Data Structures/List Structures

List Structures and Iterators

We have seen now two different data structures that allow us to store an ordered sequence of elements. However, they have two very different interfaces. The array allows us to use `get-element()` and `set-element()` functions to access and change elements. The node chain requires us to use `get-next()` until we find the desired node, and then we can use `get-value()` and `set-value()` to access and modify its value. Now, what if you've written some code, and you realize that you should have been using the other sequence data structure? You have to go through all of the code you've already written and change one set of accessor functions into another. What a pain! Fortunately, there is a way to localize this change into only one place: by using the **List** Abstract Data Type (ADT).

`List<item-type>` ADT

`get-begin``():List Iterator<item-type>`
: Returns the *list iterator* (we'll define this soon) that represents the first element of the list. Runs in time.

`get-end``():List Iterator<item-type>`
: Returns the list iterator that represents one element past the last element in the list. Runs in time.

`prepend``(`*`new-item`*`:item-type)`
: Adds a new element at the beginning of a list. Runs in time.

`insert-after``(`*`iter`*`:List Iterator<item-type>, `*`new-item`*`:item-type)`
: Adds a new element immediately after *iter*. Runs in time.

`remove-first``()`
: Removes the element at the beginning of a list. Runs in time.

`remove-after``(`*`iter`*`:List Iterator<item-type>)`
: Removes the element immediately after *iter*. Runs in time.

`is-empty``():Boolean`
: True if there are no elements in the list. Has a default implementation. Runs in time.

`get-size``():Integer`
: Returns the number of elements in the list. Has a default implementation. Runs in time.

`get-nth``(`*`n`*`:Integer):item-type`
: Returns the nth element in the list, counting from 0. Has a default implementation. Runs in time.

`set-nth``(`*`n`*`:Integer, `*`new-value`*`:item-type)`
: Assigns a new value to the nth element in the list, counting from 0. Has a default implementation. Runs in time.

The **iterator** is another abstraction that encapsulates both access to a single element and incremental movement around the list. Its interface is very similar to the node interface presented in the introduction, but since it is an abstract type, different lists can implement it differently.

`List Iterator<item-type>` ADT

`get-value``():item-type`
: Returns the value of the list element that this iterator refers to.

`set-value``(`*`new-value`*`:item-type)`
: Assigns a new value to the list element that this iterator refers to.

`move-next``()`
: Makes this iterator refer to the next element in the list.

`equal``(`*`other-iter`*`:List Iterator<item-type>):Boolean`
: True if the other iterator refers to the same list element as this iterator.

All operations run in time.

There are several other aspects of the List ADT's definition that need more explanation. First, notice that the `get-end()` operation returns an iterator that is "one past the end" of the list. This makes its implementation a little trickier, but allows you to write loops like:

```
var iter:List Iterator := list.get-begin()
while(not iter.equal(list.get-end()))
 # Do stuff with the iterator
 iter.move-next()
end while
```

Second, each operation gives a worst-case running time. Any implementation of the List ADT is guaranteed to be able to run these operation at least that fast. Most implementations will run most of the operations faster. For example, the node chain implementation of List can run `insert-after()` in .

Third, some of the operations say that they have a default implementation. This means that these operations can be implemented in terms of other, more primitive operations. They're included in the ADT so that certain implementations can implement them faster. For example, the default implementation of `get-nth()` runs in because it has to traverse all of the elements before the nth. Yet the array implementation of List can implement it in using its `get-element()` operation. The other default implementations are:

```
abstract type List<item-type>
 method is-empty()
  return get-begin().equal(get-end())
 end method

 method get-size():Integer
  var size:Integer := 0
  var iter:List Iterator<item-type> := get-begin()
  while(not iter.equal(get-end()))
   size := size+1
   iter.move-next()
  end while
  return size
 end method

 helper method find-nth(n:Integer):List Iterator<item-type>
  if n >= get-size()
   error "The index is past the end of the list"
  end if
  var iter:List Iterator<item-type> := get-begin()
  while(n > 0)
   iter.move-next()
   n := n-1
  end while
  return iter
 end method

 method get-nth(n:Integer):item-type
  return find-nth(n).get-value()
 end method

 method set-nth(n:Integer, new-value:item-type)
  find-nth(n).set-value(new-value)
 end method
end type
```

Syntactic Sugar

Occasionally throughout this book we'll introduce an abbreviation that will allow us to write, and you to read, less pseudocode. For now, we'll introduce an easier way to compare iterators and a specialized loop for traversing sequences.

