Predator, get to the choppa

Part 1: STACKS

We want a data structure that can hold a collection of objects

It is to maintain them in the order they arrive

It can only allow access to the most recently added element

Stacks allow access to the topmost element, such as recursion where we need to keep track of everything

Last-in, first-out data structure (lifo), end is referred to as the TOP

Stack operations

Top --“peaks” at the top element and returns it, doesn’t change stack, returns value

Pop - removes the top element and returns the modified stack

Push – add element to the stack

emptyStack – ensure you don’t try to access something beyond top, checks for an empty stack

size – takes stack as input, tells us the size

how to store data?

Construct our own way of doing this, *singly linked nodes*

Nodes are a structure that holds data plus a reference or points to the next node

When we hear structure, we define a class, class node()

Def class node(): - two slots

Data slot

next slot – reference to next node in the stack

singly implies there is only one way you can traverse your collection

no way to go backwards, singly linked, means you have to go only one direction

if you start in the middle, you can only go one way, not backwards

top node ----------------> repeats, or nested next fields to data nodes

data

next

data

next

none

((Image of nodes with lines connecting next to data of last node, from 30 to 10 each time))

From myStack import \*

Stack1=None #empty

Stack1=mkNode(10,stack1) #makes node, first value of 10, points to none right now

Stack1=mkNode(20,stack1) #to add more elements, second node

Stack1=mkNode(30,stack1) #third node

Print(stack1.data)

Print(stack1.next.data)

Print(stack1.next.next.data)

Prints:

30

20

10

Second way

Stack2=mkNode(60,nkNode(50,mkNode(40,None)))

Prints:

60

50

40

CODING:

#see lecture 10-28-13.py

Ingeneral you don’t use none because it can make value difficult, you define something else to replace it:

A class, class noneNode():

\_\_slots\_\_=()

Def mknoneNode():

noneNode=noneNode()

return noneNode

(the square object, separate from nodes)

Def emptyStack(myStack):

Is instance(myStack,noneNode) # allows you to check a particular variable is actually a certain type, if it is an actual node, or it is a noneNode, returns Boolean value if data is of type provided

Returns result of stack

Coding for size() in lecture 10-28-13.py

End of stack notes…..now data structure number 2, QUEUEs

Has to:

Store collection of elements

Maintain order their arrived

Allow access from one end to remove, allow access from other end to add

First in, first out, (fifo)

Fromt, back refer to the front and back of the queue

Need pointers to be able to pull from the front, and add to the back

Operations:

Enqueue – add to back end of queue

Dequeue – remove from front end

Front/back – access first or last element

Empty queue

Size

Queue has two pointers we need to maintain, one for the front, one for te back

Define a queue class

3 slots now,

Front

Back

Size – not necessary, simplifies computations, keeps track of it as we go

We can reuse our node class to represent our elements

noneNode

front

back

Size=0

Def enqueue(myQueue,el):

#since myQueue is a queue object, we can modify it and it will be returned modified

#DONT return nothing

#\* case analysis:

If queue starts empty, both front,back are updated

If queue is not empty, front stays the same, size is updated,

Back.next = newNode

Update back, back = newNode

Enqueue 20:

front

back

Size=2

Def dequeue(myQueue):

#\*case analysis:

If myQueue is empty

Raise exception

If myqueue has lots of elements:

Front=front.next

Back unchanged

Size -= 1

If myqueue has 1 element:

Front = front.next

Back needs to change too