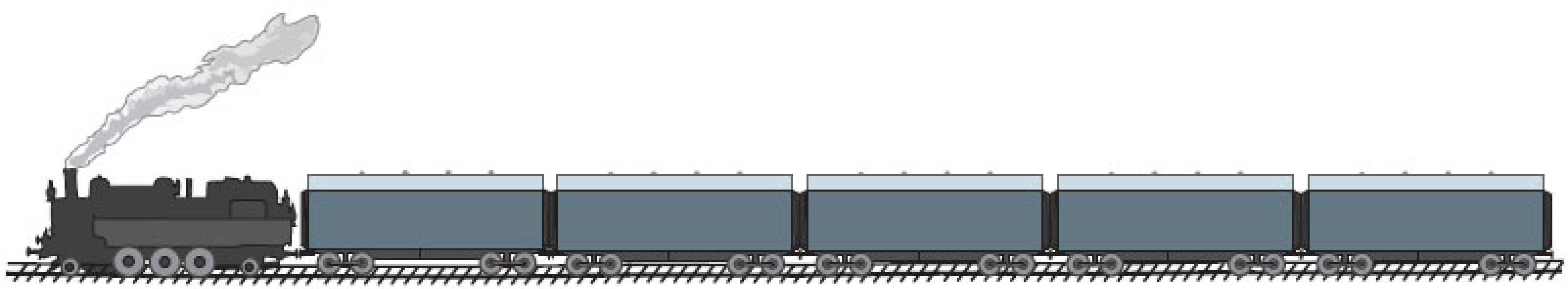
EECE 2560:
Fundamentals of Engineering Algorithms

Link-Based Implementations

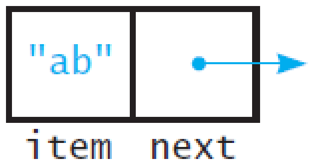
Preliminaries (1 of 2)

- Another way to organize data items
  - Place them within objects—usually called nodes
  - Linked together into a "chain," one after the other

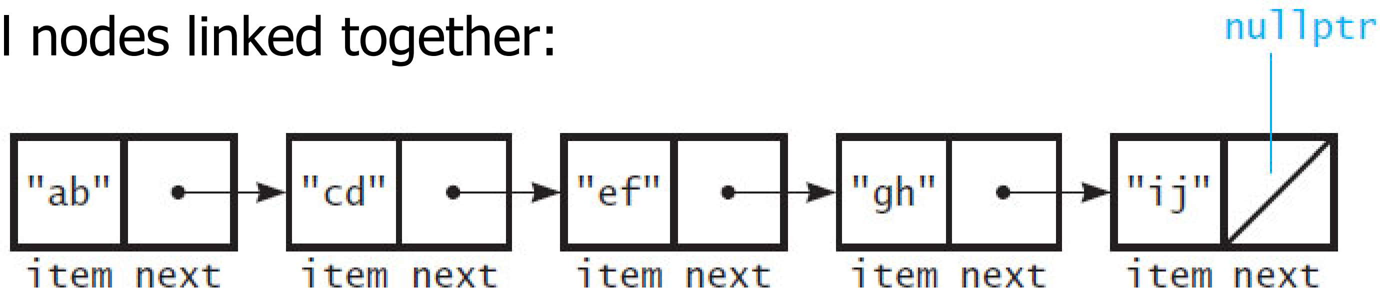


Preliminaries (2 of 2)

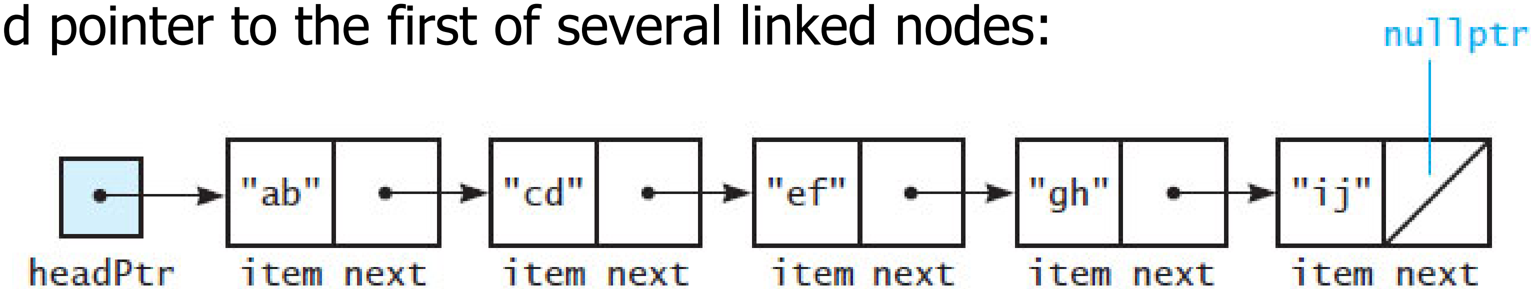
A node:



