Computer Science I
Linked Lists

CSCI-141
Lecture

11/02/2013

1 Problem

Python's built-in list library provides a rich set of functionality for manipulating a linear collection of data. However, it does suffer from some performance problems when inserting and removing from the middle/end of the collection, as well as potentially "unused" memory space. We will develop our own list data structure to solve this problem.

2 Solution Design and Analysis

Linked Lists

You have previously seen stacks and queues. The *linked list* is an extension of a stack or a queue, and it can be implemented in much the same way, except that it allows for frequent insertions and deletions from the middle/end of the list are required. Also, a linked list is more appropriate than a Python list if the structure is constantly changing and the size of the list might grow or shrink arbitrarily. For example, a linked list will make more efficient use of available memory because it allocates memory only on an element-by-element basis.

2.1 List Representation: Arrays vs. Linked Lists

A list is one of the simplest and most frequently used data structures in computer science. All of the data structures that we will study may be reduced to some kind of list-based representation.

In computer science, a list is represented as a sequence of elements (i.e. data we wish to store), where the elements may be arbitrarily complex. Most often, a list is an integral part of a language library of data structure collections. It provides common operations that manipulate elements of any specified type (e.g. numbers, strings, or objects).

Below are some basic list operations that Python supports.

List Operations

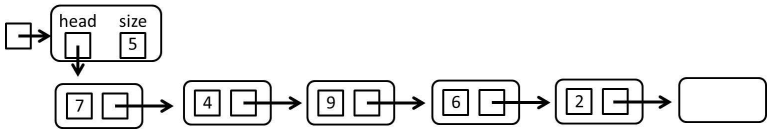
1. Append an element to the end of the list (e.g. using list's `append` method)
2. Insert an element by position into the list (e.g. using list's `insert` method)
3. Remove an element by position from the list (e.g. using list's `pop` method)
4. Access an element at a position (e.g. using list's index operator, `[]`)
5. Search for an element by value (e.g. using list's `index` method)

Note that the run-time complexity of these operations depends upon the underlying list representation used.

The built-in Python 'list' class is implemented using *arrays*. An array is a sequence of adjacent memory locations, and the array variable name refers to the location of the first element.

The indexing operation (square brackets, i.e. $a[j]$) represents the j^{th} element past the start of the array. Arrays can access arbitrary elements by index in $O(1)$ time because the memory is contiguous and the location can be computed by adding an offset.

Another common list representation is a *linked list*. This representation uses a *list node* to store an element, and a link (also called a *reference*) to the next node in the list. The figure below shows a linked list instance consisting of five nodes. The first node in the list must be referenced by an external variable, called the *head*, so that we have a means of entry into the linked list. The other nodes in the list are not named and are accessed via the link field of the preceding node. A linked list can also be empty, which is indicated when the head references an *empty list node*. It is also convenient to keep track of the number of elements in the list using a `size` attribute.



To access or find an element in a linked list, we must traverse the list starting from the head of the list. The time complexity for accessing elements is therefore $O(N)$, as opposed to $O(1)$ for accessing any element in an array. Searching through a linked list is also $O(N)$ because we must traverse from the head of the list. In contrast, if we sort the elements into an array representation, we can perform binary search, which has $O(logN)$ time complexity.

On the plus side, insertion and deletion may be faster using a linked list than an array. For example, when we delete an element from an array, all remaining values in the array must be copied back (shifted) to maintain the list structure. This makes array element deletion an $O(N)$ operation, because all remaining elements must be copied to the previous location when the first element is deleted. In a linked list however, if we are deleting the first element in the list, we can simply make the head of the list point to the node following the first node. If we only ever removed the head, this would be a stack implementation. In general, deletion remains $O(N)$ for a linked list because we must traverse the list to find the element we want to delete. In practice, though, traversing a linked list and deleting an element may be less expensive than copying elements in an array.