Instead of using the `equal()` method to compare iterators, we'll overload the `==` operator. To be precise, the following two expressions are equivalent:

```
iter1.equal(iter2)
iter1 == iter2
```

Second, we'll use the `for` keyword to express list traversal. The following two blocks are equivalent:

```
var iter:List Iterator<item-type> := list.get-begin()
while(not iter == list.get-end())
 operations on iter
 iter.move-next()
end while
```

```
for iter in list
 operations on iter
end for
```

Implementations

In order to actually use the List ADT, we need to write a *concrete* data type that implements its interface. There are two standard data types that naturally implement List: the node chain described in the Introduction, normally called a Singly Linked List; and an extension of the array type called a Vector, which automatically resizes itself to accommodate inserted nodes.

Singly Linked List

```
type Singly Linked List<item-type> implements List<item-type>
```

head refers to the first node in the list. When it's null, the list is empty.

```
  data head:Node<item-type>
```

Initially, the list is empty.

```
  constructor()
    head := null
  end constructor

  method get-begin():Sll Iterator<item-type>
    return new Sll-Iterator(head)
  end method
```

The "one past the end" iterator is just a null node. To see why, think about what you get when you have an iterator for the last element in the list and you call `move-next()`.

```
  method get-end():Sll Iterator<item-type>
    return new Sll-Iterator(null)
  end method

  method prepend(new-item:item-type)
    head = make-node(new-item, head)
  end method

  method insert-after(iter:Sll Iterator<item-type>, new-item:item-type)
    var new-node:Node<item-type> := make-node(new-item, iter.node().get-next())
    iter.node.set-next(new-node)
  end method

  method remove-first()
    head = head.get-next()
  end method
```

This takes the node the iterator holds and makes it point to the node two nodes later.

```
  method remove-after(iter:Sll Iterator<item-type>)
    iter.node.set-next(iter.node.get-next().get-next())
  end method
end type
```

If we want to make `get-size()` be an operation, we can add an Integer data member that keeps track of the list's size at all times. Otherwise, the default implementation works fine.

An iterator for a singly linked list simply consists of a reference to a node.

```
type Sll Iterator<item-type>
  data node:Node<item-type>

  constructor(_node:Node<item-type>)
    node := _node
  end constructor
```

Most of the operations just pass through to the node.

```
  method get-value():item-type
    return node.get-value()
  end method

  method set-value(new-value:item-type)
    node.set-value(new-value)
  end method

  method move-next()
    node := node.get-next()
  end method
```

For equality testing, we assume that the underlying system knows how to compare nodes for equality. In nearly all languages, this would be a pointer comparison.

```
  method equal(other-iter:List Iterator<item-type>):Boolean
    return node == other-iter.node
  end method
end type
```

Vector

Let's write the Vector's iterator first. It will make the Vector's implementation clearer.

```
type Vector Iterator<item-type>
  data array:Array<item-type>
  data index:Integer

  constructor(my_array:Array<item-type>, my_index:Integer)
    array := my_array
    index := my_index
  end constructor

  method get-value():item-type
    return array.get-element(index)
  end method

  method set-value(new-value:item-type)
    array.set-element(index, new-value)
  end method

  method move-next()
    index := index+1
  end method

  method equal(other-iter:List Iterator<item-type>):Boolean
    return array==other-iter.array and index==other-iter.index
  end method
end type
```

We implement the Vector in terms of the primitive Array data type. It is inefficient to always keep the array exactly the right size (think of how much resizing you'd have to do), so we store both a `size`, the number of logical elements in the Vector, and a `capacity`, the number of spaces in the array. The array's valid indices will *always* range from `0` to `capacity-1`.

```
type Vector<item-type>
  data array:Array<item-type>
  data size:Integer
  data capacity:Integer
```

We initialize the vector with a capacity of 10. Choosing 10 was fairly arbitrary. If we'd wanted to make it appear less arbitrary, we would have chosen a power of 2, and innocent readers like you would assume that there was some deep, binary-related reason for the choice.

```
  constructor()
    array := create-array(0, 9)
    size := 0
    capacity := 10
  end constructor

  method get-begin():Vector-Iterator<item-type>
    return new Vector-Iterator(array, 0)
  end method
```