Several nodes linked together:



A head pointer to the first of several linked nodes:



The Class Node File

```
template<class 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
```

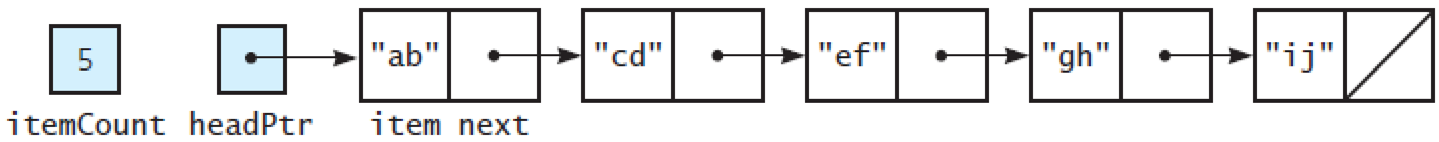
Class Node Implementation (1 of 2)

```
template<class ItemType>
Node<ItemType>::Node() : next(nullptr) // Initialization of next using the
          //Initializer List approach. Here this approach is optional , however it is
          // mandatory in cases like initializing a constant data member
{
} // end default constructor

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

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

Class Node Implementation (2 of 2)

```
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

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
```

A Link Implementation of the ADT Bag




```
+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: ItemType): boolean
+remove(anEntry: ItemType): boolean
+clear(): void
+getFrequencyOf(anEntry: ItemType): integer
+contains(anEntry: ItemType): boolean
+toVector(): vector
```

LinkedBag Header File

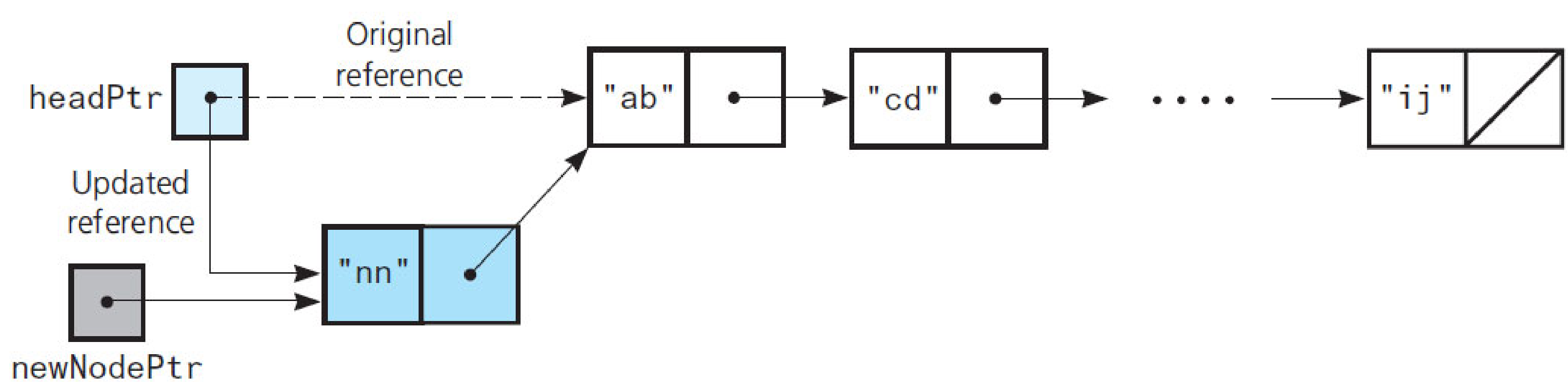
```
template<class ItemType>
class LinkedBag : public BagInterface<ItemType> {
private:
   Node<ItemType>* headPtr; // Pointer to first node
   int itemCount;           // Current count of bag items
   Node<ItemType>* getPointerTo(const ItemType& target) const;
  //Returns either a pointer to the node containing a given entry or the null pointer if the entry is not in the bag.
  //It is declared as private as returning a pointer is an implementation detail that should be hidden from the client

public:
   LinkedBag();
   LinkedBag(const LinkedBag<ItemType>& aBag); // Copy constructor
   virtual ~LinkedBag();  // Destructor
   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;
   std::vector<ItemType> toVector() const;
}; // end LinkedBag
```

LinkedBag Default Constructor

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

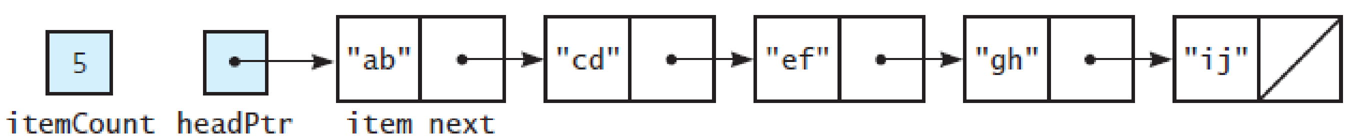
LinkedBag Inserting At the Beginning of the Chain

