Recitation 2

Sequence Interface (L02, L07)

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. By extrinsic, we mean that the first item is 'first', not because of what the item is, but because some external party put it there. 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 (L03-L08)

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.)

Sequence Implementations

Here, we will discuss three data structures to implement the sequence interface. In Problem Set 1, you will extend both Linked Lists and Dynamic arrays to make both first and last dynamic operations $O(1)$ time for each. Notice that none of these data structures support dynamic operations at arbitrary index in sub-linear time. We will learn how to improve this operation in Lecture 7.

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

Computer memory is a finite resource. On modern computers many processes may share the same main memory store, so an operating system will assign a fixed chunk of memory addresses to each active process. The amount of memory assigned depends on the needs of the process and the availability of free memory. For example, when a computer program makes a request to store a variable, the program must tell the operating system how much memory (i.e. how many bits) will be required to store it. To fulfill the request, the operating system will find the available memory in the process's assigned memory address space and reserve it (i.e. allocate it) for that purpose until it is no longer needed. Memory management and allocation is a detail that is abstracted away by many high level languages including Python, but know that whenever you ask Python to store something, Python makes a request to the operating system behind-the-scenes, for a fixed amount of memory in which to store it.

Now suppose a computer program wants to store two arrays, each storing ten 64-bit words. The program makes separate requests for two chunks of memory (640 bits each), and the operating system fulfills the request by, for example, reserving the first ten words of the process's assigned address space to the first array A, and the second ten words of the address space to the second array B. Now suppose that as the computer program progresses, an eleventh word w needs to be added to array A. It would seem that there is no space near A to store the new word: the beginning of the process's assigned address space is to the left of A and array B is stored on the right. Then how can we add w to A? One solution could be to shift B right to make room for w, but tons of data may already be reserved next to B, which you would also have to move. Better would be to simply request eleven new words of memory, copy A to the beginning of the new memory allocation, store w at the end, and free the first ten words of the process's address space for future memory requests.

A fixed-length array is the data structure that is the underlying foundation of our model of computation (you can think of your computer's memory as a big fixed-length array that your operating

system allocates from). Implementing a sequence using an array, where index i in the array corresponds to item i in the sequence allows `get_at` and `set_at` to be $O(1)$ time because of our random access machine. However, when deleting or inserting into the sequence, we need to move items and resize the array, meaning these operations could take linear-time in the worst case. Below is a full Python implementation of an array sequence.

```
class Array_Seq:
    def __init__(self):                                 # O(1)
        self.A = []
        self.size = 0

    def __len__(self):      return self.size            # O(1)
    def __iter__(self):     yield from self.A           # O(n) iter_seq

    def build(self, X):                                 # O(n)
        self.A = [a for a in X]  # pretend this builds a static array
        self.size = len(self.A)

    def get_at(self, i):    return self.A[i]            # O(1)
    def set_at(self, i, x): self.A[i] = x               # O(1)

    def _copy_forward(self, i, n, A, j):                # O(n)
        for k in range(n):
            A[j + k] = self.A[i + k]

    def _copy_backward(self, i, n, A, j):               # O(n)
        for k in range(n - 1, -1, -1):
            A[j + k] = self.A[i + k]

    def insert_at(self, i, x):                          # O(n)
        n = len(self)
        A = [None] * (n + 1)
        self._copy_forward(0, i, A, 0)
        A[i] = x
        self._copy_forward(i, n - i, A, i + 1)
        self.build(A)

    def delete_at(self, i):                             # O(n)
        n = len(self)
        A = [None] * (n - 1)
        self._copy_forward(0, i, A, 0)
        x = self.A[i]
        self._copy_forward(i + 1, n - i - 1, A, i)
        self.build(A)
        return x
                                                        # O(n)
    def insert_first(self, x):  self.insert_at(0, x)
    def delete_first(self):     return self.delete_at(0)
    def insert_last(self, x):   self.insert_at(len(self), x)
    def delete_last(self):      return self.delete_at(len(self) - 1)
```

Linked List Sequence

A **linked list** is a different type of data structure entirely. Instead of allocating a contiguous chunk of memory in which to store items, a linked list stores each item in a node, `node`, a constant-sized container with two properties: `node.item` storing the item, and `node.next` storing the memory address of the node containing the next item in the sequence.

```
class Linked_List_Node:
    def __init__(self, x):                          # O(1)
        self.item = x
        self.next = None

    def later_node(self, i):                        # O(i)
        if i == 0:  return self
        assert self.next
        return self.next.later_node(i - 1)
```

