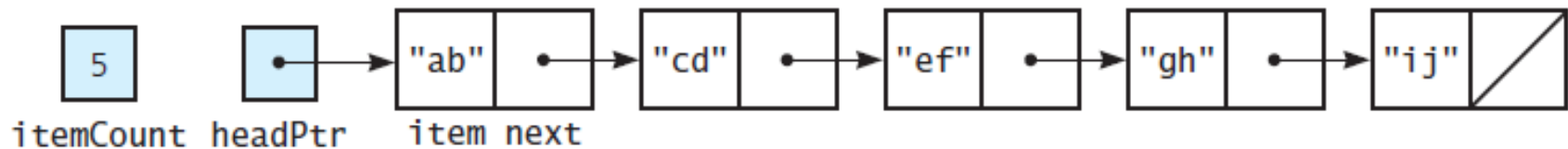


Figure 4-5 (Carrano and Henry, 2015)

# The class `LinkedBag`

- The basic elements of `LinkedBag`:

```
template<class ItemType>
class LinkedBag : public BagInterface<ItemType>
{
private:
    Node<ItemType>* headPtr;
    int itemCount;
    // something else
public:
    // some constructors and destructor
    // functions defined in BagInterface, such as getCurrentSize(),
    // isEmpty(), add(), remove(), etc.
};
```

- Let's implement those functions first.

# isEmpty() and getCurrentSize()

- Some member functions are straightforward.

```
template<class ItemType>
bool LinkedBag<ItemType>::isEmpty() const
{
    return itemCount == 0;
}

template<class ItemType>
int LinkedBag<ItemType>::getCurrentSize() const
{
    return itemCount;
}
```

# add ()

- We define **add ()** to add an item at the **beginning** of the list.
  - Why?
  - What if we add it at the end of the list?

```

template<class ItemType>
bool LinkedBag<ItemType>::add(const ItemType& newEntry)
{
    Node<ItemType>* newNodePtr = new Node<ItemType>();
    newNodePtr->setItem(newEntry);
    newNodePtr->setNext(headPtr);
    headPtr = newNodePtr;
    itemCount++;
    return true;
}

```

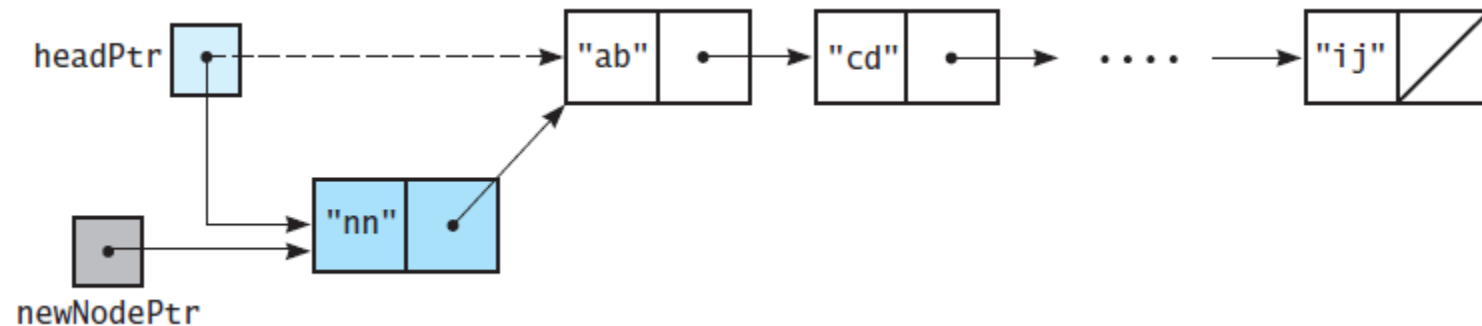


Figure 4-6 (Carrano and Henry, 2015)

# toVector () and list traversal

- We still want to put all items into a vector. We need to **traverse** the list.
  - With **headPtr**, we have the address of the first node.
  - With **headPtr->getNext ()**, we have the address of the second node.
  - How to proceed?
- We need a **location pointer curPtr** to store a node address in each iteration.

