Linked Lists

Abstract Data Structures

Abstraction:

Separate the purpose of a program's module from its implementation:

- First phase: the purpose of the code must be specified in details
- Second phase: implementation.

Abstract Data Structures:

A class of objects whose logical behaviour is defined by a set of values and a set of operations, in analogy (for example) to algebraic structures in mathematics.

In Other Words:

ATDs allow to separate the **WHAT?** from the **HOW?**

Summary up to now and plan:

The "Bag" Data Structure

- Contains generic objects
- No ordering
- Fixed size
- We implemented it with an "array" data structure.

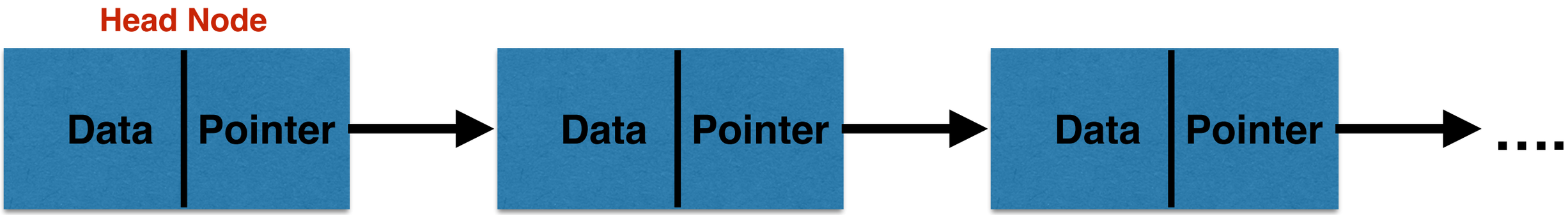
A new implementation: the LINKED LIST

- We would like to go beyond the Fixed Size limitation
- We have to move to a *dynamic* data structure
- This can be achieved with a *linked list*.

The Linked List

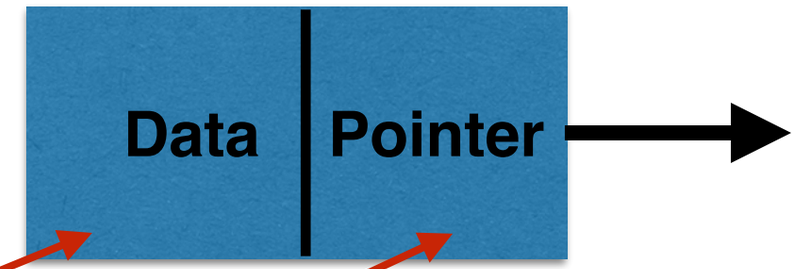
Node:

Linked List:



The LL is a sequence of nodes linked by pointers.
The first node is called “head node”

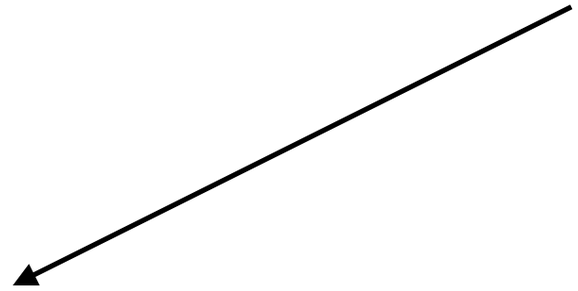
The Node




```
#ifndef _NODE
#define _NODE

template<class ItemType>
class Node {
 private:
  ItemType item;
  Node<ItemType>* next; //pointer to a Node class
 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;
};

#include "Node.cpp"
#endif
```

The LinkedBag Class

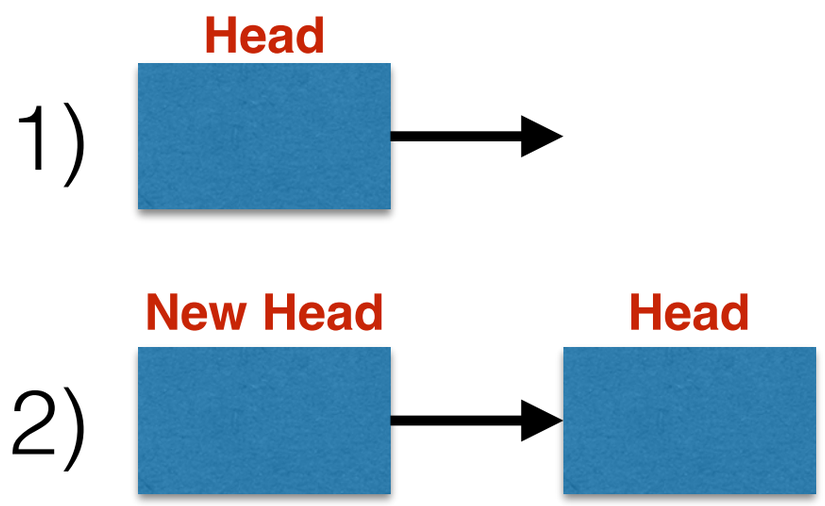
Remember the abstract class we defined for ArrayBag



```
#include "BagInterface.h"
#include "Node.h"

template<class ItemType>
class LinkedBag : public BagInterface<ItemType>
{
 private:
  Node<ItemType>* headPtr;
  int itemCount;
  Node<ItemType>* getPointerTo(const ItemType& target) const;

 public:
  LinkedBag();
  LinkedBag(const LinkedBag<ItemType>& aBag);
  virtual ~LinkedBag();
  int getCurrentSize() const;
  bool isEmpty() const;
  bool add(const ItemType& newEntry);
  bool remove(const ItemType& anEntry);
  void clear();
  int getFrequencyOf(const ItemType& anEntry) const;
  bool contains(const ItemType& anEntry) const;
  vector<ItemType> toVector() const;
};

#include "LinkedBag.cpp"
```

