6.006	Lecture 4	Sept. 18, 2018

TODAY: Data Structures (intro.)
- sequence interface & data structures:
  - linked lists
  - dynamic arrays
    → amortization
- set interface

Interface (API/ADT) vs. Data Structure

- specification
- what data can be maintained
- supported operations & their meaning
- problem

- representation
- how the data gets stored (internal vars.)
- algorithms implementing the operations
- solution

2 main interfaces: (in this class)
- sequence [TODAY]
- set [L5-L9]

2 main data structure paradigms:
- pointer based
- array based

Static sequence interface:

maintain a sequence of items/objects

$x_0, x_1, x_2, \ldots, x_{n-1}$

subject to:

- len(): return n
- seq-iter(): output $x_0, x_1, \ldots, x_{n-1}$
  $\equiv$ at(0), at(1), ..., at(n-1)
- at(i): return object x_i of index i
  - special cases: left() $\equiv$ at(0)
    right() $\equiv$ at($n-1$)
    logically equivalent, but implementations may differ
- set-at(i, x): change x_i to x

What data structure solves this?

static array $A[0..n-1]$

- $A[i]$ stores x_i
- $O(1)$ time per operation except $O(n)$ time per iter

Dynamic sequence interface: static sequence plus:

- insert-at(i, x): make x the new x_i, shifting $x_i \to x_{i+1} \to x_{i+2} \to \cdots \to x_{n-1} \to x_{n'-1}$
  - special cases: insert-left & insert-right
- delete-at(i): shift $x_i \leftarrow x_{i+1} \leftarrow \cdots \leftarrow x_{n'-1} \leftarrow x_{n-1}$
  - special cases: delete-left & delete-right

$\Rightarrow$ set-at(i, x) $\equiv$ delete-at(i); insert-at(i, x)
(but set-at may be more efficient)

Special cases of interest:

- stack = right(), insert/delete-right()
  top push pop
- queue = insert-right(), delete-left()
  enqueue dequeue
- deque = insert/delete - left/right

How well do static arrays do?

- $\Theta(n)$ for insert-at & delete-at ☹
- 2 linear costs:
  - shifting indices
  - resizing the array

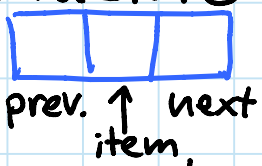
  $\Rightarrow$ even insert/delete-left/right slow!

Model for memory allocation:
can allocate an array of n words in $\Theta(n)$ time $\Rightarrow$ space = O(time)

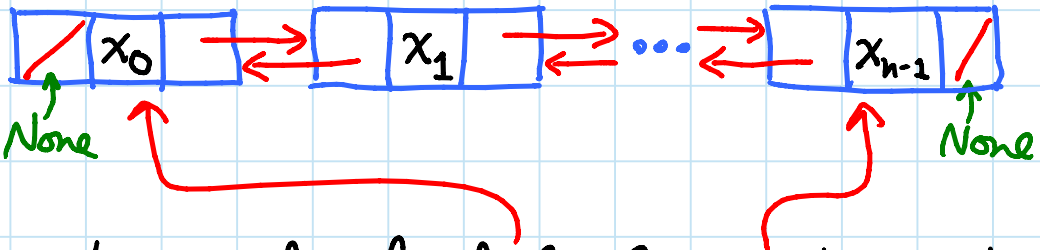
↗ unrelated to Python's "list"

Linked lists: pointer-based data structure

- allocate 1 item at a time:



- connect items together with pointers:



- maintain global left & right pointers
  ⇒ fast left() & right()
- insert/delete-left/right(x) also O(1) time:

⇒ solve stack, queue, and deque!

- Singly linked lists save space by just storing next pointers, not prev
  ⇒ delete-right() becomes hard

- but at(i) is slow now ☹

Can we get the best of both worlds?