```
template<class ItemType>
bool LinkedBag<ItemType>::add(const ItemType& newEntry){
   // Add to beginning of chain: new node references rest of chain; (headPtr is null if chain is empty)
   Node<ItemType>* nextNodePtr = new Node<ItemType>();
   nextNodePtr->setItem(newEntry);
   nextNodePtr->setNext(headPtr);  // New node points to chain

   headPtr = nextNodePtr;          // New node is now first node
   itemCount++;

   return true;
}  // end add
```



LinkedBag Defining toVector

```
#include <vector> // vector class-template definition
template<class ItemType>
std::vector<ItemType> LinkedBag<ItemType>::toVector() const
{  std::vector<ItemType> bagContents; // create vector of ItemType
   Node<ItemType>* curPtr = headPtr;
   int counter = 0;
   while ((curPtr != nullptr) && (counter < itemCount)) {
// function push_back() to add an element to the end of the vector.
// If an element is added to a full vector, the vector increases its size
      bagContents.push_back(curPtr->getItem());
      curPtr = curPtr->getNext();
      counter++; //counter , while not necessary, provides a defense against
                                //going beyond the end of the chain
   }  // end while

   return bagContents;
}  // end toVector
```

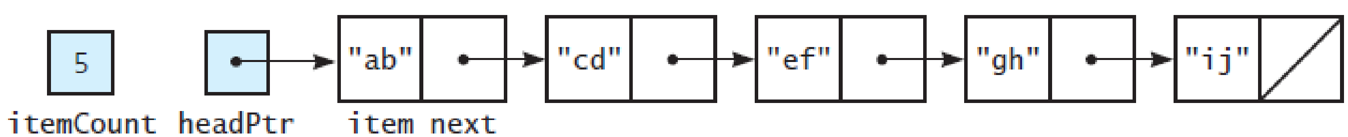


LinkedBag Defining isEmpty and getCurrentSize

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


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

LinkedBag Defining getFrequencyOf

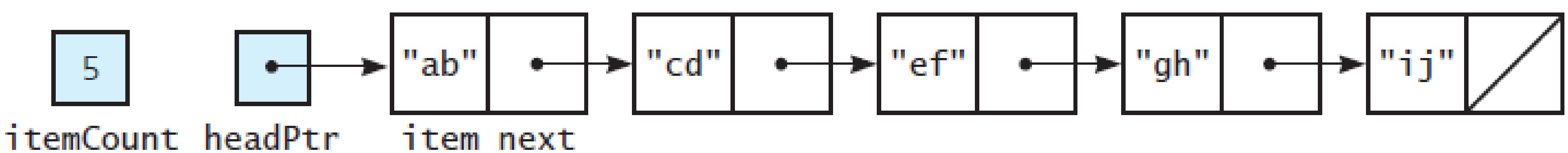
```
template<class ItemType>
int LinkedBag<ItemType>::getFrequencyOf(const ItemType& anEntry) const
{
   int frequency = 0;
   int counter = 0;
   Node<ItemType>* curPtr = headPtr;
   while ((curPtr != nullptr) && (counter < itemCount))
   {
      if (anEntry == curPtr->getItem()) frequency++;

      counter++;
      curPtr = curPtr->getNext();
   } // end while

  return frequency;
}  // end getFrequencyOf
```



LinkedBag Defining getPointerTo

// Returns a pointer to the node containing a given entry or the null pointer if the entry is not in the bag.

```
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();
   } // end while
   return curPtr;
} // end getPointerTo
```

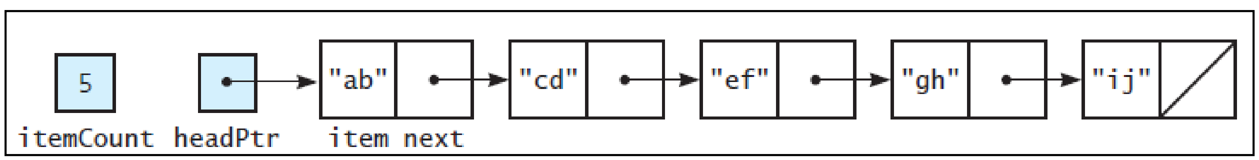


LinkedBag Defining contains

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

LinkedBag Defining remove

