Ch. 4 Data Structure

Tuesday, April 28, 2020 6:13 PM

Fundamentals Ch. 4

- Checking to see whether a key is in the dictionary or not is also O(1
- Get item and set item operations on a dictionary are O(1)

Table 3: Big-O Efficiency of Python Dictionary Operations

operation	Big-O Efficiency
copy	O(n)
get item	O(1)
set item	O(1)
delete item	O(1)
contains (in)	O(1)
iteration	O(n)

4 Data Structure

Content

- Stack

Stack: "an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end."

- Last in First Out
- Staks are fundamentally important since they reverse the order of items
- i.e. Your browser has a back button to retrieve the last (most recent p age)

For example, if `s` is a stack that has been created and starts out empty, then Table 1 shows the results of a sequence of stack operations. Under stack contents, the top item is listed at the far right.

Table 1: Sample Stack Operations

Stack Operation	Stack Contents	Return Value
`s.isEmpty()`	`[]`	`True`
`s.push(4)`	`[4]`	
`s.push('dog')`	`[4,'dog']`	
`s.peek()`	`[4,'dog']`	`'dog'`
`s.push(True)`	`[4,'dog',True]`	
`s.size()`	`[4,'dog',True]`	`3`
`s.isEmpty()`	`[4,'dog',True]`	`False`
`s.push(8.4)`	`[4,'dog',True,8.4]`	
`s.pop()`	`[4,'dog',True]`	`8.4`
`s.pop()`	`[4,'dog']`	`True`
`s.size()`	`[4,'dog']`	`2`

In [2]:
```
# Example of Implementing a Stack: Simple Parenthesis Balance

# Initializing and Creating Data Structure Class, Stack
class Stack:
     def __init__(self):
         self.items = []

     def isEmpty(self):
         return self.items == []

     def push(self, item):
         self.items.append(item)

     def pop(self):
         return self.items.pop()

     def peek(self):
         return self.items[len(self.items)-1]

     def size(self):
         return len(self.items)


# Parenthesis Checker Function
def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol == "(":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                s.pop()

        index = index + 1

    if balanced and s.isEmpty():
        return True
    else:
        return False

print(parChecker('((()))'))
print(parChecker('(()'))
```

```
True
False
```

Infix, Prefix and Postfix Expressions

Infix: operators in between expressions/operands (i.e. *, +)

```
* Infixes have a difference precedence level and () is the only thing tha
t can change that
* Fully parenthesized expressions leave no room for ambiguity and needs n
ot to remember any precedence rules
```

Prefix: A notation that has the operator precede the two operands they work on (i.e. + A * B C) and you perform the functions

Postfix: A notation that has the operators coming after all the corresponding operands (i.e. A B C +*), denoting the sum of A and B*C. Imagine working inside out, with the operation touching performing a function on the two closest variables first.

Prefix and Postfix can do away with parenthesis because the order of operations are completely dictated by the position of the variables and symbols

Table 4: Additional Examples of Infix, Prefix, and Postfix

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

4.10 Queue

A **queue** is an ordered collection of items where the addition of new items happens at one end, called the "rear," and the removal of existing items occurs at the other end, commonly called the "front."

This leads to a First in, First out ordering principle (FIFO)

```
* Keyboard functions (each letter must display on the screen in the order
we type them in
```

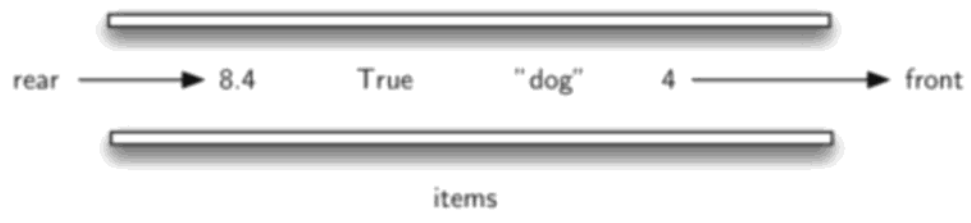

Figure 1: A Queue of Python Data Objects

Queue Abstract Data Type

- Queue() creates a new queue that is empty. It needs no parameters and returns an empty queue.
- enqueue(item) adds a new item to the rear of the queue. It needs the item and returns nothing.
- dequeue() removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
- isEmpty() tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the queue. It needs no parameters and returns an integer