Such data structures are sometimes called **pointer-based** or **linked** and are much more flexible than array-based data structures because their constituent items can be stored anywhere in memory. A linked list stores the address of the node storing the first element of the list called the **head** of the list, along with the linked list's size, the number of items stored in the linked list. It is easy to add an item after another item in the list, simply by changing some addresses (i.e. relinking pointers). In particular, adding a new item at the front (head) of the list takes $O(1)$ time. However, the only way to find the i^{th} item in the sequence is to step through the items one-by-one, leading to worst-case linear time for `get_at` and `set_at` operations. Below is a Python implementation of a full linked list sequence.

```
class Linked_List_Seq:
    def __init__(self):                             # O(1)
        self.head = None
        self.size = 0

    def __len__(self):  return self.size            # O(1)

    def __iter__(self):                             # O(n) iter_seq
        node = self.head
        while node:
            yield node.item
            node = node.next

    def build(self, X):                             # O(n)
        for a in reversed(X):
            self.insert_first(a)

    def get_at(self, i):                            # O(i)
        node = self.head.later_node(i)
        return node.item

```

```
22      def set_at(self, i, x):                             # O(i)
23          node = self.head.later_node(i)
24          node.item = x
25
26      def insert_first(self, x):                          # O(1)
27          new_node = Linked_List_Node(x)
28          new_node.next = self.head
29          self.head = new_node
30          self.size += 1
31
32      def delete_first(self):                             # O(1)
33          x = self.head.item
34          self.head = self.head.next
35          self.size -= 1
36          return x
37
38      def insert_at(self, i, x):                          # O(i)
39          if i == 0:
40              self.insert_first(x)
41              return
42          new_node = Linked_List_Node(x)
43          node = self.head.later_node(i - 1)
44          new_node.next = node.next
45          node.next = new_node
46          self.size += 1
47
48      def delete_at(self, i):                             # O(i)
49          if i == 0:
50              return self.delete_first()
51          node = self.head.later_node(i - 1)
52          x = node.next.item
53          node.next = node.next.next
54          self.size -= 1
55          return x
56                                                          # O(n)
57      def insert_last(self, x):   self.insert_at(len(self), x)
58      def delete_last(self):      return self.delete_at(len(self) - 1)
```

Dynamic Array Sequence

The array's dynamic sequence operations require linear time with respect to the length of array A. Is there another way to add elements to an array without paying a linear overhead transfer cost each time you add an element? One straight-forward way to support faster insertion would be to over-allocate additional space when you request space for the array. Then, inserting an item would be as simple as copying over the new value into the next empty slot. This compromise trades a little extra space in exchange for constant time insertion. Sounds like a good deal, but any additional allocation will be bounded; eventually repeated insertions will fill the additional space, and the array will again need to be reallocated and copied over. Further, any additional space you reserve will mean less space is available for other parts of your program.

Then how does Python support appending to the end of a length n Python List in worst-case $O(1)$ time? The answer is simple: **it doesn't**. Sometimes appending to the end of a Python List requires $O(n)$ time to transfer the array to a larger allocation in memory, so **sometimes** appending to a Python List takes linear time. However, allocating additional space in the right way can guarantee that any sequence of n insertions only takes at most $O(n)$ time (i.e. such linear time transfer operations do not occur often), so insertion will take $O(1)$ time per insertion **on average**. We call this asymptotic running time **amortized constant time**, because the cost of the operation is amortized (distributed) across many applications of the operation.

To achieve an amortized constant running time for insertion into an array, our strategy will be to allocate extra space in proportion to the size of the array being stored. Allocating $O(n)$ additional space ensures that a linear number of insertions must occur before an insertion will overflow the allocation. 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**. However, allocating any constant fraction of additional space will achieve the amortized bound. Python Lists allocate additional space according to the following formula (from the Python source code written in C):

```
new_allocated = (newsize >> 3) + (newsize < 9 ? 3 : 6);
```

Here, the additional allocation is modest, roughly one eighth of the size of the array being appended (bit shifting the size to the right by 3 is equivalent to floored division by 8). But the additional allocation is still linear in the size of the array, so on average, $n/8$ insertions will be performed for every linear time allocation of the array, i.e. amortized constant time.

What if we also want to remove items from the end of the array? Popping the last item can occur in constant time, simply by decrementing a stored length of the array (which Python does). However, if a large number of items are removed from a large list, the unused additional allocation could occupy a significant amount of wasted memory that will not available for other purposes. When the length of the array becomes sufficiently small, we can transfer the contents of the array to a new, smaller memory allocation so that the larger memory allocation can be freed. How big should this new allocation be? If we allocate the size of the array without any additional allocation, an immediate insertion could trigger another allocation. To achieve constant amortized running time for any sequence of n appends or pops, we need to make sure there remains a linear fraction of unused allocated space when we rebuild to a smaller array, which guarantees that at least $\Omega(n)$ sequential dynamic operations must occur before the next time we need to reallocate memory.