```
template<class ItemType>
bool LinkedBag<ItemType>::remove(const ItemType& anEntry) {
   Node<ItemType>* entryNodePtr = getPointerTo(anEntry);
   bool canRemoveItem = !isEmpty() && (entryNodePtr != nullptr);
   if (canRemoveItem){
      // Copy data from first node to located node
      entryNodePtr->setItem(headPtr->getItem());
      // Delete first node
      Node<ItemType>* nodeToDeletePtr = headPtr;
      headPtr = headPtr->getNext();
      // Return node to the system
      nodeToDeletePtr->setNext(nullptr); //defensive step but not needed
      delete nodeToDeletePtr;
      nodeToDeletePtr = nullptr; //defensive step but not needed

      itemCount--;
   } // end if
      return canRemoveItem;
}  // end remove
```



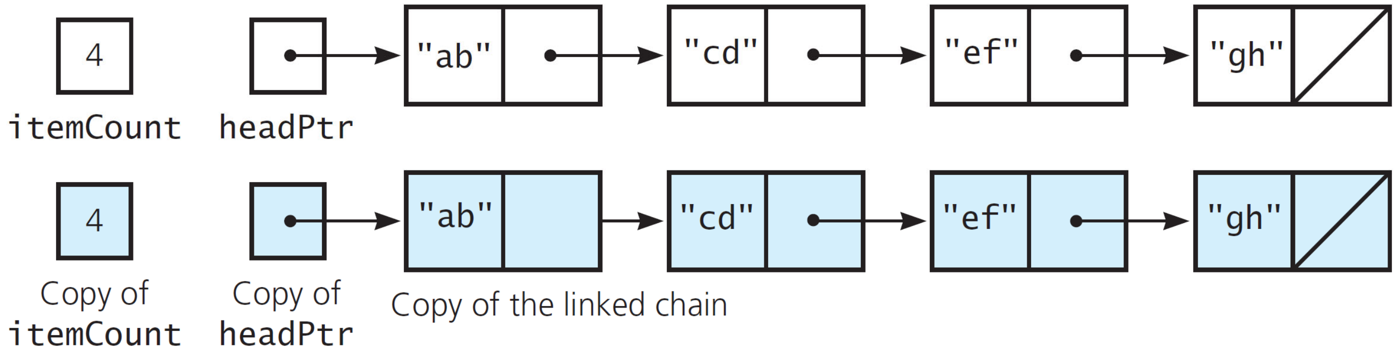
LinkedBag Defining clear

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

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

      nodeToDeletePtr = headPtr;
   }  // end while
   // headPtr is nullptr; nodeToDeletePtr is nullptr

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

LinkedBag Defining Destructor

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

LinkedBag Defining Copy Constructor (1 of 2)




```
template<class ItemType>
LinkedBag<ItemType>::LinkedBag(const LinkedBag<ItemType>& aBag)
{
   itemCount = aBag.itemCount;
   // Points to nodes in original chain
   Node<ItemType>* origChainPtr = aBag.headPtr;
   if (origChainPtr == nullptr)
      headPtr = nullptr;  // Original bag is empty
   else {
      // Copy first node
      headPtr = new Node<ItemType>();
      headPtr->setItem(origChainPtr->getItem());
```

LinkedBag Defining Copy Constructor

```
      // Copy remaining nodes
      Node<ItemType>* newChainPtr = headPtr; // Points to last node in new chain
      origChainPtr = origChainPtr->getNext(); // Advance original-chain pointer

      while (origChainPtr != nullptr) {
         // Get next item from original chain
         ItemType nextItem = origChainPtr->getItem();

         // Create a new node containing the next item
         Node<ItemType>* newNodePtr = new Node<ItemType>(nextItem);
         newChainPtr->setNext(newNodePtr); // Link new node to end of new chain
         // Advance pointer to new last node
         newChainPtr = newChainPtr->getNext();
         origChainPtr = origChainPtr->getNext();// Advance original-chain pointer
      }  // end while

      newChainPtr->setNext(nullptr); // Flag end of chain
   }  // end if
}  // end copy constructor
```

Array-Based vs. Link-Based Implementations (1 of 2)

- Arrays easy to use, but have fixed size
  - Not always easy to predict number of items in A D T
  - Array could waste space
  - Increasing size of dynamically allocated array can waste storage and time
  - Can access array items directly with equal access time
  - An array-based implementation is a good choice for a small bag

Array-Based vs. Link-Based Implementations (2 of 2)

- Linked chains do not have fixed size
  - In a chain of linked nodes, an item points explicitly to the next item
  - Link-based implementation requires more memory
  - Must traverse a linked chain to access its i^{th} node
  - Time to access i^{th} node in a linked chain depends on i