Linked lists

A python list under the hood is implemented as an ARRAY

ARRAY – a sequential or contiguous block of memory

myLst=[4,’cat’,5.9,True] might have some unused blocks on the end of the memory block

when you have an array based representation of a memory block, you are accessing a referenced location

myLst[3] should return True

under the hood – we are getting there by using a starting point and an offset to find the location

immediate accessing o(1) because of the location+offset method

when full it will resize and reallocate memory and find more space, guessing what final size could be. Wasting space-doesn’t use memory allocated for expansion – needed to keep computational time constant

COMPLEXITY OF ARRAY BASED LIST OPERATION

Want to: access, search, insert, append, remove

Fast o(1)

Append (only problem is if we have to resize first)

Access fast (start location+offset

Slow(o(n))-

Insert(shift things to make room

Remove (same idea as insert)

Search(if sorted, get log(n))

Push something, to open end in constant time, that way you don’t shift all elements

My queue – x1,x2,x3,x4,[],[]

Front back

Want to add/remove values to static queue, don’t want to shift values, just move front,back valies

CIRCULAR LIST

myLst=(data,next) 🡪 (data,next) 🡪 [nonenode] ,,, linked list operations

in general for now, assume no back feature

slow o(n)

access – due to nature of nodes etc, slower

search – worst case is unsorted I think

insert(general)

fast o(1) – insert at front, insert at back

when to use array-based list

frequent accesss of arbitrary locations

ixed siz e data 9not doing a lot of inserts/deletes) at arbitrary locations

whe to use linked lists

dynamic data (size changes)

memory allocated only as needed

dependning on where you have your references,m insert/delete is efficient

cant go backwards without a pointer to go to previous

ENCAPSULATION:

Wrap data inside an object

Stack – did not use it

Queue – used it

Def change It(myStack) # remove first element from stack

#creates a local copy, exactly like lists in functions

#with assigning to new lists and keeping data changes after retunring,

#can’t change what myStack references and still have that change persist after function ends

#return myStack which has been modified

Encapsulated queue:

Def changeIt(myQueue):

#want to change what front/back/size points to, change contents and it will chanf

#the changes to persisting function persist

#no need to return queue, changes made in queue are permanent

Code for lecture 11-4-13