- at() performance of arrays
- insert/delete-left/right() of linked lists

Dynamic arrays: ≈ Python's list

- instead of resizing array to exactly n, allow array to have size $= \Theta(n)$
- when inserting & size $= n$:
  double size! size *= 2

- worst-case insert-right() still $\Theta(n)$
- but resize only when $n = 2^i$
- ⇒ n insert-right()s cost $\Theta(1+2+4+8+\cdots+n)$
  (really the next power of 2 ↑)
  $= \Theta(n)$
  - a few inserts cost linear time, but $\Theta(1)$ "on average"

Amortized analysis — common technique in DSs

- like paying rent: \$1500/month ≈ \$50/day
- operation has amortized cost $T(n)$ if k operations cost $\leq k \cdot T(n)$
- "$T(n)$ amortized" roughly means $T(n)$ "on average", but averaged over all ops.
- e.g. inserting into a hash table takes $O(1)$ amortized time

Dynamic array delete-right:

- $O(1)$ time without effort
- but then may reach state with $n \ll$ size
  - e.g. $n \times$ insert-right, $n \times$ delete-right
- solution: when $n <$ size$/4$:
  resize to size$/2$ size //= 2

$\Rightarrow O(1)$ amortized for both insert/delete-right
- analysis harder: see 6.046/CLRS 17.4

Exercise: why not shrink when $n <$ size$/2$?

In fact, any constants $1 < 2 < 4$ work too.

In PS2, you'll see how to make insert/delete-left fast too!

Sequence all about index in given ordering
Set is all about key in each item/object

Set interface: ≈ frozenset/dict

maintain a set S of items with keys s.t.

- find-key(k): find item with key k (if exists)
- iter(): output items in arbitrary order

Dynamic set interface: ≈ set/dict

- insert(x): add x to S (overwriting colliding key)
- delete-key(k): remove $x \in S$ with $x.key = k$

Ordered set interface: keys must be ordered

- find-next(k): find $x \in S$ with smallest key $> k$
- find-prev(k): find $x \in S$ with largest key $< k$
- find-min(): find $x \in S$ with smallest key
  $\equiv$ find-next($-\infty$)
- find-max(): find $x \in S$ with largest key
  $\equiv$ find-prev(∞)
- order-iter(): output items in sorted order by key

Dynamic ordered set interface:

- delete-min() $\equiv$ delete-key(find-min().key)
- delete-max() $\equiv$ delete-key(find-max().key)

Priority queue interface: special D.O.S. studied in L5

- insert(x)
- [find/delete-min() OR find/delete-max()]

```
def allocate(n): # pretend static array
    return [None] * n

class StaticArray (ArraySequence):
    def __init__(self, n):
        self.n = n
        self.array = allocate(n)
    def len(self): return self.n
    def at(self, i): return self.array[i]
    def set_at(self, i, x):
        self.array[i] = x
```

```
class ArraySequence:
    def seq_iter(self):
        for i in range(self.len()): yield self.at(i)
    def left(self): return self.at(0)
    def right(self): return self.at(self.len()-1)
    def set_left(self, x): self.set_at(0, x)
    def set_right(self, x): self.set_at(self.len()-1, x)

class StaticArray (ArraySequence):
    def __init__(self, n):
        self.n = n
        self.array = allocate(n)
    def len(self): return self.n
    def at(self, i): return self.array[i]
    def set_at(self, i, x): self.array[i] = x
```

```
def allocate(n): # pretend static array
    return [None] * n

class StaticArray (ArraySequence):
    def delete_at(self, i):
        old_array = self.array
        self.n -= 1
        self.array = allocate(self.n)
        for k in range(i):
            self.array[k] = old_array[k]
        # skip old_array[i]
        for k in range(i, self.n):
            self.array[k] = old_array[k+1]
```

```
def allocate(n): # pretend static array
    return [None] * n

class StaticArray (ArraySequence):
    def insert_at(self, i, x):
        old_array = self.array
        self.n += 1
        self.array = allocate(self.n)
        for k in range(i):
            self.array[k] = old_array[k]
        self.array[i] = x
        for k in range(i+1, self.n):
            self.array[k] = old_array[k-1]
```

