R2 Sequence&Set

3.22 https://github.com/GUMI-21/MIT6.006_note

Sequence Interface

Sequences maintain a collection of items in an extrinsic order, where each item stored has a rank in the sequence, including a first item and a last item.
Sequences are generalizations of stacks and queues, which support a subset of sequence operations.

Container	`build(X)`	given an iterable `X`, build sequence from items in `X`
	`len()`	return the number of stored items
Static	`iter_seq()`	return the stored items one-by-one in sequence order
	`get_at(i)`	return the i^{th} item
	`set_at(i, x)`	replace the i^{th} item with x
Dynamic	`insert_at(i, x)`	add x as the i^{th} item
	`delete_at(i)`	remove and return the i^{th} item
	`insert_first(x)`	add x as the first item
	`delete_first()`	remove and return the first item
	`insert_last(x)`	add x as the last item
	`delete_last()`	remove and return the last item

(Note that `insert_` / `delete_` operations change the rank of all items after the modified item.)

Set Interface

By contrast, Sets maintain a collection of items based on an intrinsic property involving what the items are, usually based on a unique key, x.key, associated with each item x. Sets are generalizations of dictionaries and other intrinsic query databases.

Container	`build(X)`	given an iterable `X`, build set from items in `X`
	`len()`	return the number of stored items
Static	`find(k)`	return the stored item with key `k`
Dynamic	`insert(x)`	add `x` to set (replace item with key `x.key` if one already exists)
	`delete(k)`	remove and return the stored item with key `k`
Order	`iter_ord()`	return the stored items one-by-one in key order
	`find_min()`	return the stored item with smallest key
	`find_max()`	return the stored item with largest key
	`find_next(k)`	return the stored item with smallest key larger than `k`
	`find_prev(k)`	return the stored item with largest key smaller than `k`

(Note that `find` operations return `None` if no qualifying item exists.)

Implementations

Data Structure	Operation, Worst Case $O(\cdot)$				
	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first()`	`insert_last(x)` `delete_last()`	`insert_at(i, x)` `delete_at(i)`
Array	n	1	n	n	n
Linked List	n	n	1	n	n
Dynamic Array	n	1	n	$1_{(a)}$	n

Array Sequence

place fixed

Linked List Sequence

`code see r2_linked_list.py`
also called pointer-based or linked.
their constituent items can be stored anywhere in memory.

Dynamic Array Sequnce

One straight-forward way to support faster insertion would be to over-allocate additional space when you request space for the array.

- How python does : it doesn't
  amortized constant time
  A typical implementation of a dynamic array will allocate double the amount of space needed to store the current array, sometimes referred to as table doubling. H
  Python Lists allocate additional space according to the following formula (from the Python source code written in C): 1 new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);

When attempting to append past the end of the allocation, the contents of the array are transferred to an allocation that is twice as large. When removing down to one fourth of the allocation, the contents of the array are transferred to an allocation that is half as large.

how amortized constant append and pop could be implemented.

Exercise

1.Suppose the next pointer of the last node of a linked list points to an earlier node in the list, creating a cycle. Given a pointer to the head of the list (without knowing its size), describe a linear-time algorithm to find the number of nodes in the cycle. Can you do this while using only constant additional space outside of the original linked list?

answer: use tow pointers pointing at the head of the linked list.Use Fast-slow pointers.
slow pointer: move to next
fast pointer: inital at head.next and move to next.next
when slow equal fast means fast point has made a full loop around cycle, then fixed slow pointer, make fast pointer take one step until slow pointer again to count nodes in cycle.

2.Given a data structure implementing the Sequence interface, show how to use it to implement the Set interface. (Your implementation does not need to be efficient.)