The end iterator has an index of `size`. That's one more than the highest valid index.

```
  method get-end():List Iterator<item-type>
    return new Vector-Iterator(array, size)
  end method
```

We'll use this method to help us implement the insertion routines. After it is called, the `capacity` of the array is guaranteed to be at least `new-capacity`. A naive implementation would simply allocate a new array with exactly `new-capacity` elements and copy the old array over. To see why this is inefficient, think what would happen if we started appending elements in a loop. Once we exceeded the original capacity, each new element would require us to copy the entire array. That's why this implementation at least doubles the size of the underlying array any time it needs to grow.

```
  helper method ensure-capacity(new-capacity:Integer)
```

If the current capacity is already big enough, return quickly.

```
    if capacity >= new-capacity
      return
    end if
```

Now, find the new capacity we'll need,

```
    var allocated-capacity:Integer := max(capacity*2, new-capacity)
    var new-array:Array<item-type> := create-array(0, allocated-capacity - 1)
```

copy over the old array,

```
    for i in 0..size-1
      new-array.set-element(i, array.get-element(i))
    end for
```

and update the Vector's state.

```
    array := new-array
    capacity := allocated-capacity
  end method
```

This method uses a normally-illegal iterator, which refers to the item one before the start of the Vector, to trick `insert-after()` into doing the right thing. By doing this, we avoid duplicating code.

```
  method prepend(new-item:item-type)
    insert-after(new Vector-Iterator(array, -1), new-item)
  end method
```

`insert-after()` needs to copy all of the elements between *iter* and the end of the Vector. This means that in general, it runs in time. However, in the special case where *iter* refers to the last element in the vector, we don't need to copy any elements to make room for the new one. An append operation can run in time, plus the time needed for the `ensure-capacity()` call. `ensure-capacity()` will sometimes need to copy the whole array, which takes time. But much more often, it doesn't need to do anything at all.

Amortized Analysis
In fact, if you think of a series of append operations starting immediately after `ensure-capacity()` increases the Vector's capacity (call the capacity here), and ending immediately after the next increase in capacity, you can see that there will be exactly appends. At the later increase in capacity, it will need to copy elements over to the new array. So this entire sequence of function calls took operations. We call this situation, where there are operations for function calls "*amortized* time".

```
  method insert-after(iter:Vector Iterator<item-type>, new-item:item-type)
    ensure-capacity(size+1)
```

This loop copies all of the elements in the vector into the spot one index up. We loop backwards in order to make room for each successive element just before we copy it.

```
    for i in size-1 .. iter.index+1 step -1
      array.set-element(i+1, array.get-element(i))
    end for
```

Now that there is an empty space in the middle of the array, we can put the new element there.

```
    array.set-element(iter.index+1, new-item)
```

And update the Vector's size.

```
    size := size+1
  end method
```

Again, cheats a little bit to avoid duplicate code.

```
  method remove-first()
   remove-after(new Vector-Iterator(array, -1))
  end method
```

Like `insert-after()`, `remove-after` needs to copy all of the elements between *iter* and the end of the Vector. So in general, it runs in time. But in the special case where *iter* refers to the last element in the vector, we can simply decrement the Vector's size, without copying any elements. A remove-last operation runs in time.

```
  method remove-after(iter:List Iterator<item-type>)
    for i in iter.index+1 .. size-2
      array.set-element(i, array.get-element(i+1))
    end for
    size := size-1
  end method
```

This method has a default implementation, but we're already storing the size, so we can implement it in time, rather than the default's

```
  method get-size():Integer
    return size
  end method
```

Because an array allows constant-time access to elements, we can implement `get-` and `set-nth()` in , rather than the default implementation's

```
  method get-nth(n:Integer):item-type
    return array.get-element(n)
  end method

  method set-nth(n:Integer, new-value:item-type)
    array.set-element(n, new-value)
  end method
end type
```

Bidirectional Lists

Sometimes we want Data Structures/List Structures to move backward in a list too. A Bidirectional List allows the list to be searched forwards and backwards. A Bidirectional list implements an additional **iterator** function, **`move-previous()`**.

`Bidirectional List<item-type>` ADT

`get-begin``():Bidirectional List Iterator<item-type>`
: Returns the *list iterator* (we'll define this soon) that represents the first element of the list. Runs in time.

`get-end``():Bidirectional List Iterator<item-type>`
: Returns the list iterator that represents one element past the last element in the list. Runs in time.