Data Structure	Static Sequence			Dynamic Sequence			Space ~
	at(i)	left(), set-left(x)		insert-at(i, x)	insert-left(x)	insert-right(x)	
	set-at (i, x)	right(), set-right(x)	seq-iter()	delete-at(i)	delete-left(x)	delete-right(x)	
static array	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$1 \cdot n$
linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$3 \cdot n$
dynamic array	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$ amortized	$[n, 4n]$

```
class Node:
    def __init__(self, item):
        self.item = item
        self.prev = self.next = None

def link(a, b): # a <--> b
    a.next = b
    b.prev = a

def unlink(a, b): # a <-/-> b
    assert a.next is b and b.prev is a
    a.next = b.prev = None
```

```
def link(a, b): # a <--> b
    a.next = b
    b.prev = a

class LinkedList:
    def __init__(self):
        self.left = self.right = None
    def insert_right(self, x):
        new_right = Node(x)
        if self.right is None: # was empty
            self.left = self.right = new_right
        else:
            link(self.right, new_right)
            self.right = new_right
```

```
def link(a, b): # a <--> b
    a.next = b
    b.prev = a

class LinkedList:
    def __init__(self):
        self.left = self.right = None
    def insert_left(self, x):
        new_left = Node(x)
        if self.left is None: # was empty
            self.left = self.right = new_right
        else:
            link(new_left, self.left)
            self.left = new_left
```

```
def unlink(a, b): # a <-/-> b
    assert a.next is b and b.prev is a
    a.next = b.prev = None

class LinkedList:
    def __init__(self):
        self.left = self.right = None
    def delete_left(self):
        new_left = self.left.next
        if new_left is None: # now empty
            self.left = self.right = None
        else:
            unlink(self.left, new_left)
            self.left = new_left
```

```
def unlink(a, b): # a <-/-> b
    assert a.next is b and b.prev is a
    a.next = b.prev = None

class LinkedList:
    def __init__(self):
        self.left = self.right = None
    def delete_right(self):
        new_right = self.right.prev
        if new_right is None: # now empty
            self.left = self.right = None
        else:
            unlink(new_right, self.right)
            self.right = new_right
```

```
class LinkedList:
    def __init__(self):
        self.left = self.right = None

    def at(self, i):
        node = self.left
        for k in range(i):
            node = node.next
        return node.item

    def left(self):
        return self.left.item
    def right(self):
        return self.right.item
```

Data Structure	Static Sequence			Dynamic Sequence			Space ~
	at(i)	left(), set-left(x)		insert-at(i, x)	insert-left(x)	insert-right(x)	
	set-at (i, x)	right(), set-right(x)	seq-iter()	delete-at(i)	delete-left(x)	delete-right(x)	
static array	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$1 \cdot n$
linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$3 \cdot n$
dynamic array	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$ amortized	$[n, 4n]$

