Worksheet 23: Introduction to the Iterator

Group 11:
Tatyana Vlaskin
Danny Mejia
Katherine Jensen
Michael Sigel – did not participate at all

In Preparation: Read Chapter 8 to learn more about the idea of encapsulation and the iterator. If you have not done it already, complete Worksheets 14 and 15 to learn more about adding and removing values from a dynamic array.

One of the primary design principles for collection classes is *encapsulation*. The internal details concerning how an implementation works are hidden behind a simple and easy to remember interface. To use a Bag, for example, all you need know is the basic operations are add, collect and remove. The inner workings of the implementation for the bag are effectively hidden.

When using collections a common requirement is the need to loop over all the elements in the collection, for example to print them to a window. Once again it is important that this process be performed without any knowledge of how the collection is represented in memory. For this person the conventional solution is to use a mechanism termed an *Iterator*.

```
/* conceptual interface */
 Boolean  hasNext ( );
 TYPE next ( );
 void remove ( );
```

```
dynArrayIterator itr;
TYPE current;
dynArray data;
…
dynArrayIteratorInit (&data,&itr);
while (dynArrayHasNext(&itr)) {
   current = dynArrayNext(&itr);
   … /* do something with current */
}
```

Each collection provides an associated iterator structure. Functions are used to initialize and manipulate this iterator. These functions are used in combination to write a simple loop that will cycle over the values in the collection. For example, the code at left illustrates the use of our dynamic array iterator.

Notice that the iterator loop exposes nothing of the structure of the container class. The method remove is used to delete from the collection the value most recently returned by next. Calls on the functions hasNext and next must always be interleaved, as shown.

Note that an iterator is an object that is separate from the collection itself. The iterator is a *facilitator object*, one that provides access to the container values.

An iterator for the dynamic array will work by maintaining an index into the array representing the current location. This value is initially zero, and must be incremented by either the function hasNext or the function next? Think about this for a moment. Which one makes more sense? Then think carefully about the third function, **remove**. You must ensure that when the loop continues after a remove the next value following the removed element is not skipped. You can assume you have access to the `dynArrayRemoveAt` function you wrote in Lesson 21.

```
struct dynArrayIterator {
   struct dynArray * da;
   int currentIndex;
};

void dynArrayRemoveAt (struct dynArray * dy, int index);
{
            int i;
            assert(index < dy->size);
            assert(index >= 0);

             ( i = index; i < dy->size; i++)
             dy->data[index] = dy->data[index+1];
             dy->size--;
}

void dynArrayIteratorInit (struct dynArray *da, struct dynArrayIterator *itr) {
             struct dynArrayIterator itr = malloc (sizeof(struct dynArrayIterator));
             assert( itr != 0);

             itr->da = da;

             return(Itr);
             }

int dynArrayIteratorHasNext (struct dynArrayIterator *itr) {

             if ( itr->currentIndex->next != itr->backSentinal ) // checks if
link points to the back sentinal
                         {
                         Itr -> currentIndex = itr -> currentIndex -> next ; //
setting current index to the next node
                          return 1 ;  // if true returns 1
                         }
             else
                         return 0 ;   // else returns 0
}

TYPE dynArrayIteratorNext (struct dynArrayIterator *itr) {

             return dynArrayIterator -> da; // returning the value
}

void dynArrayIteratorRemove (struct dynArrayIterator *itr) {

             itr->currentIndex = itr->currentIndex->previous;
             itr->currentIndex->previous = itr->currentIndex;
             free (itr);
}
```