`insert``(`*`iter`*`:Bidirectional List Iterator<item-type>, `*`new-item`*`:item-type)`
: Adds a new element immediately before *iter*. Runs in time.

`remove``(`*`iter`*`:Bidirectional List Iterator<item-type>)`
: Removes the element immediately referred to by *iter*. After this call, *iter* will refer to the next element in the list. Runs in time.

`is-empty``():Boolean`
: True iff there are no elements in the list. Has a default implementation. Runs in time.

`get-size``():Integer`
: Returns the number of elements in the list. Has a default implementation. Runs in time.

`get-nth``(`*`n`*`:Integer):item-type`
: Returns the nth element in the list, counting from 0. Has a default implementation. Runs in time.

`set-nth``(`*`n`*`:Integer, `*`new-value`*`:item-type)`
: Assigns a new value to the nth element in the list, counting from 0. Has a default implementation. Runs in time.

Bidirectional List Iterator<item-type> ADT

get-value():item-type
: Returns the value of the list element that this iterator refers to. Undefined if the iterator is past-the-end.

set-value(*new-value*:item-type)
: Assigns a new value to the list element that this iterator refers to. Undefined if the iterator is past-the-end.

move-next()
: Makes this iterator refer to the next element in the list. Undefined if the iterator is past-the-end.

move-previous()
: Makes this iterator refer to the previous element in the list. Undefined if the iterator refers to the first list element.

equal(*other-iter*:List Iterator<item-type>):Boolean
: True iff the other iterator refers to the same list element as this iterator.

All operations run in time.

Doubly Linked List Implementation

Vector Implementation

The vector we've already seen has a perfectly adequate implementation to be a Bidirectional List. All we need to do is add the extra member functions to it and its iterator; the old ones don't have to change.

```
type Vector<item-type>
  ... # already-existing data and methods
```

Implement this in terms of the original `insert-after()` method. After that runs, we have to adjust *iter*'s index so that it still refers to the same element.

```
  method insert(iter:Bidirectional List Iterator<item-type>, new-item:item-type)
    insert-after(new Vector-Iterator(iter.array, iter.index-1))
    iter.move-next()
  end method
```

Also implement this on in terms of an old function. After `remove-after()` runs, the index will already be correct.

```
  method remove(iter:Bidirectional List Iterator<item-type>)
    remove-after(new Vector-Iterator(iter.array, iter.index-1))
  end method
end type
```

Tradeoffs

In order to choose the correct data structure for the job, we need to have some idea of what we're going to *do* with the data.

- Do we know that we'll never have more than 100 pieces of data at any one time, or do we need to occasionally handle gigabytes of data ?
- How will we read the data out ? Always in chronological order ? Always sorted by name ? Randomly accessed by record number ?
- Will we always add/delete data to the end or to the beginning ? Or will we be doing a lot of insertions and deletions in the middle ?

We must strike a balance between the various requirements. If we need to frequently read data out in 3 different ways, pick a data structure that allows us to do all 3 things not-too-slowly. Don't pick some data structure that's unbearably slow for *one* way, no matter how blazingly fast it is for the other ways.

Often the shortest, simplest programming solution for some task will use a linear (1D) array.

If we keep our data as an ADT, that makes it easier to temporarily switch to some other underlying data structure, and objectively measure whether it's faster or slower.

Advantages / Disadvantages

For the most part, an advantage of an array is a disadvantage of a linked list, and vice versa.

- Array Advantages (vs. Linked Lists)

1. *Index* - **Indexed access to any element in an array is fast**; a linked list must be traversed from the beginning to reach the desired element.
2. *Faster* - In general, **accessing an element in an array is faster** than accessing an element in a linked list.

- Linked-List Advantages (vs. Arrays)

1. *Resize* - **A linked list can easily be resized by adding elements** without affecting the other elements in the list; an array can be enlarged only by allocating new chunk of memory and copying all the elements.
2. *Insertion* - **An element can easily be inserted into the middle of a linked list**: a new link is created with a pointer to the link after it, and the previous link is made to point to the new link.

Side-note: - **How to insert an element in the middle of an array**. If an array is not full, you take all the elements **after** the spot or index in the array you want to insert, and move them forward by 1, then insert your element. If the array is already full and you want to insert an element, you would have to, in a sense, 'resize the array.' A new array would have to be made one size larger than the original array to insert your element, then all the elements of the original array are copied to the new array taking into consideration the spot or index to insert your element, then insert your element.