Implementation: Constructor and "Add"

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

//The "add" method
template<class ItemType>
bool LinkedBag<ItemType>::add(const ItemType& newEntry)
{
  Node<ItemType>* newNodePtr = new Node<ItemType>();
  //Insert a new node at the beginning of the linked list
  newNodePtr->setItem(newEntry);
  newNodePtr->setNext(headPtr);
  headPtr = newNodePtr;

  itemCount++;

  return true;
}
```



Implementation: the accessory method 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();
  }
  return curPtr;
}
```

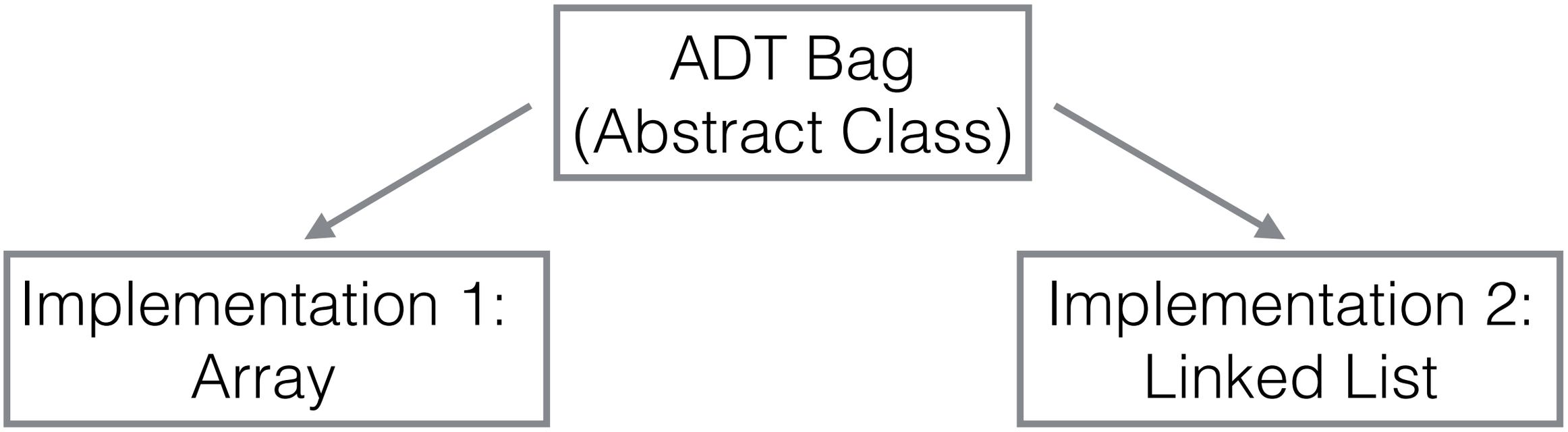
Private method which returns the “pointer to next” of a specific item.
Other public methods need this service.

Implementation: 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 item from first node

    //remove first node
    Node<ItemType>* nodeToDeletePtr = headPtr;
    headPtr = headPtr->getNext();

    nodeToDeletePtr->setNext(nullptr);
    delete nodeToDeletePtr;      //delete node pointed
    nodeToDeletePtr = nullptr;   //avoid “dangling” pointers.

    itemCount--;
  }
  return canRemoveItem;
}
```

Replace the item to remove with the head’s item.
Remove the old head.
The node next to the old head becomes the new head.

Bags Comparison:



How do you choose the implementation?

- Fixed vs Dynamic array
- Algorithms performance
- Memory Usage
- …

Interlude: STL Container Classes

STL: Standard Template Libraries

Contain many template collections ready to use.

Example: **`vector`**

```
#include <vector>

vector<int> v;
v.push_back(2);
cout << v[0] << endl;

for (int i;i<v.size();i++) cout << v[i];
```

Other available methods:

Assign vector content
push_back Add element at the end
pop_back Delete last element
insert Insert elements
erase Erase elements
swap Swap content
clear Clear content

It works very much like our LinkedList Bag!

Interlude: STL Container Classes

Other Available Containers:

```
<array>
<deque>
<forward_list>
<list>
<map>
<queue>
<set>
<stack>
<unordered_map>
<unordered_set>
```