CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College

September 25, 2017

Outline

1. Chapter 4: Linked Structures and Iterators
   - LList: A Linked Implementation of a List ADT
   - Iterators
   - Links vs. Arrays

Outline

1. Chapter 4: Linked Structures and Iterators
   - LList: A Linked Implementation of a List ADT
   - Iterators
   - Links vs. Arrays

Using the ListNode Class

Using the ListNode Class

The class `LList`, an Abstract Data Type which will provide the necessary interface operations for its objects to behave like lists will be ListNode's Only "customer".

Since no other class will use ListNode objects, we don't provide public accessors or mutators (get_item, get_link, set_item, set_link) for (private) ListNode attributes.

Rather, we allow LList to access the attributes directly via dot-notation.

Properties of the LList Class

Class Invariants

A Class Invariant of a class is a condition which must be true for the concrete representation of every instance (object) of that class. For the `LList` class, these are:

- `self.size` is the number of nodes currently in the list.
- If `self.size == 0` then `self.head` is `None`; otherwise `self.head` is a reference to the first `ListNode` in the list.
- The last `ListNode` (at position `self.size - 1`) in the list has its link set to `None`, and all other `ListNode` links refer to the next `ListNode` in the list.

Methods of the LList Class

__INIT__

```
def __init__(self, seq=()):
   if seq == ():
      self.head = None
   else:
      self.head = ListNode(seq[0], None)
      last = self.head
      for item in seq[1:]:
         last.link = ListNode(item, None)
         last = last.link
   self.size = len(seq)
```

Methods of the LList Class

__len__

```
def __len__(self):
    return self.size
```

Methods of the LList Class

_find

```
def _find(self, position):
    assert 0 <= position < self.size
    node = self.head
    for i in range(position):
        node = node.link
    return node
```

Methods of the LList Class

append

```
def append(self, x):
   newNode = ListNode(x)
   if self.head is not None:
      node = self._find(self.size - 1)
      node.link = newNode
   else:
      self.head = newNode
   self.size += 1
```

Methods of the LList Class

__GETITEM__

```
def __getitem__(self, position):
   node = self._find(position)
   return node.item
```

Methods of the LList Class

__SETITEM__

```
def __setitem__(self, position, value):
    node = self._find(position)
    node.item = value
```

Methods of the LList Class

__DELITEM__

```
def __delitem__(self, position):
    assert 0 <= position < self.size
    self._delete(position)
```

Methods of the LList Class

_delete

```
def _delete(self, position):
    if position == 0:
        item = self.head.item
        self.head = self.head.link
    else:
        prev_node = self._find(position - 1)
        prev_node.link = prev_node.link.link
    self.size -= 1
    return item
```

Methods of the LList Class

pop

```
def pop(self, i=None):
   assert self.size > 0 and (i is None or (0 <= i <
self.size))
   if i is None:
      i = self.size - 1
   return self._delete(i)
```

Methods of the LList Class

INSERT

```
def insert(self, i, x):
    assert 0 <= i <= self.size
    if i == 0:
        self.head = ListNode(x, self.head)
    else:
        node = self._find(i - 1)
        node.link = ListNode(x, node.link)
    self.size += 1
```

A Common Problem For any Container Class: Traversal

Iteration is an Abstraction of Traversal

Container classes can provide efficient access to their contents in various ways:

- random access indexed: (arrays, Python lists, dictionaries)
- sequential access: Linked Lists

A Common Problem For any Container Class: Traversal

Traversal Depends on Structure

To process a container class, each item must be visited exactly once. Different structures will do this differently.

- random access indexed:

```
n = len(lst)
for i in range(n):
    print(lst[i])
```

- sequential access: Linked Lists

```
node = myLList.head
while node is not None:
    print(node.item)
    node = node.link
```

A Common Problem For any Container Class: Traversal

Iteration is Traversal Without Seeing Internal Structure

A Design Pattern is a strategy which occurs repeatedly in object-oriented design.
The iterator pattern provides each container class with an associated iterator class, whose behavior is simply to produce each item exactly once in some sequence.

Iterators in Python

The Interface of an Iterator: next()

```
>>> from LList import *
>>> myList=[1,2,3]
>>> it=iter(myList)
>>> it.next()
1
>>> it.next()
2
>>> it.next()
3
Traceback (most recent call last):
File "<pyshell>", line 1, in <module>
it.next()
StopIteration
```

Iterators in Python

The Interface of an Iterator: the StopIteration exception

```
>>> while True:
        try:
            a = it.next()
        except StopIteration:
            break
        print(a)
1
2
3
```

Iterators in Python

The Interface of an Iterator: in

```
>>> for a in myList:
        print(a)
 1
 2
 3
```

Adding an Iterator to LList

An Iterator Class for LList

```
class LListIterator(object):
 def __init__(self, head):
    self.currnode = head
 def next(self):
    if self.currnode is None:
       raise StopIteration
    else:
       item = self.currnode.item
       self.currnode = self.currnode.link
       return item
```

Adding an Iterator to LList

__iter__ Method for LList Class

```
def __iter__(self):
 return LListIterator(self.head)
```

Adding an Iterator to LList

Python for loop

```
>>> from LList import *
>>> nums = LList([1, 2, 3, 4])
>>> for item in nums:
      print(item)
1
2
3
4
```

Iterating With A Python Generator

A Generator Object

A Generator Object has the same interface as an iterator.

- It is used whenever a computation needs to be stopped to return a partial result.
  (Just as an iterator stops after each item when traversing a list, and returns that item.)
- It continues the computation in steps when called repeatedly.
  (Just as an iterator continues its traversal of a container, returning successive items.)

Iterating With A Python Generator

A Generator Definition

A Generator Definition combines properties of a function definition with those of the `__init__` method of a class.

- It has the format of a function definition.
- Instead of `return` it uses `yield`, to indicate where a partial result is returned and the computation frozen until the next call.
- Like a constructor (`__init__`), it returns a generator object, which behaves according to the body of the definition.

Iterating With A Python Generator

Example: Generating A Sequence of Squares

```
def squares():
    num = 1
    while True:
        yield num * num
        num += 1
>>> seq = Squares()
>>> seq.next()
1
>>> seq.next()
4
>>> seq.next()
9
```

Iterating With A Python Generator

LList Iterator Reimplemented as Generator

```
class LList(object):
 ...
 def __iter__(self):
    node = self.head
    while node is not None:
       yield node.item
       node = node.link
```

Trade-offs When Storing Sequential Information

Costs and Benefits of Array Storage

- Fast random access.
- Slow insertion and deletion.
- Efficient memory usage for homogeneous data (no links to store).

Trade-offs When Storing Sequential Information

Costs and Benefits of Linked Storage

- Slow random access.
- Faster insertion and deletion.
- Requires more memory (link information). If each data item is small this may double the storage required.