Linked Lists - Run time analysis

Announcements

- Lab 11 will be handed back Friday
- Final review sessions next week
  - Wednesday: me
  - Thursday: Prof. Newhall
  - Friday: Jeff
  - Times and locations TBD

Today's plan

- Go over quiz 6
- Review Monday
- Run-time analysis of linked lists
- Comparison with Python lists

Review

- We can remove the head of a linked list by advancing the `self.head` pointer.

- Once we can add at the head and tail and remove at the tail (all in *O(1)* time) we have a data structure that is sufficient for a number of applications.

  - Linked lists will outperform Python lists for applications that only need these operations.

- Traversing a linked list takes *O(n)* time.

Python lists vs. linked lists

- To understand difference between Python lists and linked lists we need to talk about how these data structures are laid out in memory.

- Imagine your computer's memory as a row of consecutively **numbered lockers** where each "locker" can hold a single piece of data: int, str, float, etc.

  - We refer to a locker's number as its **address**

Python list

Address	Value
8000	
8001	
8002	"A"
8003	"B"
8004	"C"
8005	
8006	
8007	
…	

“Python list of length 3 starting at 8002”

linked list

Address	Value
8000	"C"
8001	None
8002	"A"
8003	8006
8004	
8005	
8006	"B"
8007	8000
…	

“Linked list starting at 8002”

Indexing

- You can index into a Python list with a constant, *O(1)*, number of operations. To get the `i`$^{\texttt{th}}$ item, `L[i]`
    1. Calculate: start address + `i`
    2. Retrieve item at this address
- With a linked list we can't just jump to the `i`$^{\texttt{th}}$ item, we have to follow links. This is *O(n).*

Indexing

- To get item at index 3 in Python list starting at address 8002 we calculate:

  8002 + 3 = 8005

Then retrieve item at address 8005, "D"

Address	Value
8000	
8001	
8002	"A"
8003	"B"
8004	"C"
8005	"D"
8006	"E"
8007	
8008	

A Python list may not have room to grow:

Python list

Address	Value
8000	12
8001	3.4
8002	"A"
8003	"B"
8004	"C"
8005	1
8006	2
8007	3
…	

“Python list of length 3 starting at 8002”

linked list

Address	Value
8000	"C"
8001	NULL
8002	"A"
8003	8006
8004	
8005	
8006	"B"
8007	8000
…	

“Linked list starting at 8002”

	Python list	Linked list
Insert at beginning	O(n)	**O(1)**
Insert at end	O(n)*	**O(1)**
Remove from beginning	O(n)	**O(1)**
Remove from end	**O(1)**	O(n)
Get item at index	**O(1)**	O(n)

Sorting a linked list

- Because indexing in a linked list is *O(n)*, the swapping step alone is O(n) and the sorting algorithms we learned will be slower than $O(n^2)$.

- But we can use a modified version of insertion sort to achieve $O(n^2)$ for linked lists.

- Binary search will not work, again because indexing is *O(n)*.

Python lists > linked lists?

- If indexing (sometimes called **random access**) or space efficiency is important, Python lists outperform linked lists.

  - Indexing is necessary to do a swap in a sorting algorithm or to do binary search

Linked lists > Python lists?

- If we want a data structure that is **first-in, first-out** (a queue/line) or **last-in, first-out** (a stack) linked lists will outperform Python lists.

It depends…

- Choose based on the requirements of your particular application:
  - Which operations do I need?
  - Is space efficiency a concern?

More linked list methods

- `getAtIndex(index)`: return the item at a particular index
- `find(x)`: search for `x`, returning `True` if it's found, `False` otherwise
- `insertSorted(item)`: assuming the linked list is in sorted order, insert a new item while maintaining the sort.
- `removeTail()`: remove last item
- `testStructure()`: test the integrity of the linked list structure, i.e. following links gets you from `self.head` to `self.tail` with `self.size` nodes along the path.
- recursive versions of `getAtIndex(index)` and `find(x)`
- See lab 12