CS313E: Elements of Software Design

Yet Another ADT: Lists

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

Last updated: October 25, 2012 at 13:48

The List ADT

A list is a finite, ordered sequence of elements:

$$[A_1, A_2, ...A_n]$$

Some questions to ask about any List:

- 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

A list in Python is *heterogenous* and *dynamic*. What does that mean?

Lists in Python

A list in Python is *heterogenous* and *dynamic*. What does that mean?

The list can contain elements of different types. 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 Python list:

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

Lists in Python (2)

To obtain the length of a list you can use the `len()` function. What does that suggest to you about the definition of lists?

```
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 up to but not including the ending index.

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

List Methods

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 x 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 i $<$ len(L), else at end
L.pop()	remove and return the last element
L.pop(i)	remove and return the ith element

List Methods

x in L	boolean membership test
L.index(x)	index of first x in L, error not present
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

Implementations

There are two common ways to implement a List:

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

The List data type in Python is implemented (in C) via resizable arrays. That is, they use the contiguous implementation.

Contiguous Implementation

Our list $[A_1, A_2, ...A_n]$ is stored in an array, as follows:

That's not exactly right. What is actually stored in the slots? 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.

Aside: Arrays

An *array* is just a list of items stored in successive addresses in memory. All of the items must be of the same size. Why does that matter? Why is it easy to ensure in Python?

Why is an array convenient for many purposes?

Aside: Arrays

An *array* is just a list of items stored in successive addresses in memory. All of the items must be of the same size. Why does that matter? Why is it easy to ensure in Python?

Why is an array convenient for many purposes?

Because you can access an arbitrary element of the array in *constant* time. If the array begins at address `B` and each element is of size `S`, then the ith element is at address `B + S*i`. Why?

The List Operations

Below is a potential abstract interface for a List ADT. 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
insert (item, index)	add an item at index
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? Does it depend on the implementation? Under what conditions does the contiguous 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 items, and you want to compute the length of the list. This operation *might be constant or linear*, depending on the implementation.

Efficiency of Contiguous List Operations

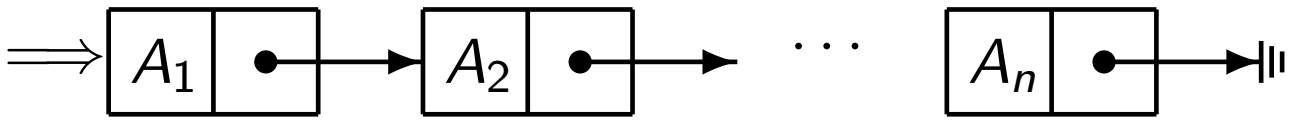
List()	create a new List
get(index)	fetch the item at index
add (item)	add an item to the list
insert (item, index)	add an item at index
remove (item)	remove an item from the list
search (item)	see if the item is in the list
isEmpty ()	is the list empty?
len ()	how many items are in the list

Given a contiguous List implementation, which operations are constant time, linear, quadratic, exponential? What does that say about when this implementation is appropriate?

Linked Lists

Another common way to implement Lists is by *Linked Lists*. These have *the same interface* but different efficiency properties from a contiguous list 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.

Python Node Class

Linked lists are composed of nodes, defined as follows:

```
class Node:
    """A node in a simple, singly linked structure.
       Each node has a data value and next pointer."""

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

    # Accessors (getters)

    def getData(self):
        return self._data
    def getNext(self):
        return self._next

    # Mutators (setters)

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

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
insert (item, index)	add an item at index
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 of these operations are efficient? Which aren't? By what measurement?

Linked Lists

In general, in a linked structure you can't access an arbitrary element unless you "traverse" the list from one end. So why would anyone use them?

- They use memory rather efficiently; you only allocate storage for the nodes used.
- Storage can be allocated anywhere in memory, making memory management easier.
- Many operations are quite efficient. You may choose a linked structure if you seldom do expensive operations.
- Sequential access is fast; random access is slow.

Unordered List Implementation

```
class UnorderedList:
    """A linked list in which the items are unordered."""
    def __init__(self):
        """Create an empty list of no items and
           length 0."""
        self._head = None
        self._length = 0

    def isEmpty(self):
        return not self._head

    def __str__(self):
        """Adding all items to the printstring requires
           traversing the list."""
        output = "[ "
        ptr = self._head
        while ptr != None:
            output += str(ptr.getData()) + " "
            ptr = ptr.getNext()
        return output + "]"
```

Unordered List (continued)

Get exemplifies traversing a linked List.

```
# Continues the UnorderedList class

def get(self, index):
    """Fetch the item at the indexed position."""
    # Is the index within bounds?
    if index < 0 or index >= self._length:
        print ("Index out of range.")
        return None
    # Count down the list until you reach the
    # right node.
    cursor = self._head
    for i in range(index):
        cursor = cursor.getNext()
    return cursor.getData()

def add(self, item):
    # Add an item to the front of the list.
    temp = Node(item)
    temp.setNext( self._head )
    self._head = temp
    self._length += 1
```

Unordered List ADT (continued)

```
# Continues the UnorderedList class

def remove(self, item):
    """Remove first occurrence of item, if any."""
    current = self._head
    previous = None
    found = False
    while not found and current != None:
        if current.getData() == item:
            self._length -= 1
            found = True
        else:
            previous = current
            current = current.getNext()
    if current == None:
        # item is not in the list
        return
    elif previous == None:
        # item is at head of the list
        self._head = current.getNext()
    else:
        previous.setNext(current.getNext())
```

Why is remove so much harder than add?