Inserting an element into an array representation is also $O(N)$. In the worst case, if we insert a value into the first array location, we must first copy all the elements in the array forward to make space for the new element. In contrast, if we already have navigated to the location of the value to remove from a linked list, then both insertion and deletion are $O(1)$.

Another difference between arrays and linked lists is their storage requirements. Arrays are created with a *static*, fixed initial size, while linked lists vary in size *dynamically*. When an array is allocated in memory, a fixed number of memory locations are reserved. If the number of elements grows past the array's capacity, the entire array must be copied into a new, larger array in memory, an $O(N)$ operation. In contrast, creating a node and setting its link is inexpensive; finding the insertion point requires an $O(N)$ traversal, but creating and initializing a node is $O(1)$, regardless of the list size.

Generally speaking, if one has a fixed-size list that is expected to be accessed frequently, an array is the better representation to use. However, if the list will frequently change, a linked list representation is preferable for the reasons given above.

2.2 Linked List Implementation

We will use several classes for the linked list implementation. The first set of classes will represent the nodes in the linked list. You should remember from the Stack and Queues lecture that we will want to make the distinction between a node class that contains data, `Node`, and a node class that does not, `EmptyNode`. (See `myNode.py` for full details.)

```
class EmptyNode():
    __slots__ = ()

class Node():
    __slots__ = ('data', 'next')
```

These are the builders for each new class type.

```
def mkEmptyNode():
    return EmptyNode()

def mkNode(data, next):
    node = Node()
    node.data = data
    node.next = next
    return node
```

Recall, that to determine whether or not an object is of a certain class type, we will use the built-in boolean function, `isinstance`.

```
nodeA = mkEmptyNode()
nodeB = mkNode('a', mkEmptyNode())
if isinstance(nodeA, EmptyNode):
    print('nodeA is an EmptyNode')
if not isinstance(nodeB, EmptyNode):
    print('node B is not an EmptyNode')
```

With these two node class definitions, we can construct our first linked list. For example, the code below builds a linked list, `letters`. If it were a Python linked list, `print( letters )` would display `['a', 'b', 'c']`.

```
letters = mkNode('a', mkNode('b', mkNode('c', mkEmptyNode())))
```

Lists exhibit the same recursive structure as strings. In the example above, we will refer to the node containing `'a'` as the *head* of the list, and the sub-list, `['b', 'c']` as the *tail*.

While this is a perfectly fine representation for a linked list, there are certain advantages to combining similar properties of the list, like its length, into another class. In programming, this is referred to as *encapsulation*. We make a new class, `MyList` that encapsulates both the head and current number of elements in the list.

```
class MyList():
    """A class that encapsulates a node based linked list"""
    __slots__ = ('head', 'size')

def mkMyList():
    lst = MyList()
    lst.head = mkEmptyNode()
    lst.size = 0
    return lst
```

We will develop both an iterative, `myListIter.py`, and a recursive, `myListRec.py`, implementation of a linked list structure. Each implementation contains the five basic, linked list routines identified earlier, plus some other useful, supporting functions.

NOTE: this code will serve as the basis for this topic's assignments.

3 Implementation

3.1 Append

We'll implement the first function, `append`, to provide an example of how the iterative and recursive implementations work.

Each implementation requires the notion of a `current` *pointer*, a variable of either `Node` or `EmptyNode` type, that refers to the current node in the list. This pointer starts at the head of the list and advances until the last element.

These are the two cases to address when appending a new element to a list:

- The list is currently empty (i.e. `lst.data` is an instance of `EmptyNode`). If that is the case, the code must create the new node and set the list's head to point to the new node.)
- The list is non-empty (i.e. `lst.data` is not an instance of `EmptyNode`). If that is the case, the code must advance the `current` node pointer from the head to the last node in the list. After the pointer reaches the last non-empty element (i.e. `current.next` is an instance of `EmptyNode`), the code can create the new node and *link* the current node's *next* to the new node.

