HW #4 (Bag ADT)

- **A bag is an ordered collection of items with duplicates allowed.**
- **In this project you will write a program (using C++) to implement a couple of algorithms that operate on the bag ADT (Abstract Data Type).**

HW #4 (2)

- **Consider the Bag interface (here the Bag is a template, ItemType can be any class or primitive):**

```
template < ItemType >
class Bag {
public:
    Bag();        // create an empty bag.
    bool empty() const;  // return true iff the bag is empty.
```

HW #4 (3)

int size() const;

// return the number of items in the bag. For
// example, the size of a bag containing “duck”,
// “duck”, “duck”, “goose” is 4.

int uniqueSize() const;

// return the number of distinct items in the bag.
// For example, the uniqueSize of a bag
// containing “duck”, “duck”, “duck”, “goose” is
// 2.

HW #4 (4)

bool insert(const ItemType& value);

// insert value into the bag. Return true iff the
// value could be inserted. (It must not if the bag
// has a fixed capacity.)

int erase(const ItemType& value);

// remove one instance of value from the bag if
// present. Return the number of instances
// removed, which will be 1 or 0.

HW #4 (5)

int eraseAll(const ItemType& value);

// remove all instances of value from the bag if
// present. Return the number of instances
// removed.

bool contains(const ItemType& value) const;

// return true iff value is in the bag.

int count(const ItemType& value) const;

// return the number of instances of value in the
// bag.

HW #4 (6)

```
    // Iteration functions:
    void start();
    void next();
    bool ended() const;
    const ItemType& currentValue() const;
    int currentCount() const;
};
```

HW #4 (7)

- **The iteration functions allow a client to write code that will visit every item in the bag.**
- **Image a finger that will be used to point to each distinct item in the bag.**
- **The *start()* function initializes the finger to point to one of those items.**
- **Each call to the *next()* function makes the finger point to another distinct item not yet visited since *start()* was last called.**

HW #4 (8)

- **Eventually, however, all items will have been visited, so *next()* makes the finger not point to any item.**
- **The *ended()* function returns true iff the finger is not pointing to any item.**
- **If the bag is empty when *start()* is called, *ended()* will return true.**

HW #4 (9)

- **When the finger is pointing to an item, *currentValue()* returns a reference to that item, and *currentCount()* returns the number of instances of that item in the bag. This specification does not require that the iteration visit the distinct items in any particular order.**

HW #4 (10)

- **The behavior of the functions in the following situations is underfined – if a client does any of the these things, the program might do anything: appear to work, produce incorrect results, crash immediately, crash later, do horrible things, etc.**

  1. Calling *next(), ended(), currentValue(),* or *currentCount()* before having called *start().* (In other words, a client must call *start()* to initialize an iteration through the bag.)

HW #4 (11)

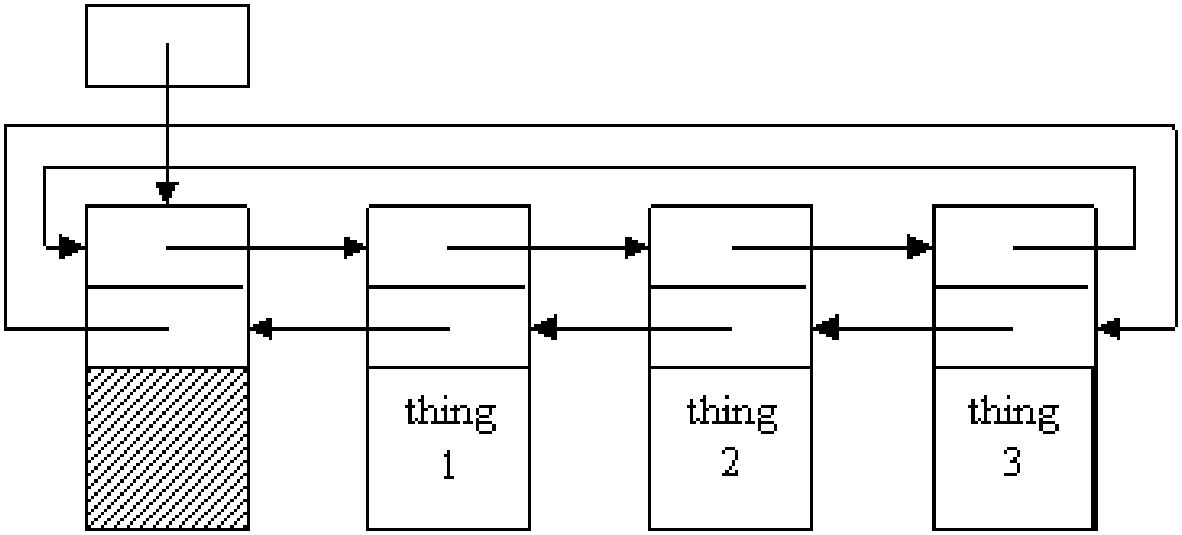
2. Calling *next(), currentValue(),* or *currentCount()* when *ended()* would return true. (In other words, when the iteration has ended and the finger is not pointing to any item, these functions should not be called.)
3. Modifying the bag and later calling *next(), ended(), currentValue(),* or *currentCount()* without an intervening call to *start().*