Table 1: Example Queue Operations

Queue Operation	Queue Contents	Return Value
`q.isEmpty()`	`[]`	`True`
`q.enqueue(4)`	`[4]`	
`q.enqueue('dog')`	`['dog',4]`	
`q.enqueue(True)`	`[True,'dog',4]`	
`q.size()`	`[True,'dog',4]`	`3`
`q.isEmpty()`	`[True,'dog',4]`	`False`
`q.enqueue(8.4)`	`[8.4,True,'dog',4]`	
`q.dequeue()`	`[8.4,True,'dog']`	`4`
`q.dequeue()`	`[8.4,True]`	`'dog'`
`q.size()`	`[8.4,True]`	`2`

In [6]:
```
# Implementing a Queue on Python

class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)

# A hot potato queue implementation where kids are dequeued from the front, only

def hotPotato(namelist, num):
    simqueue = Queue()
    for name in namelist:
        simqueue.enqueue(name)

    while simqueue.size() > 1:
        for i in range(num):
            simqueue.enqueue(simqueue.dequeue())

        simqueue.dequeue()

    return simqueue.dequeue()

print(hotPotato(["Bill","David","Susan","Jane","Kent","Brad"],7))
```

```
Susan
```

Read Printer Queue Simulation to learn about using Queues to Simulate Real World Problems

Printer Queue Simulation (https://runestone.academy/runestone/books/published/pythonds/BasicDS/SimulationPrintingTasks.ht

4.15 Deque

A double-ended queue that by default has a rear and a front but is unrestrictive nature of adding and removing items, and therefore has both the properties of stacks and queues in a single data structure

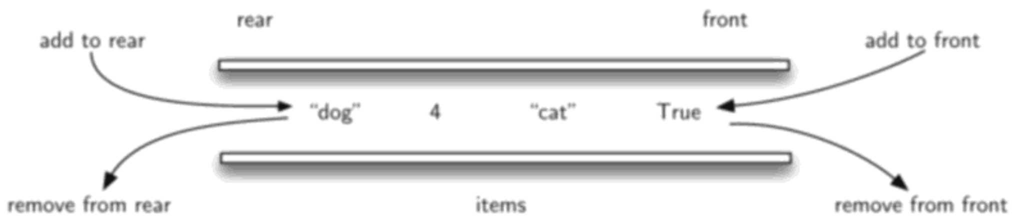

Figure 1: A Deque of Python Data Objects

Table 1: Examples of Deque Operations

Deque Operation	Deque Contents	Return Value
`d.isEmpty()`	`[]`	`True`
`d.addRear(4)`	`[4]`	
`d.addRear('dog')`	`['dog',4,]`	
`d.addFront('cat')`	`['dog',4,'cat']`	
`d.addFront(True)`	`['dog',4,'cat',True]`	
`d.size()`	`['dog',4,'cat',True]`	`4`
`d.isEmpty()`	`['dog',4,'cat',True]`	`False`
`d.addRear(8.4)`	`[8.4,'dog',4,'cat',True]`	
`d.removeRear()`	`['dog',4,'cat',True]`	`8.4`
`d.removeFront()`	`['dog',4,'cat']`	`True`

When following the implementation below, you may notice that the implementation of adding and removing items from the front is O(1) whereas adding and removing from the rear is O(n).

```
In [1]: # Implementing a Deque

        # Define Deque Class
        class Deque:
            def __init__(self):
                self.items = []

            def isEmpty(self):
                return self.items == []

            def addFront(self, item):
                self.items.append(item)

            def addRear(self, item):
                self.items.insert(0,item)

            def removeFront(self):
                return self.items.pop()

            def removeRear(self):
                return self.items.pop(0)

            def size(self):
                return len(self.items)


        d=Deque()
        print(d.isEmpty())
        d.addRear(4)
        d.addRear('dog')
        d.addFront('cat')
        d.addFront(True)
        print(d.size())
        print(d.isEmpty())
        d.addRear(8.4)
        print(d.removeRear())
        print(d.removeFront())
```

```
True
4
False
8.4
True
```

```
In [2]: # Palindrome Checker

        def palchecker(aString):
            chardeque = Deque()

        # We will process the string from left to right and we can add each character to
            for ch in aString:
                chardeque.addRear(ch)

            stillEqual = True
        # However, we can now make use of the dual functionality of the deque by comparir
            while chardeque.size() > 1 and stillEqual:
                first = chardeque.removeFront()
                last = chardeque.removeRear()
                if first != last:
                    stillEqual = False

            return stillEqual

        print(palchecker("lsdkjfskf"))
        print(palchecker("radar"))
```

```
False
True
```

4.19 Lists

A list is an abstract data type that is a collection of items where each item holds a relative position with respect to the others (lists can be unordered or ordered).