Here is the implementation expressed recursively:

```
def append(lst, value):
    if isinstance(lst.head, EmptyNode):
        lst.head = mkNode(value, EmptyNode())
    else:
        appendRec(lst.head, value)

    lst.size += 1

def appendRec(node, value):
```

```
        if isinstance(node.next, EmptyNode):
            node.next = mkNode(value, EmptyNode())
        else:
            appendRec(node.next, value)
```

Notice with the recursive implementation, we implement a separate recursive function. This is because `append` operates on a `MyList`, and we really want another function that can work recursively with nodes (e.g. `Node` or `EmptyNode`). If the list is non-empty, we invoke the recursive function with the head node and recurse until we reach the last node in the list.

Here is the implementation expressed iteratively:

```
    def append(lst, value):
        if isinstance(lst.head, EmptyNode):
            lst.head = mkNode(value, EmptyNode())
        else:
            curr = lst.head
            while not isinstance(curr.next, EmptyNode):
                curr = curr.next
            curr.next = mkNode(value, EmptyNode())

        lst.size += 1
```

There is no need to write a separate function in the iterative implementation because we can declare a local variable, `curr`, to initially point to the head of the list, and use a `while` loop to iterate the pointer through the node links.

3.2 InsertAt

The next function, `insertAt` will be different in that it needs to modify several different pointers in order to do the insert. InsertAt takes 3 parameters, the first is the list itself, followed by where you want the item inserted, and the item to insert.

With insert, we need to initial check the bounds of the insertion location to ensure it is at least 0, but doesn't exceed the size of the list. This is a key reason that encapsulation is a useful implementation choice.

From there, we have 2 choices, either the insertion is at position 0, and we need to adjust the head, or it's somewhere else.

When the index location is not the head, we will decrement the index location until we reach 0, then update the next pointers on the nodes.

Here is the implementation expressed recursively:

```
    def insertAt(lst, index, value);
        if index < 0 or index > lst.size:
            raise IndexError(str(index) + ' is out of range.')
        if index == 0:
            lst.head = mkNode(value, lst.head)
```

```
        else:
            insertRec(lst.head, index-1, value)
        lst.size += 1

    def insertRec(node, index, value):
        if index == 0:
            node.next = mkNode(value, node.next)
        else:
            insertRec(node.next, index-1, value)
```

Notice again, how with the recursive implementation, we implement a separate recursive function.

Here is the implementation expressed iteratively:

```
    def insertAt(lst, index, value):
        if index < 0 or index > lst.size:
            raise IndexError(str(index) + ' is out of range.')
        if index == 0:
            lst.head = mkNode(value, lst.head)
        else:
            prev = lst.head
            while index > 1:
                prev = prev.next
                index -= 1
            prev.next = mkNode(value, prev.next)
        lst.size += 1
```

To understand better how both the iterative and recursive forms operate, study the `myListIter.py` and `myListRec.py` files. These implement the basic list operations identified earlier: append, remove, access, and search for a value.

4 Testing

We should test the list by creating an empty list, a list with one element, two elements, and a larger number of elements. We should also traverse and print an empty list and a list with many elements. When we are confident that these functions work correctly, we should attempt to remove elements from the list, check the size of the list, find the indexes of specific list elements, and get an element at a specific index. After we are sure that these functions all work as expected, then we can use them as a linked list function library.

See the `testList.py` file for an exhaustive test suite for our linked list. To test the iterative version, change the import statement to replace `myListRec` with `myListIter`. It is a good idea to run the tests after developing each function.

5 Time Complexity

The time complexity for linked lists should be straight forward. In nearly every operation, we have the potential to iterate through every item in the list to find the one we are looking for. Append must go through all the items (as it appends to the end), and it is $O(N)$. InsertAt time complexity based on the position of the insert, but since that can again, be at the end, it is still $O(N)$. The other common function is Remove, which again takes an index, so it will behave much like insert.

Since linked lists are $O(N)$ for every operation, why would you ever use them over an array? Simple, they can grow, easily.