Let **curPtr** point to the first node in the list

*while* **curPtr** is not a null pointer

```
{  
    Assign the data portion of the current node to the next element in a vector  
    Set curPtr to the next pointer portion of the current node  
}
```



# toVector () and list traversal

```
template<class ItemType>
vector<ItemType> LinkedBag<ItemType>::toVector() const
{
    vector<ItemType> bagContents;
    Node<ItemType>* curPtr = headPtr;
    while (curPtr != nullptr)
    {
        bagContents.push_back (curPtr->getItem());
        curPtr = curPtr->getNext();
        // the above line is equivalent to:
        // Node<ItemType>* temp = curPtr->getNext();
        // curPtr = temp
    }
    return bagContents;
}
```

```
curPtr = curPtr->getNext()
```

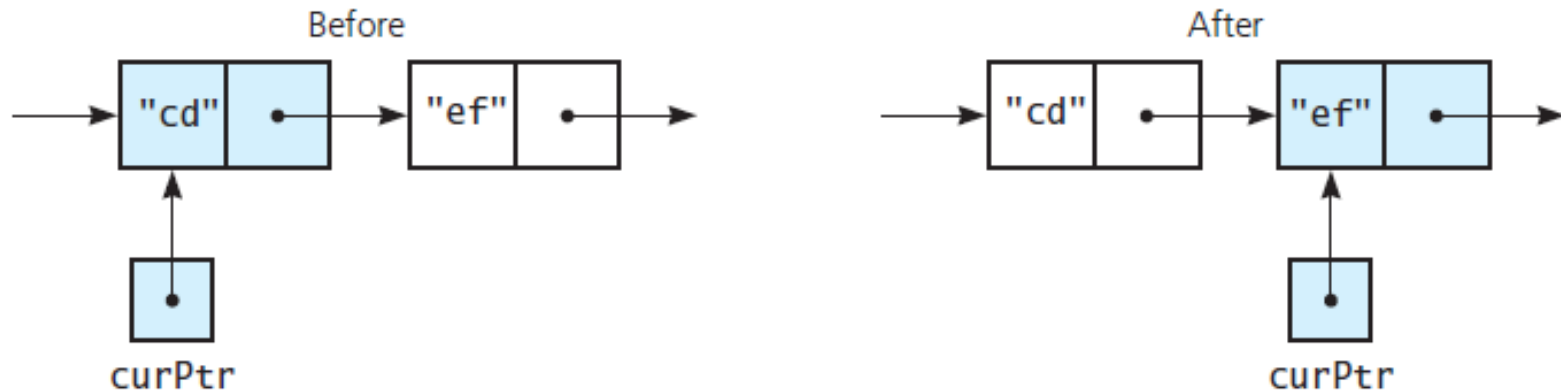


Figure 4-7 (Carrano and Henry, 2015)

# remove ()

- To remove an item, we will remove **the first copy** (the one closest to the head).
  - First we **locate** that copy (if there is one).
  - Then we **rewrite** the data portion of that node as that of the first node.
  - Finally, we **redirect headPtr** to the second node and **release** the first node.

```
Node<ItemType>* nodeToDeletePtr = headPtr;  
headPtr = headPtr->getNext(); // redirect  
delete nodeToDeletePtr; // release
```

- We will create a private function **getPointerTo ()** to do the locating task.
  - Recall that for **ArrayBag**, we have **getIndexOf ()**.