Operations include:

```
    * adding/removing values
    * searching through items/return index number of an item
    * The size of the list/if empty
    * Append and insert items
    * Pop items by position or by name
```

Note how you can inswert directly into a specific position

4.20 Linked List

You can maintain the position of items in a list using explicit links. The external reference is the start of the list (head) and each number can contain information about the location of the next item (node)

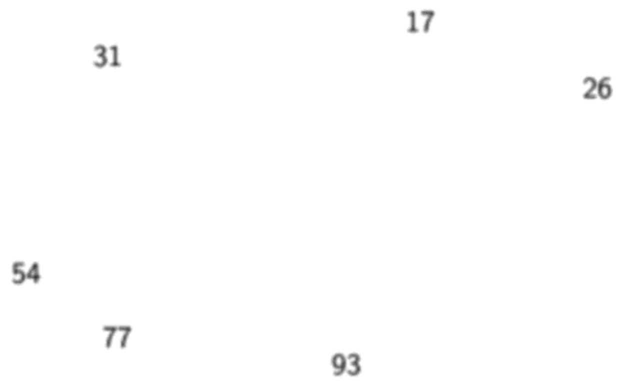

Figure 1: Items Not Constrained in Their Physical Placement

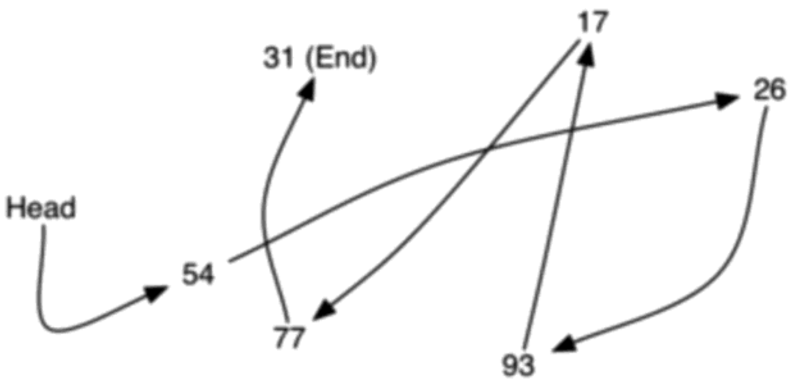

Figure 2: Relative Positions Maintained by Explicit Links.

Nodes Each linked list item is a node. And nodes have two pieces of information

```
1. The list item itself (The Data Field)
2. Reference to the next Node
```

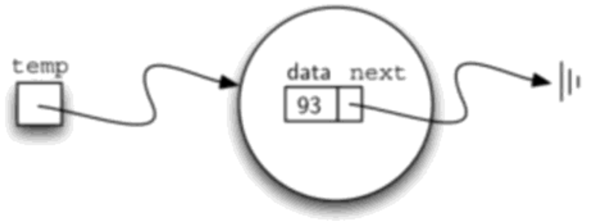

Figure 3: A Node Object Contains the Item and a Reference to the Next Node

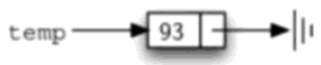

Figure 4: A Typical Representation for a Node

- None reference value indicates the end of the list, though we begin the link by grounding the first node with a None for the next reference value
- Linked list structure provides us with only one entry point, the head of the list. All the other items can be reached from the head and the next links

In [17]:
```
# Implementing a Linked List

# Define the new Node Class
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

# Creating an Unordered List

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

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

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

    def size(self):
        current = self.head
        count = 0
        while current != None:
            count = count + 1
            current = current.getNext()

        return count

    def search(self,item):
        current = self.head
        found = False
        while current != None and not found:
            if current.getData() == item:
                found = True
            else:
                current = current.getNext()

        return found

    def printList(self):
        current = self.head
        while (current):
            print(current.getData())
            current = current.getNext()

    def remove(self,item):
        current = self.head
        previous = None
        found = False
        while not found:
            if current.getData() == item:
                found = True
            else:
                previous = current
                current = current.getNext()

        if previous == None:
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())


mylist = UnorderedList()
mylist.add(31)
mylist.add(32)
mylist.add(33)

mylist.printList()
```

```
33
32
31
```

