

CS313E: Elements of Software Design

Another ADT: Linked Lists

Dr. Bill Young
 Department of Computer Sciences
 University of Texas at Austin

Last updated: November 23, 2011 at 06:34

A list is a finite sequence of elements:

$$A_1, A_2, \dots, A_n$$

Is this structure *homogeneous* or *heterogeneous*?

What operations are naturally performed on a list?

How does (or should) this affect the implementation?

Lists *are an abstract data type*, with a pretty rich interface.

Lists in Python

The analogue of an array in Python is a `list`. A list is *heterogenous* and *dynamic*. **What does that mean?** The size is not specified at creation and can grow and shrink as needed.

Python provides built-in functions to manipulate a list and its contents.

There are several ways in which to create a list:

- enumerate all the elements
- create an empty list and then append or insert items into the list.

Lists in Python (2)

To obtain the length of a list you can use the `len()` function.

```
a = [1, 2, 3]
length = len(a)           # length = 3
```

The items in a list are indexed starting at 0 and ending at index `length - 1`.

You can also slice a list by specifying the starting index and the ending index and Python will return to you a sub-list from the starting index and upto but not including the end index.

```
a = [1, 9, 2, 8, 3, 7, 4, 6, 5]
b = a[2:5]                 # b = [2, 8, 3]
```

Use `L[int expr]` to access the element at a given position. Use `L[start : end]` to slice for a sublist.

Here are some other useful methods:

- `L + L` list concatenation
- `L.append(x)` add element to the end of L
- `L.extend(lst)` add elements of lst to the end of L
- `L.insert(i, x)` insert x at i if in L, else at end
 - `L.pop()` remove and return the last element
 - `L.pop(i)` remove and return the element at i

`x in L` boolean membership test

- `L.index(x)` index of x in L, or error if doesn't appear
- `L.count(x)` count occurrences of x in L
- `L.remove(x)` remove first occurrence of x from L
- `L.reverse()` reverse L in place
- `L.sort()` sort L in place

List Operations

Which Operations Matter?

Let's think of lists more abstractly than just as a Python data type. What is the abstract interface:

<code>List()</code>	create a new List
<code>get(index)</code>	fetch the item at index
<code>add (item)</code>	add an item to the list
<code>remove (item)</code>	remove an item from the list
<code>search (item)</code>	see if the item is in the list
<code>isEmpty ()</code>	is the list empty?
<code>len ()</code>	how many items are in the list

<code>List()</code>	create a new List
<code>get(index)</code>	fetch the item at index
<code>add (item)</code>	add an item to the list
<code>remove (item)</code>	remove an item from the list
<code>search (item)</code>	see if the item is in the list
<code>isEmpty ()</code>	is the list empty?
<code>len ()</code>	how many items are in the list

A tradeoff is necessary between being minimalist and being useful. In many languages, lists are homogenous, but not in Python. **Why is that?**

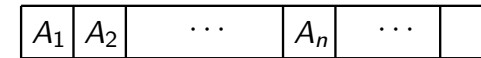
Which of these can be performed efficiently? Which of these are you likely to perform most often?

There are two common ways to implement a List:

- *Contiguous*: items are stored in an array structure, with successive elements stored in successive memory locations;
- *Linked*: items are stored in nodes, where each node also contains a pointer to the next node.

Note that arrays are not built into Python, but the underlying implementation of Python lists are arrays at the implementation level.

Our list A_1, A_2, \dots, A_n is stored in an array, as follows:



What's in the slots after A_n ? Does it matter?

Notice that there are many different array values that represent the same List. Thus, the *abstraction function* is not one to one.

The List Operations

How would you implement the various operations of our List interface with a contiguous implementation?

List()	create a new List
get(index)	fetch the item at index
add (item)	add an item to the list
remove (item)	remove an item from the list
search (item)	see if the item is in the list
isEmpty ()	is the list empty?
len ()	how many items are in the list

Which are efficient? Which aren't? By what measurement? Under what conditions does this implementation make sense?

An Aside on Efficiency

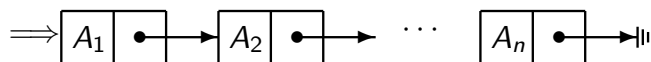
When we measure the *efficiency* of operations on a particular data structure, it is usually in terms of the size of the data structure.

The time required to perform an operations might be:

- **constant**: independent of the size of the structure;
- **linear**: is some constant k times the size of the structure;
- **quadratic**: relating to the square of the size of the structure;
- **exponential**: is some constant raised to a power related to the size of the structure.

For example, suppose that you have a list of n items, and you want to ask about the length of the list. This can be constant or linear, depending on the implementation.

A linked list is a collection of *nodes* chained together with pointers. Each node contains an element of the list and a pointer to the following node.



The final node *may* have a null pointer (None). There may also be a *header* node.

