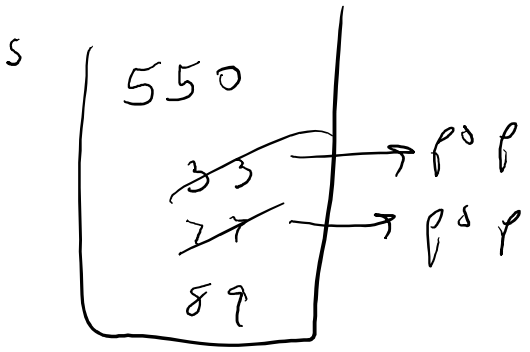
Objectives

- Be able to convert expressions between any two of the standard notations: infix, prefix, and postfix
- Be able to describe the public interface to the primary data structures: Stack, Queue, and Deque
- Be able to use Stacks and Queues in client code (Python)
- Be able to describe the postfix evaluation algorithm
- Be able to code in Python both versions of the list implementation of a Stack (“has a” and “is a”)
- Be able to describe the class SingleNode and how it would be used to create a singly linked list
- Be able to describe the class Node and the public interface of class LinkedList
- Be able to describe the abstract data type OrderedList

A warning about terminology

- The word “list” is usually not used to denote a data structure that can be subscripted (i.e. for which random access is possible).
- The Python list is closer to what C++ and Java call a “vector”

Example Application



Evaluating a postfix expression (for example 89 77 33 + 5 * -)

Algorithm valueOf(pe) /* pe is a queue of operands and operators in which is stored the expression to be evaluated */
  Create a stack s
  While pe is not empty, do
    Dequeue item from pe
    if item is an operator
      pop two operands op2 and op1 from the stack s, in that order
      apply the operator to the operands, op1 being the left-hand operand
      push the resulting value on stack s
    if item is an operand, push it on stack s
  Pop the value on top of s into variable rv
  Return rv

Translation into Python

```
from nuqueue import NuQueue
from nustack import NuStack

def getPostfixExpression():
    string = input()
    tokens = string.split()
    inputQueue = NuQueue()
    for i in range(len(tokens)):
        try:
            num = float(tokens[i])
            inputQueue.enqueue(num)
        except:
            inputQueue.enqueue(tokens[i])
    return inputQueue
```

Translation into Python (continued)

```
if (__name__=="__main__"):
    inputQ = getPostfixExpression()
    operandStack = NuStack()
    while not inputQ.isEmpty():
        op = inputQ.dequeue()
        if type(op) is str:
            op2 = operandStack.pop()
            op1 = operandStack.pop()
            if op == '-':
                operandStack.push(op1 - op2)
            elif op == '+':
                operandStack.push(op1 + op2)
            elif op == '*':
                operandStack.push(op1 * op2)
            elif op == '/':
                operandStack.push(op1 / op2)
            elif op == '//':
```

Translation into Python (concluded)

```
                operandStack.push(op1 // op2)
            elif op == '%':
                operandStack.push(op1 % op2)
            elif op == '**':
                operandStack.push(op1 ** op2)
        else:
            operandStack.push(op)
    value = operandStack.pop()
    if value == int(value):
        value = int(value)
    print(value)
```

The List (vector) Implementation of `Stack` ("Has a" version)

```
class Stack:
def __init__(self):
    self.lis = list()
def isEmpty(self):
    return len(self.lis) == 0
def push(self,item):
    self.lis.append(item)
def pop(self):
    return self.lis.pop()
def peek(self):
    return self.lis[len(self.lis)-1]
def size(self):
    return len(self.lis)
```

The List (vector) Implementation of `Stack` ("Is a" version, using inheritance)

```
class Stack(list):
    def isEmpty(self):
        return len(self) == 0
    def push(self, item):
        self.append(item)
    etc
```

The List (vector) Implementation of `Queue` ("Is a" version, using inheritance)

```
class Queue(list):
    def isEmpty(self):
        return len(self) == 0
    def enqueue(self, item):
        self.append(item)
    def dequeue(self):
        return self.items.pop(0)
    etc
```

Singly Linked lists

Class SingleNode

```
class SingleNode:
    def __init__(self, item=0):
        self.item = item
        self.link = None
```

Doubly Linked lists

Class Node

```
class Node:
    def __init__(self, item=0):
        self.item = item
        self.prev = None
        self.next = None
```

Class LinkedList public interface

LinkedList()
addFirst(item)
removeFirst(item)
addLast(item)
removeLast(item)
search(item) (Returns a Node)
isEmpty()
size()
index(item)
insert(pos,item)
insertAfter(node,item)
insertBefore(node,item)
pop()
pop(pos) (Alternately pop(node))
first() (Returns a Node)
last() (Returns a Node)

Class OrderedList public interface

OrderedList()
add(item)
remove(item)
search(item)
isEmpty()
size()
index(item)
pop()
pop(pos)