# remove ()

```
template<class ItemType>
bool LinkedBag<ItemType>::remove(const ItemType& anEntry)
{
    Node<ItemType>* entryNodePtr = getPointerTo(anEntry); // step 1
    bool canRemoveItem = !isEmpty() && (entryNodePtr != nullptr);
    if(canRemoveItem)
    {
        entryNodePtr->setItem(headPtr->getItem()); // step 2
        Node<ItemType>* nodeToDeletePtr = headPtr; // step 3
        headPtr = headPtr->getNext();
        delete nodeToDeletePtr;
        nodeToDeletePtr = nullptr;
        itemCount--;
    }
    return canRemoveItem;
}
```

# getPointerTo ()

- Given an item, we want to find the **location** of the first copy of this item.
  - In **ArrayBag**, the location is represented by an array index.
  - In **LinkedBag**, the location is represented by an **address**.
- We want to define the following private function (why private?):

```
template<class ItemType>
Node<ItemType>* LinkedBag<ItemType>
    ::getPointerTo(const ItemType& anEntry) const
```

- If the item does not exist, return **nullptr**.
  - Note that returning a pointer is nothing but returning an address.
- The idea: Traverse the list, stop when a node contains the given item, and then return the address.

# getPointerTo ()

- The implementation:
- If the bag is empty, **curPtr** will be set to **nullptr** at initialization.
- If the item does not exist, **curPtr** will be set to **nullptr** when we assign **curPtr->getNext ()** to **curPtr** for the last time.
- In either case, **nullptr** will be returned.

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

# contains ()

- With `getPointerTo()`, `contains()` is easy.

```
template<class ItemType>
bool LinkedBag<ItemType>::contains(const ItemType& anEntry) const
{
    return (getPointerTo(anEntry) != nullptr);
}
```

# getFrequencyOf ()

- A single traversal determines the frequency of a given item:

```
template<class ItemType>
int LinkedBag<ItemType>::getFrequencyOf(const ItemType& anEntry) const
{
    int frequency = 0;
    Node<ItemType>* curPtr = headPtr;
    while(curPtr != nullptr)
    {
        if(anEntry == curPtr->getItem())
            frequency++;
        curPtr = curPtr->getNext();
    }
    return frequency;
}
```



# clear()

- The function **clear()** releases all the **dynamically allocated spaces**:

```
template<class ItemType>
void LinkedBag<ItemType>::clear()
{
    Node<ItemType>* nodeToDeletePtr = headPtr;
    while(headPtr != nullptr)
    {
        headPtr = headPtr->getNext();
        delete nodeToDeletePtr;
        nodeToDeletePtr = headPtr;
    }
    itemCount = 0;
}
```

# Outline

- Nodes.
- Implementations of member functions.
- **Constructors and destructor.**
- Remarks.

# Constructors and the destructor

- As always, the class **LinkedBag** should have some constructors.
  - It should have a **default constructor** (and probably some other constructors with various parameters).
  - It should have a **copy constructor** to do deep copy.
- It should have a **destructor** to deallocate dynamically allocated nodes.

# The class `LinkedBag`

- The basic elements of `LinkedBag`:

```
template<class ItemType>
class LinkedBag : public BagInterface<ItemType>
{
private:
    Node<ItemType>* headPtr;
    int itemCount;
    // ...
public:
    LinkedBag(); // default constructor
    LinkedBag(const LinkedBag<ItemType>& aBag); // copy constructor
    virtual ~LinkedBag(); // destructor
    // ...
};
```

# Default constructor and destructor

- The default constructors and destructor of **LinkedList**:

```
template<class ItemType>
LinkedList<ItemType>::LinkedList()
    : headPtr(nullptr), itemCount(0)
{
}

template<class ItemType>
LinkedList<ItemType>::~~LinkedList()
{
    clear();
}
```

- The destructor is set **virtual**. Why?