Unordered Lists

Step One: Initializes and created the node class object and sets the .setNext method of the new node to the current head of the linked list

Step Two: Resets the head of the list to refer to the newly added item

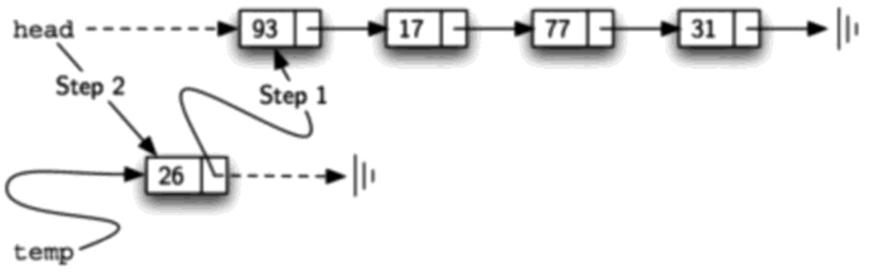

Figure 7: Adding a New Node is a Two-Step Process

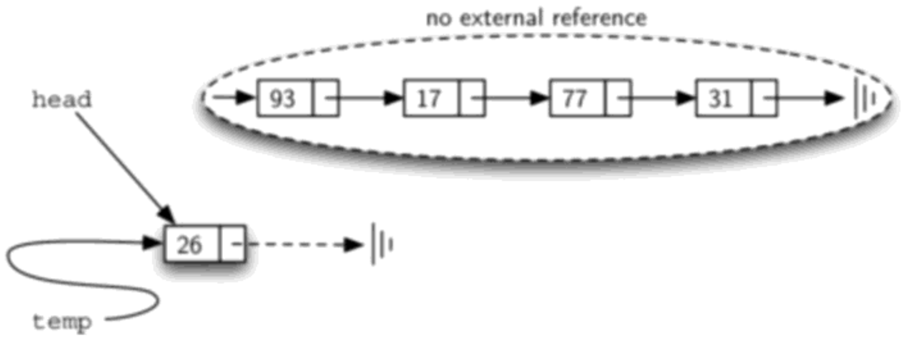

Figure 8: Result of Reversing the Order of the Two Steps

Linked List traversal (i.e. search, size, and remove) is the process of visiting every node.

Size The external reference is called current and is initialized to the head of the list in line 2. At the start of the process we have not seen any nodes so the count is set to 0. Lines 4–6 actually implement the traversal. As long as the current reference has not seen the end of the list (None), we move current along to the next node via the assignment statement in line 6. Again, the ability to compare a reference to None is very useful. Every time current moves to a new node, we add 1 to count. Finally, count gets returned after the iteration stops.

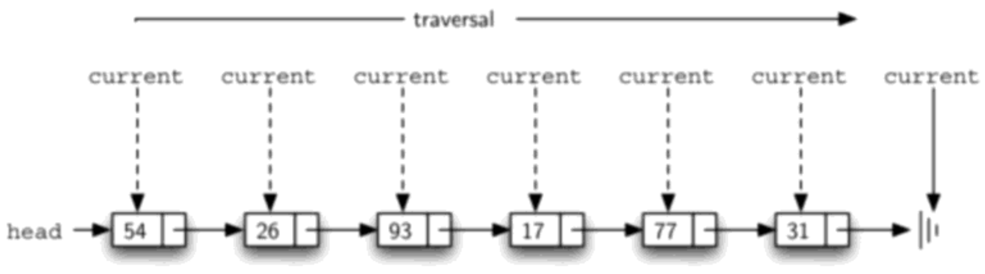

Figure 9: Traversing the Linked List from the Head to the End

Remove Since nodes cannot move backwards, the solution to this dilemma is to use two external references as we traverse down the linked list. First the pointer, current will behave just as it did before, marking the current location of the traverse. The new reference, which we will call previous, will always travel one node behind current. That way, when current stops at the node to be removed, previous will be referring to the proper place in the linked list for the modification.