Linked lists are composed of nodes, defined as follows:

```
class Node:
    def __init__(self, initdata):
        self._data = initdata
        self._next = None

    def getData(self):
        return self._data

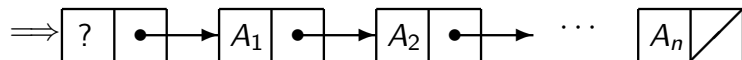
    def getNext(self):
        return self._next

    def setData(self, newdata):
        self._data = newdata

    def setNext(self, newnext):
        self._next = newnext
```

Header Node

Having a header node eliminates a lot of special case processing.



What's an example of that?

The Operations

How would you implement the various operations of our List interface with a linked implementation?

List()	create a new List
get(index)	fetch the item at index
add (item)	add an item to the list
remove (item)	remove an item from the list
search (item)	see if the item is in the list
isEmpty ()	is the list empty?
len ()	how many items are in the list

Which are efficient? Which aren't? By what measurement?

```

class UnorderedList:
    def __init__(self):
        self._head = None

    def __str__(self):
        output = "[ "
        ptr = self._head
        while ptr != None:
            output += str(ptr.getData()) + " "
            ptr = ptr.getNext()
        return output + "]"

    def get(self, index):
        if index < 0 or index >= self._length:
            print ("Index out of range.")
            return None
        cursor = self._head
        for i in range(index):
            cursor = cursor.getNext()
        return cursor.getData()

    def isEmpty(self):
        return self._head == None

```

```

    def add(self, item):
        temp = Node(item)
        temp.setNext(self._head)
        self._head = temp

    def remove(self, item):
        # This doesn't assume the item is in the list.
        current = self._head
        previous = None
        found = False
        while not found and current != None:
            if current.getData() == item:
                found = True
            else:
                previous = current
                current = current.getNext()
        if current == None:
            return
        elif previous == None:
            self._head = current.getNext()
        else:
            previous.setNext(current.getNext())

```

```

    def search(self, item):
        current = self._head
        while current != None:
            if current.getData() == item:
                return True
            else:
                current = current.getNext()
        return False

    def length(self):
        current = self._head
        count = 0
        while current != None:
            count += 1
            current = current.getNext()
        return count

```

The midterm covered material to this point.

Given a linked list L, it would be nice to be able to write:

```
for x in L: ....
```

You can do that if you define an *iterator* over the class. An iterator is a funny type of method that yields a different value each time it is called.

```
def __iter__(self):
    cursor = self._head
    while True:
        if cursor is None:
            raise StopIteration
        yield cursor.data
        cursor = cursor.next
```

From the Python documentation:

The yield statement is only used when defining a generator function, and is only used in the body of the generator function.

When a yield statement is executed, the state of the generator is frozen and the value of expression list is returned to next()'s caller. By frozen we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, and the internal evaluation stack: enough information is saved so that the next time next() is invoked, the function can proceed exactly as if the yield statement were just another external call.

Yield Example

The following function gives all permutations of a list.

```
def all_perms(lst):
    """This function generates all permutations of the
    input list."""
    if len(lst) <= 1:
        yield lst
    else:
        for perm in all_perms(lst[1:]):
            for i in range(len(perm)+1):
                yield perm[:i] + lst[0:1] + perm[i:]
```

Why do this with yield, rather than just the standard list mechanisms? If you have a list of n elements, you'll have $n!$ permutations. Maybe you only want to generate as many as you need "on the fly."

Using Permutations

Challenge from a friend of mine: find a permutation of the digits from 1 .. 9 and use each to replace a single x in the following, to satisfy this equation: $x/xx + x/xx + x/xx = 1$.

```
count = 0
tStart = time.clock()
for p in all_perms([ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]):
    n1 = p[0]; d1 = p[1] * 10 + p[2]
    n2 = p[3]; d2 = p[4] * 10 + p[5]
    n3 = p[6]; d3 = p[7] * 10 + p[8]
    ans = ( n1 / d1 ) + ( n2 / d2 ) + ( n3 / d3 )
    count += 1
    if ( ans == 1 ):
        print( "Found it: ", end = "" )
        print( n1, d1, n2, d2, n3, d3 )
        break
tEnd = time.clock()
interval = tEnd - tStart
print( "Tried " + str(count) + " permutations")
print( "Took = " + str(interval) + " seconds to execute")
```

Here's the output of my program:

```
felix:~/cs313e/python> python permutations.py
Found it: 5 34 7 68 9 12
Tried 15767 permutations
Took = 0.11 seconds to execute
```

If I hadn't stopped after the first one, it would have found several other solutions. [Can you guess how many and what they are?](#)

Notice that I only generated 15767 permutations out of a possible 363880. If I hadn't used a yield statement, I probably would have generated this much larger set before testing even the first one!

Another “magic method” is `__getitem__`. It is associated with indexing into a structure. If you define:

```
def __getitem__(self, index):
    ...
```

You can then use the `S[i]` syntax to get an element of the structure. For example, in the `UnorderedList` class:

```
def __getitem__(self, index):
    if index < 0 or index >= self._len:
        print ("Index out of range.")
        return None
    cursor = self._head
    for i in range(index):
        cursor = cursor.getNext()
    return cursor.getData()
```

```
>>> from RadixSort import *
>>> ul = UnorderedList()
>>> ul.add(3)
>>> ul.add(2)
>>> ul.add(4)
>>> print(ul)
[ 4 2 3 ]
>>> ul[1]
2
>>> ul[0]
4
>>> ul[3]
Index out of range.
>>> for index in ul: print (index)
...
4
2
3
```

Suppose we want our list to be *ordered*. As usual, you have to have an ordering relation defined on the node value types.