Data Structure	Static Sequence			Dynamic Sequence			Space ~
	at(i)	left(), set-left(x)		insert-at(i, x)	insert-left(x)	insert-right(x)	
	set-at (i, x)	right(), set-right(x)	seq-iter()	delete-at(i)	delete-left(x)	delete-right(x)	
static array	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$1 \cdot n$
singly linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$ ins. $\Theta(n)$ del.	$2 \cdot n$
doubly linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$3 \cdot n$
dynamic array	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$ amortized	$[n, 4n]$

```
def allocate(n): # pretend static array
    return [None] * n

class DynamicArray (ArraySequence):
    def __init__(self):
        self.n = 0
        self.size = 1
        self.array = allocate(self.size)
    def len(self): return self.n
    def at(self, i): return self.array[i]
    def set_at(self, i, x): self.array[i] = x
```

```
class DynamicArray (ArraySequence):
    def resize(self, new_size):
        old_array = self.array
        self.size = new_size
        self.array = allocate(self.size)
        for i in range(self.n):
            self.array[i] = old_array[i]
```

```
class DynamicArray (ArraySequence):
    def resize(self, new_size):
        old_array = self.array
        self.size = new_size
        self.array = allocate(self.size)
        for i in range(self.n):
            self.array[i] = old_array[i]

    def insert_right(self, x):
        if self.n == self.size:
            self.resize(self.size * 2)
        self.array[self.n] = x
        self.n += 1
```

```
class DynamicArray (ArraySequence):
    def resize(self, new_size):
        old_array = self.array
        self.size = new_size
        self.array = allocate(self.size)
        for i in range(self.n):
            self.array[i] = old_array[i]

    def delete_right(self, x): # bad space-wise
        self.n -= 1
```

```
class DynamicArray (ArraySequence):
    def resize(self, new_size):
        old_array = self.array
        self.size = new_size
        self.array = allocate(self.size)
        for i in range(self.n):
            self.array[i] = old_array[i]

    def delete_right(self):
        if self.n <= self.size // 4:
            self.resize(self.size // 2)
        self.n -= 1
```

```
class DynamicArray (ArraySequence):
    def resize(self, new_size):
        old_array = self.array
        self.size = new_size
        self.array = allocate(self.size)
        for i in range(self.n):
            self.array[i] = old_array[i]
    def delete_at(self, i):
        if self.n <= self.size // 4:
            self.resize(self.size // 2)
        self.n -= 1
        for k in range(i, self.n):
            self.array[k] = self.array[k+1]
```

```
class DynamicArray (ArraySequence):
    def resize(self, new_size):
        old_array = self.array
        self.size = new_size
        self.array = allocate(self.size)
        for i in range(self.n):
            self.array[i] = old_array[i]
    def insert_at(self, i, x):
        if self.n == self.size:
            self.resize(self.size * 2)
        self.n += 1
        for k in reversed(range(i+1, self.n)):
            self.array[k] = self.array[k-1]
        self.array[i] = x
```

Data Structure	Static Sequence			Dynamic Sequence			Space ~
	at(i)	left(), set-left(x)		insert-at(i, x)	insert-left(x)	insert-right(x)	
	set-at (i, x)	right(), set-right(x)	seq-iter()	delete-at(i)	delete-left(x)	delete-right(x)	
static array	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$1 \cdot n$
linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$3 \cdot n$
dynamic array	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$ amortized	$[n, 4n]$

Data Structure	Static Set		Dynamic Set		D.O.S.	Ordered Set			Space ~
	find-key(k)	iter()	insert(x)	delete-key(k)	delete-min/ max()	find-next/ prev(k)	find-min/ max()	order-iter()	
static array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \lg n)$	$1 \cdot n$
linked list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \lg n)$	$3 \cdot n$
dynam. array	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$ a.	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \lg n)$	$[n, 4n]$
sorted array	$\Theta(\lg n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(1)$	$\Theta(n)$	$1 \cdot n$
direct-access array	$\Theta(1)$	$\Theta(u)$	$\Theta(1)$	$\Theta(1)$	$\Theta(u)$	$\Theta(u)$	$\Theta(u)$	$\Theta(u)$	u
binary heap	$\Theta(n)$	$\Theta(n)$	$\Theta(\lg n)$ a.	$\Theta(n)$	one in $\Theta(\lg n)$	$\Theta(n)$	one in $\Theta(1)$	$\Theta(n \lg n)$	$1 \cdot n$
AVL tree	$\Theta(\lg n)$	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(n)$	$5 \cdot n$
hash table	$\Theta(1)$ e.	$\Theta(n)$	$\Theta(1)$ e. a.	$\Theta(1)$ e. a.	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \lg n)$	$4 \cdot n$