CS 142
Stacks & Queues

Rhodes College

Announcements

Program 3 has been assigned – due 2/18 by 11:55pm

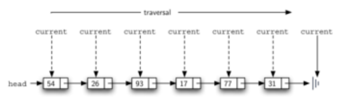
Program details on website

Creating new Data Structures

- We've used objects to create new data types such as Point, Date, Rational, BankAccount, etc.
- Now we're using objects to create new data structures
- Linked lists are a dynamic data structure, allocating the needed memory when the program is initiated.
- Insertion and deletion node operations are easily implemented in a linked list.
- Modification to the head of the linked list is a special case.
- Linked lists are a very common data structure used in computer science.

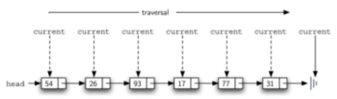
Traversing Thru a Linked List

- Traversal refers to the process of systematically visiting each node.
  - Use an external reference that starts at the first node in the list
  - As we visit each node, we move the reference to the next node by "traversing" the next reference.



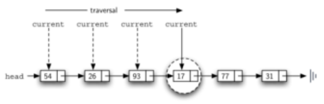
Size Method

- Returns total number of Nodes in the list
  - If list is empty, returns 0



Search Method

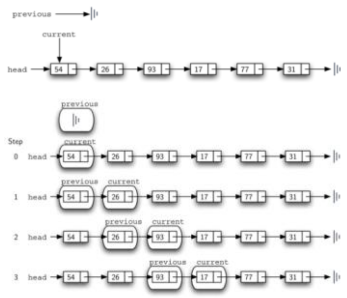
```
mylist.search(17)   #Returns True or False
```



Remove Method

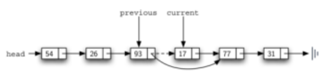
2 step process again

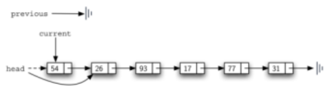
1. Find the node



Remove Method

2. Remove node from the linked list


Removing an Item from the Middle of the List


Removing the First Node from the List

Ordered Linked List

- What are the differences?
- How do we keep the list in order at all times?

Runtimes of Methods

Method	Linked List (Unordered)	Linked List (Ordered)
isEmpty	O(1)	O(1)
Add	O(1)	O(n)
Size	O(n)	O(n)
Search	O(n)	O(n)
Remove	O(n)	O(n)

Method	Python List
Append	O(1)
Insert	O(n)
Remove	O(n)
In (contains)	O(n)
Len	O(1)

Stacks

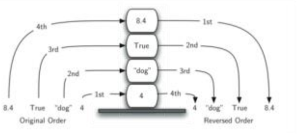
- Another new data structure!
- Commonly implemented using linked lists
- However, in Python, easier to use the built in list data structure, rather than a linked structure

Stack Definition

- Very useful for maintaining particular orders of items
- List where you only have access to the end
  - Insert or remove items from 1 end of the list
  - Can only see contents of single item at end of the list
- Think of this as a Pez dispenser
  - You load it from the top, and at any given time, you can only see the top pez and remove the top pez (you can't remove the bottom pez)
  - Pushing an item onto the stack (loading a pez into dispenser)
  - Popping: removing top item from stack
- Last item added is always first item popped back off
- LIFO = last in, first out

Applications for a Stack

- Editing commands (undo function)
- Back button on your web browser
- Useful for designing algorithms to evaluate and translate expressions.
- Can provide a reversal characteristic.



Typical Stack Methods

- `Stack()`- creates a new stack that is empty. It needs no parameters and returns an empty stack.
- `push(item)` - adds a new item to the top of the stack. It needs the item and returns nothing.
- `pop()` - removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
- `peek()` - returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
- `isEmpty()` - tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
- `size()` - returns the number of items on the stack. It needs no parameters and returns an integer.

* The "top item" refers to the last item in the list

Stack.py

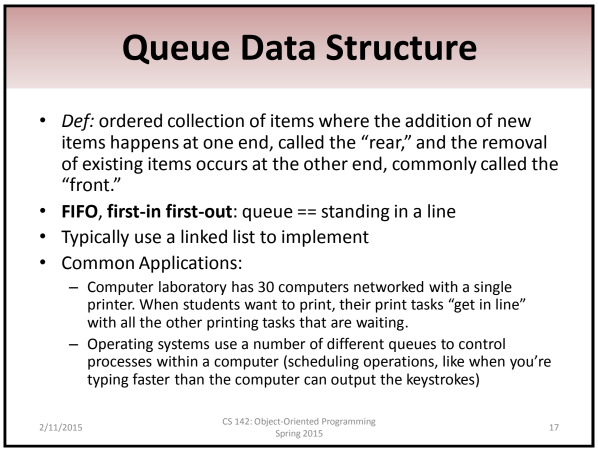
```
class Stack(object):
     def __init__(self):
         self.items = []

     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)
```

Stack.py Using Linked List

```
class Stack(object):
    def __init__(self):
        self.head = None

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

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

    def pop(self):
        if self.isEmpty():
            raise IndexError(“Empty stack.")
        else:
            temp = self.head.getData()
            self.head = self.head.getNext()
            return temp
```

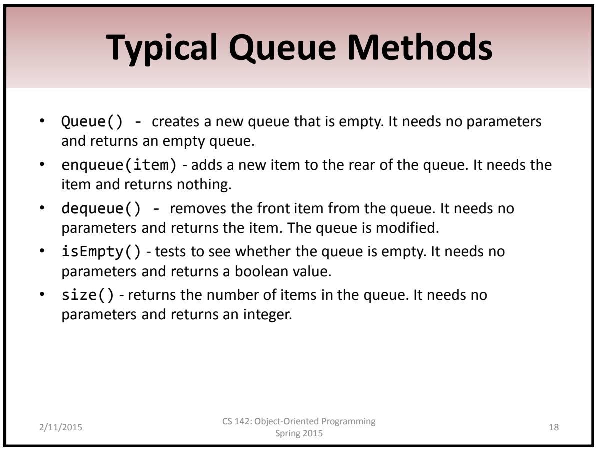
```
    def peek(self):
        return self.head.getData()

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

* Code in Public directory



Queue Data Structure

- *Def:* 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.”
- **FIFO**, **first-in first-out**: queue == standing in a line
- Typically use a linked list to implement
- Common Applications:
  - Computer laboratory has 30 computers networked with a single printer. When students want to print, their print tasks “get in line” with all the other printing tasks that are waiting.
  - Operating systems use a number of different queues to control processes within a computer (scheduling operations, like when you’re typing faster than the computer can output the keystrokes)



Typical Queue Methods

- `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.