# Virtual destructor

- Suppose the destructor is not virtual:
  - If **DerivedLB** has its own dynamic contents, it needs to have **its own destructor** to clean them up.
- If we use polymorphism on the **LinkedList-DerivedLB** relationship:
  - When we **delete b** to “release the space,” the behavior is **undefined**.
  - **LinkedList**’s destructor may be called.
- To ensure the invocation of **DerivedLB**’s destructor, set **LinkedList**’s destructor to be virtual.

```
class LinkedList
{
    // ...
    ~LinkedList() { // some cleaning }
};
class DerivedLB : public LinkedList
{
    // ...
    ~DerivedLB() { // more cleaning }
}
```

```
LinkedList* b = new DerivedLB();
// use b to do something
delete b; // What will happen?
```

# Copy constructor (shallow copy)

- If we do not define a copy constructor by ourselves, the **default copy constructor** of **LinkedList** will be like:

```
template<class ItemType>
LinkedList<ItemType>::LinkedList
    (const LinkedList<ItemType>& aBag)
{
    itemCount = aBag->itemCount;
    headPtr = aBag->headPtr;
}
```

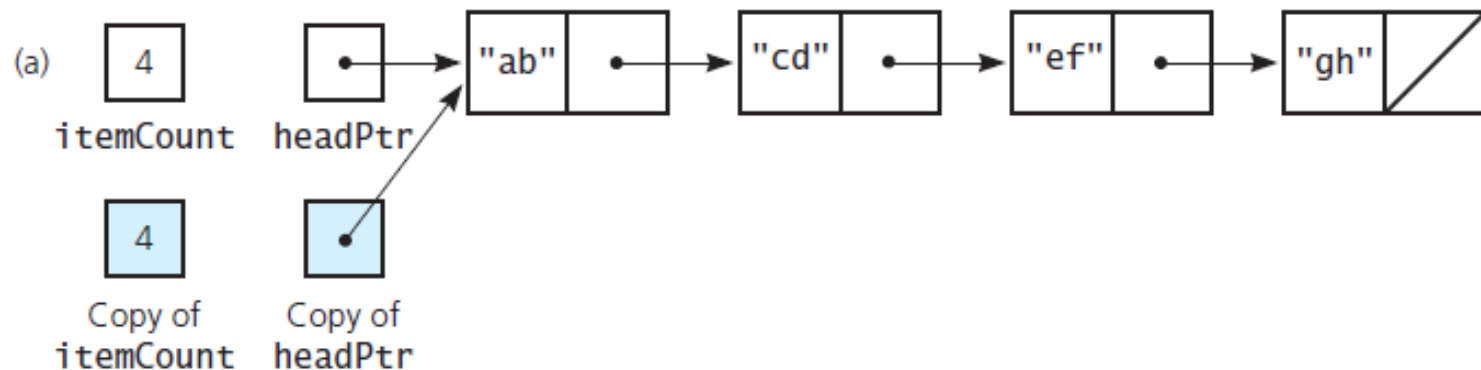


Figure 4-8 (a) (Carrano and Henry, 2015)

# Copy constructor (deep copy)

- What we want is **deep copy**: A new list of items should be dynamically created.