Below is a Python implementation of a dynamic array sequence, including operations `insert_last` (i.e., Python list `append`) and `delete_last` (i.e., Python list `pop`), using table doubling proportions. 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. Of course Python Lists already support dynamic operations using these techniques; this code is provided to help you understand how **amortized constant** `append` and `pop` could be implemented.

```
class Dynamic_Array_Seq(Array_Seq):
    def __init__(self, r = 2):                          # O(1)
        super().__init__()
        self.size = 0
        self.r = r
        self._compute_bounds()
        self._resize(0)

    def __len__(self):  return self.size                # O(1)

    def __iter__(self):                                 # O(n)
        for i in range(len(self)): yield self.A[i]

    def build(self, X):                                 # O(n)
        for a in X: self.insert_last(a)

    def _compute_bounds(self):                          # O(1)
        self.upper = len(self.A)
        self.lower = len(self.A) // (self.r * self.r)

    def _resize(self, n):                               # O(1) or O(n)
        if (self.lower < n < self.upper): return
        m = max(n, 1) * self.r
        A = [None] * m
        self._copy_forward(0, self.size, A, 0)
        self.A = A
        self._compute_bounds()

    def insert_last(self, x):                           # O(1)a
        self._resize(self.size + 1)
        self.A[self.size] = x
        self.size += 1

    def delete_last(self):                              # O(1)a
        self.A[self.size - 1] = None
        self.size -= 1
        self._resize(self.size)

    def insert_at(self, i, x):                          # O(n)
        self.insert_last(None)
        self._copy_backward(i, self.size - (i + 1), self.A, i + 1)
        self.A[i] = x

    def delete_at(self, i):                             # O(n)
        x = self.A[i]
        self._copy_forward(i + 1, self.size - (i + 1), self.A, i)
        self.delete_last()
        return x
                                                        # O(n)
    def insert_first(self, x):  self.insert_at(0, x)
    def delete_first(self):     return self.delete_at(0)
```

Exercises:

- 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?

  Solution: Begin with two pointers pointing at the head of the linked list: one slow pointer and one fast pointer. The pointers take turns traversing the nodes of the linked list, starting with the fast pointer. On the slow pointer's turn, the slow pointer simply moves to the next node in the list; while on the fast pointer's turn, the fast pointer initially moves to the next node, but then moves on to the next node's next node before ending its turn. Every time the fast pointer visits a node, it checks to see whether it's the same node that the slow pointer is pointing to. If they are the same, then the fast pointer must have made a full loop around the cycle, to meet the slow pointer at some node v on the cycle. Now to find the length of the cycle, simply have the fast pointer continue traversing the list until returning back to v, counting the number of nodes visited along the way.

  To see that this algorithm runs in linear time, clearly the last step of traversing the cycle takes at most linear time, as v is the only node visited twice while traversing the cycle. Further, we claim the slow pointer makes at most one move per node. Suppose for contradiction the slow pointer moves twice away from some node u before being at the same node as the fast pointer, meaning that u is on the cycle. In the same time the slow pointer takes to traverse the cycle from u back to u, the fast pointer will have traveled around the cycle twice, meaning that both pointers must have existed at the same node prior to the slow pointer leaving u, a contradiction.

- 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.)

  Solution:

```
def Set_from_Seq(seq):
    class set_from_seq:
        def __init__(self):   self.S = seq()
        def __len__(self):    return len(self.S)
        def __iter__(self):   yield from self.S

        def build(self, A):
            self.S.build(A)

        def insert(self, x):
            for i in range(len(self.S)):
                if self.S.get_at(i).key == x.key:
                    self.S.set_at(i, x)
                    return
            self.S.insert_last(x)

```

```
17          def delete(self, k):
18              for i in range(len(self.S)):
19                  if self.S.get_at(i).key == k:
20                      return self.S.delete_at(i)
21
22          def find(self, k):
23              for x in self:
24                  if x.key == k:  return x
25              return None
26
27          def find_min(self):
28              out = None
29              for x in self:
30                  if (out is None) or (x.key < out.key):
31                      out = x
32              return out
33
34          def find_max(self):
35              out = None
36              for x in self:
37                  if (out is None) or (x.key > out.key):
38                      out = x
39              return out
40
41          def find_next(self, k):
42              out = None
43              for x in self:
44                  if x.key > k:
45                      if (out is None) or (x.key < out.key):
46                          out = x
47              return out
48
49          def find_prev(self, k):
50              out = None
51              for x in self:
52                  if x.key < k:
53                      if (out is None) or (x.key > out.key):
54                          out = x
55              return out
56
57          def iter_ord(self):
58              x = self.find_min()
59              while x:
60                  yield x
61                  x = self.find_next(x.key)
62
63      return set_from_seq
```

6.006 Introduction to Algorithms

Spring 2020