HW #4 (12)

(In other words, when a bag is modified, a client should not continue with any iteration that may be in progress, but a fresh iteration may be started.)

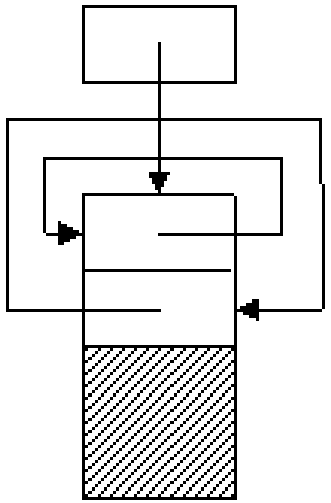
HW #4 (13)

- **Implement the Bag interface using a circular doubly-linked list (not array) with a dummy element.**
- **Here is a sketch of what such a list with three things in it looks like.**



HW #4 (14)

- **Here is what an empty list looks like.**
- **The presence of a dummy node greatly simplifies the code for inserting or removing nodes in the list.**
- **The node is a “dummy” because no code will ever do anything with any part of it except the two pointers.**



HW #4 (15)

- **You will have to declare and implement other three member functions as well.**
- **Destructor: when a bag is destroyed, the nodes in the linked list must be deallocated.**
- **Copy constructor: when a brand new Bag is created as a copy of an existing bag, enough new nodes must be allocated to hold a duplicate of the original list.**

HW #4 (16)

- **Assignment operator: when an existing Bag (the left-hand side) is assigned the value of another Bag (the right-hand side), the result must be that the left-hand side object is a duplicate of the right-hand side object, with no memory leak of list nodes (i.e., no list node from the old value of the left-hand side should be still allocated yet inaccessible).**
- **Note that there is no a priori limit on the maximum number of items in the Bag.**

HW #4 (17)

- **Using only the public interface of Bag, implement the following two non-member functions.**
- ***void combine(Bag<ItemType>& bag1, Bag<ItemType>& bag2, Bag<ItemType>& result);***

  When this function returns, "result" must contain all the elements that appear in "bag1" or "bag2" or both, and must not contain any other values. (You may not assume "result" is empty when it is passed to this function.)

HW #4 (18)

If a value appears *n1* times in "bag1" and *n2* times in "bag2", it must appear *n1*+ *n2* times in "result".

- ***void subtract(Bag<ItemType>& bag1, Bag<ItemType>& bag2, Bag<ItemType>& result);***

  If a value appears *n1* times in "bag1" and *n2* times in "bag2", then when this function returns, it must appear *n1 – n2* times in "result" if *n1* > *n2*; otherwise, it must not appear in "result". (You may not assume result is empty when it is passed to this function.)

HW #4 (19)

- **Be sure these functions behave correctly in the face of aliasing: what if “bag1” and “result” refer to the same Bag, for example?**
- **Unless “bag1” and “result” refer to the same Bag, these functions must not change the number of each item stored in “bag1”.**
- **Unless “bag2” and “result” refer to the same Bag, these functions must not change the number of each item stored in “bag2”.**

HW #4 (20)

- **Your code may not write anything to cout. If you want to print thing out for debugging purposes, write to cerr instead of cout.**
- **Turn it in with a list of test cases that would thoroughly test the functions. Be sure to indicate the purpose of the tests. For example, here is the beginning of a presentation in the form of code:**

HW #4 (21)

The tests were performed on a bag of strings (i.e., Itemtype was std::string).

```
// default constructor
Bag < std::string > a;
// for an empty bag
assert (a.size() == 0);          // test size
assert (a.empty() );             // test empty
assert (a.erase("duck") == 0);   // nothing to remove
```