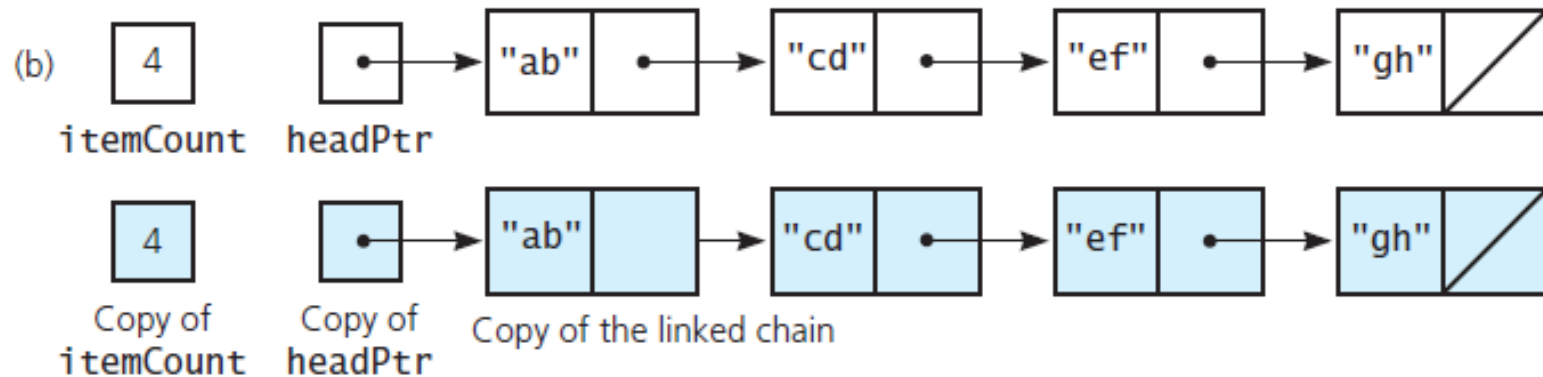


Figure 4-8 (b) (Carrano and Henry, 2015)

- To do so, we need to define our own copy constructor for **LinkedList**:



# Copy constructor (deep copy)

- The copy constructor of **LinkedBag**:

```
template<class ItemType>
LinkedBag<ItemType>::LinkedBag(const LinkedBag<ItemType>& aBag)
{
    itemCount = aBag->itemCount;
    Node<ItemType>* origChainPtr = aBag->headPtr;
    if(origChainPtr == nullptr)
        headPtr = nullptr;
    else
    {
        headPtr = new Node<ItemType>();
        headPtr->setItem(origChainPtr->getItem());
        // continue to the next page
    }
}
```

# Copy constructor (deep copy)

- The copy constructor of **LinkedBag**:

```
// continue from the previous page
Node<ItemType>* newChainPtr = headPtr;
while(origChainPtr != nullptr)
{
    origChainPtr = origChainPtr->getNext();
    ItemType nextItem = origChainPtr->getItem();
    Node<ItemType>* newNodePtr = new Node<ItemType>(nextItem);
    newChainPtr->setNext(newNodePtr);
    newChainPtr = newChainPtr->getNext();
}
newChainPtr->setNext(nullptr);
}
```

# Outline

- Nodes.
- Implementations of member functions.
- Constructors and destructor.
- **Remarks.**

# Recursion

- **Recursion** can again be applied to implement member functions.
  - E.g., when the function does a list traversal.
- Let's use **getPointerTo ()** and **toVector ()** to illustrate the idea.
  - Other functions may also be implemented with recursion.

# Recursive `getPointerTo()`

- `getPointerTo()` may be redefined as a recursive function.
  - A new argument is added.

```
template<class ItemType>
Node<ItemType>* LinkedList<ItemType>
    ::getPointerTo(const ItemType& target,
                 Node<ItemType>* curPtr) const
{ // pass headPtr as the 2nd argument
  // to traverse the whole bag
  Node<ItemType>* result = nullptr;
  if(curPtr != nullptr)
  {
    if(target == curPtr->getItem())
      result = curPtr;
    else
      result = getPointerTo(target, curPtr->getNext());
  }
  return result;
}
```

# Recursive toVector ()

- Similarly, `toVector ()` can be re-implemented:
  - We add a **recursive member function** `fillVector ()`.

```
template<class ItemType>
vector<ItemType> LinkedBag<ItemType>
    ::toVector() const
{
    vector<ItemType> bagContents;
    fillVector (bagContents, headPtr);
    return bagContents;
}
```