Unordered List ADT (continued)

```
# Continues the UnorderedList class

def search(self, item):
    """Return a boolean indicating whether
       item is in the list."""

    current = self._head
    # Search to find item or fall off the end.
    while current != None:
        if current.getData() == item:
            return True
        else:
            current = current.getNext()
    # If you reach the end of the list.
    return False
```

Efficiency of List Operations

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

For each of these List operations, do you think they have efficiency that is constant, linear, quadratic or exponential in the linked implementation? In the contiguous implementation?

Ordered Lists

Suppose you want to define an OrderedList type. It could be a subtype of UnorderedList. Which of the standard methods would you inherit and which would you override? Would there be any new methods?

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

Ordered Lists

```
class OrderedList(UnorderedList):

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

    def search(self, item):
        """Boolean valued search function."""
        current = self._head
        # search until we find it or fall off the end
        while current != None:
            if current.getData() == item:
                # item has been found
                return True
            else:
                if current.getData() > item:
                    # We've passed where the item could be.
                    # Only works for ordered lists.
                    return False
                else:
                    current = current.getNext()
        return False
```

Ordered Lists

```
def add(self, item):
    """Add an item at the right spot
    in a sorted list."""
    # must keep two pointers marching
    # in synch down the list.
    current = self._head
    previous = None
    while current != None:
        if current.getData() > item:
            # we've reached the insertion spot
            break
        else:
            # otherwise, advance both pointers
            previous = current
            current = current.getNext()
    temp = Node(item)
    if previous == None:
        # insert at the start of the list
        temp.setNext(self._head)
        self._head = temp
    else:
        temp.setNext(current)
        previous.setNext(temp)
```

Ordered Lists

We should have asked: "ordered with respect to what relation?"
Notice that we used $>$ to compare values in the list. We can do that for any type of values for which we have an ordering relation defined. How do you do that?

Ordered Lists

We should have asked: "ordered with respect to what relation?"
Notice that we used $>$ to compare values in the list. We can do that for any type of values for which we have an ordering relation defined. How do you do that?

Recall that you can treat *any* type as ordered if you define ordering functions such as the `__lt__` function. If you can compare two elements of the type, you can order them.

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

Magic Method: An Iterator

Given a linked list L (ordered or unordered), 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
```

Aside: About Yield

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")
```

The Result

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 15,767 permutations out of a possible 363,880. 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: __getitem__

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()
```

Notice this is just the get function we already defined.

Some Examples

```
>>> 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 x in ul: print( x )
...
4
2
3
```

Special Lists

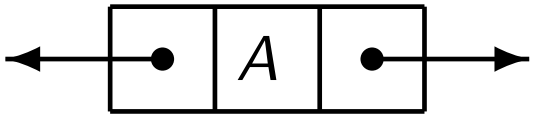
You might have an application where you always want to add at the end of the list rather than at the beginning. You could then maintain an additional pointer to the last element.

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

Doubly Linked List

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, you can use a doubly linked list. Each node has *both* a forward and a backward pointer; nodes at each end are given appropriate null pointer values



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

Circular Lists

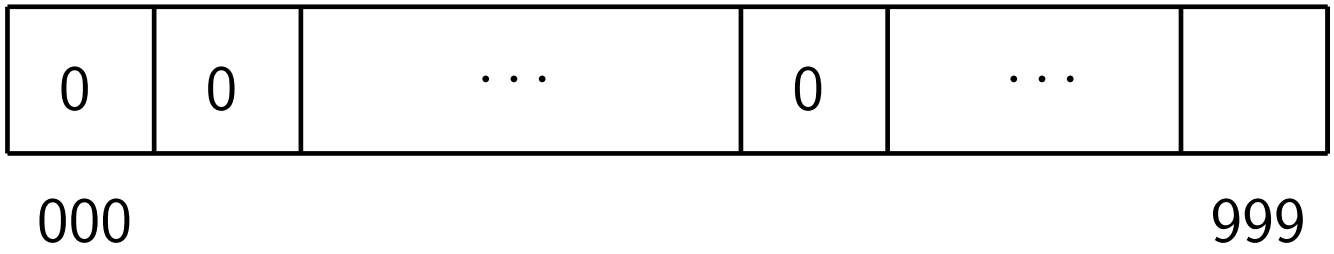
Another approach is to make the list circular. That is, the next pointer in the last element points to the front of the list, rather than null. This is very convenient for many applications.

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?

Bucket Sort

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: $[x_1, x_2, \ldots, x_N]$. 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.

This algorithm, often called *Bucket Sort*, works only because of a special characteristic of the input set. What is it? It is not a *general purpose* sort algorithm.

Radix Sort

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.

Radix Sort

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 maximimum number of radix positions in any datum.

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

Radix Sort

Eg. Suppose the input is: 64, 216, 512, 27, 729, 0, 1, 343, 125, 42. We know there will be three passes. Why? 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 nth pass, by the n least significant digits. To maintain this we must be careful to maintain the order of the elements thus far.

Radix Sort (cont.)

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		343					
			729							

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

Radix Sort (cont.)

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.

Radix Sort

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? How might you choose which is most important?