Previous, however, is assumed to always travel one node behind current. For this reason, previous starts out with a value of None since there is no node before the head (see Figure 11). The boolean variable found will again be used to control the iteration. As current travels down a node and misses its search, previous will take its place before current moves to the next node. When current find the node, then previous will not take current's place and remain on the node before current. Then previous.setNext (the node's next reference) is changed to current's current next reference essentially skipping over this node, joining previous with the node after current.

The if previous == None refers to when you happen to remove the first node in a linked list.

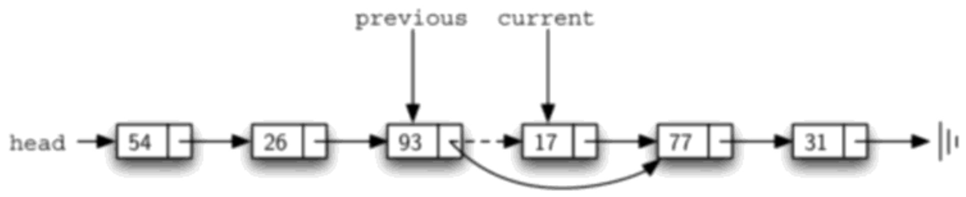

Figure 13: Removing an Item from the Middle of the List

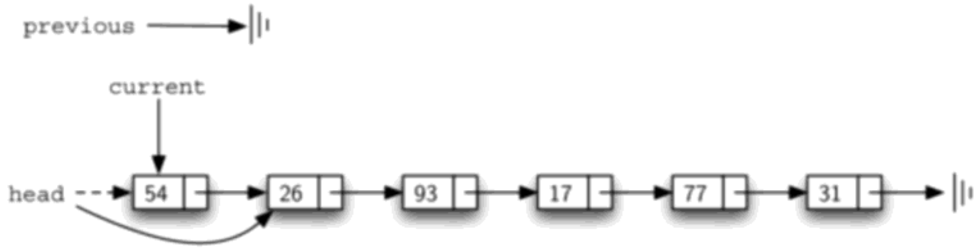

Figure 14: Removing the First Node from the List

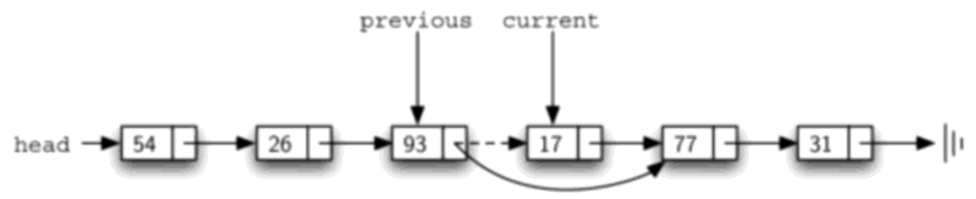

Figure 13: Removing an Item from the Middle of the List

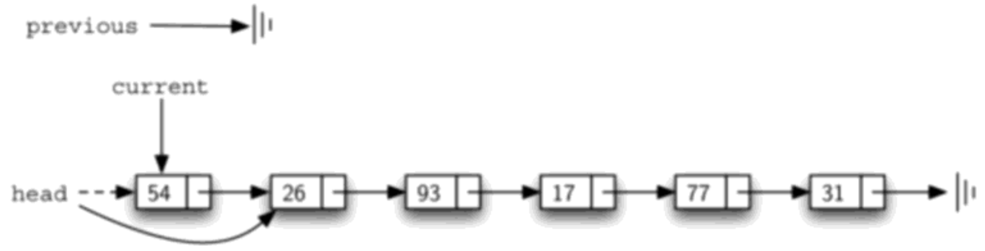

Figure 14: Removing the First Node from the List

5.2 Recursion

Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially. Usually recursion involves a function calling itself. While it may not seem like much on the surface, recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program

In [1]:
```
# Addition with recursion instead of using for/while loops

def listsum(numList):
   if len(numList) == 1:
        return numList[0]

   else: # This function strips the first item of the list until you get a large
        return numList[0] + listsum(numList[1:])

print(listsum([1,3,5,7,9]))
```

```
25
```

```
When we reach the point where the problem is as simple as it can get, we begin
to piece together the solutions of each of the small problems until the
initial problem is solved. Figure 2 shows the additions that are performed as
listsum works its way backward through the series of calls

![image.png](attachment:image.png)
```