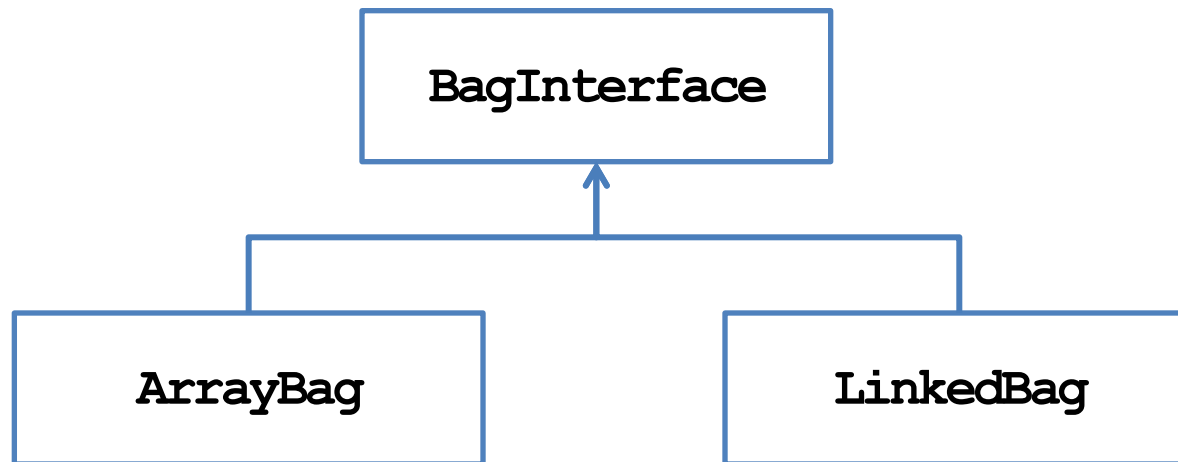
# Recursive toVector ()

```
template<class ItemType>
void LinkedBag<ItemType>
    ::fillVector (vector<ItemType>& bagContents,
                 Node<ItemType>* curPtr) const
{
    if (curPtr != nullptr)
    {
        bagContents.push_back (curPtr->getItem());
        fillVector (bagContents, curPtr->getNext());
    }
}
```

- Why do we define **fillVector ()**? How about **getPointerTo ()**?
- Should **fillVector ()** be private, public, or protected?
- Does recursion improve the efficiency of the program?

# Polymorphism regarding the ADT bag

- We have defined a pure virtual base class **BagInterface**.
- We have implemented two concrete derived classes **ArrayBag** and **LinkedBag**.



- Thanks to polymorphism, we may now **dynamically** determine which implementation to use at the run time.



# Polymorphism regarding the ADT bag

```
int main()
{
    BagInterface<string>* bagPtr = nullptr;
    char userChoice = 0;
    cin >> userChoice;
    if(userChoice == 'A')
        bagPtr = new ArrayBag<string>();
    else
        bagPtr = new LinkedBag<string>();
    bagTester(bagPtr);
    delete bagPtr;
    bagPtr = nullptr;
    return 0;
}
```

```
void bagTester
    (BagInterface<string>* bagPtr)
{
    string items[] = {"aa", "bb", "cc"};
    for(int i = 0; i < 3; i++)
        bagPtr->add(items[i]);
    cout << bagPtr->isEmpty();
    cout << bagPtr->getCurrentSize();
    bagPtr->remove("two");
    cout << bagPtr->isEmpty();
    cout << bagPtr->getCurrentSize();
}
```

# Benefits of link-based implementations

---

## Array-based Implementations

A static array has a size limit

A dynamic array requires a lot of time to increase the size

May need to predict the maximum number of items

May waste space

## Link-based Implementations

Does not have a size limit

Does not have a size limit

No need to predict this

Use space when you need it

---

# Benefits of array-based implementations

---

## Array-based Implementations

Require less space to store an item

One may directly access any item

Accessing each item requires a constant time

Easier to write the program

## Link-based Implementations

Require more space to store an item

A traversal is needed to access an item

Accessing the  $i$ th items takes more time for larger  $i$

Harder to write the program

---

# Comparisons

- Use arrays if:
  - The maximum number of items is fixed and known.
  - The maximum number of items may vary but will be small.
  - Accessing efficiency is critical for you.
- Use links if:
  - The number of items varies a lot.
  - The data portion occupies huge space.