[How do you do that?](#) Define ordering functions such as the `__lt__` function. If you can compare two elements of the type, you can sort them.

Note that the ordering relation is not defined in the `OrderedList` class, but in the types of things being ordered.

```

class OrderedList(UnorderedList):

    def __init__(self):
        UnorderedList.__init__(self)

    def search(self, item):
        """Boolean valued search function."""
        current = self._head
        while current != None:
            if current.getData() == item:
                return True
            else:
                if current.getData() > item:
                    return False
                else:
                    current = current.getNext()
        return False

```

```

def add(self, item):
    """Add an item at the right spot
    in a sorted list."""
    current = self._head
    previous = None
    while current != None:
        if current.getData() > item:
            break
        else:
            previous = current
            current = current.getNext()
    temp = Node(item)
    if previous == None:
        temp.setNext(self._head)
        self._head = temp
    else:
        temp.setNext(current)
        previous.setNext(temp)

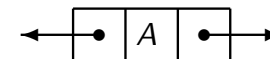
```

You might have an application where you always want to add at the end of the list rather than at the beginning. You could then maintain an additional pointer to the last element.

As an exercise, add this functionality and rewrite the List ADT.

The standard linked list allows you only to move in one direction. For example, to find the “predecessor” of a node, you have to start from the beginning.

If it is often important to go “backwards” in the list, use a doubly linked list. Each node in the list has both a forward and a backward pointer, with the pointers at each end given the appropriate null values



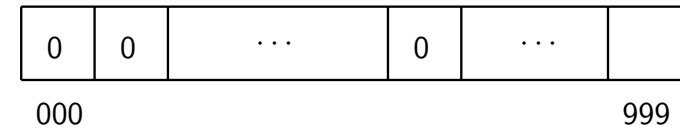
The gain in convenience comes at the expense of storage. Such tradeoffs are common in ADTs.

Another approach is to make the list circular. That is, the next pointer in the last element points to the front of the list, rather than null. This is very convenient for many applications.

However, it requires careful programming to avoid getting into an infinite loop. Eg. how do you search a circular list for a given element?

In general, whether to use a singly-linked list, doubly-linked list, or circular list depends on the application and the constraints of the implementation (i.e., is space more important than time). [Does the interface differ?](#)

If you know that all elements lie in a small enough range, simply declare an array indexed by that range with each slot initialized to zero.



Read the input list: x_1, x_2, \dots, x_N . For each x_i , increment the corresponding element $A[x_i]$ of the array. Then simply read the array elements in order and, for each index i , print that index $A[i]$ times.

Notice that this algorithm, usually called *Bucket Sort*, works only because we know some special characteristic of the input set.

Bucket Sort is very *time efficient* (linear) if you're using an array (or an array-based list structure), but may require a lot of space. Consider sorting 100,000 social security numbers using Bucket Sort.

[If you were using a linked list structure rather than an array, what implications would this have for the efficiency of this algorithm?](#)

A related algorithm, Radix Sort, works on integers in a certain range and may be much more *space efficient*. It uses an array of (initially empty) linked lists.

In the simplest version, each *pass* of Radix Sort sorts the list according to one radix position, starting with the least significant. There are as many passes as the maximum number of radix positions in any datum.

For the decimal version, declare an array of linked lists indexed from $0 \dots 9$. Read the input values and place each in turn onto the list at the index represented by the least significant digit.

Eg. Suppose the input is: 64, 216, 512, 27, 729, 0, 1, 343, 125, 42.
We know there will be three passes. After the first pass we have:

index	0	1	2	3	4	5	6	7	8	9
list	0	1	512 42	343	64	125	216	27		729

After the first pass the data is sorted by the least significant digit.
After the n th pass, by the n least significant digits. To maintain this we must be careful to maintain the order of the elements thus far.

For Pass 2, we read the data in order from the Pass 1 array, left to right and in order on any lists. Put each element onto the initially empty Pass 2 array according to the second least significant (10's) digit, or 0 if there is none. The result is:

index	0	1	2	3	4	5	6	7	8	9
list	0	512	125		42		64			
	1	216	27 729		343					

Notice that the array is sorted according to the last two digits.

In the third and last pass, we concentrate on the third most significant (100's) digit

index	0	1	2	3	4	5	6	7	8	9
list	0	125	216	343		512		729		
	1									
	27									
	42									
	64									

We stop since no number has more than three digits. We read off the sorted result left to right, and following the structure of the lists.

Suppose you wanted to sort 9-digit numbers. You could use 9 passes on an array of length 10. Alternatively, you could use one pass on an array of length 1,000,000,000. (See the similarity to Bucket Sort).

Instead, we can use an intermediate strategy. We'll make 3 passes with an array of length 1000. Each pass consumes 3 digits at a time. How would this work?

Which approach is most efficient? with respect to time? with respect to space?