Lecture 2: Data Structures

Data Structure Interfaces

- A **data structure** is a way to store data, with algorithms that support **operations** on the data
- Collection of supported operations is called an **interface** (also API or ADT)
- Interface is a **specification**: what operations are supported (the problem!)
- Data structure is a **representation**: how operations are supported (the solution!)
- In this class, two main interfaces: **Sequence** and **Set**

Sequence Interface (L02, L07)

- Maintain a sequence of items (order is **extrinsic**)
- Ex: $(x_0, x_1, x_2, \ldots, x_{n-1})$ (zero indexing)
- (use n to denote the number of items stored in the data structure)
- Supports 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

- Special case interfaces:

stack	`insert_last(x)` and `delete_last()`
queue	`insert_last(x)` and `delete_first()`

Set Interface (L03-L08)

- Sequence about **extrinsic** order, set is about **intrinsic** order
- Maintain a set of items having **unique keys** (e.g., item `x` has key `x.key`)
- (Set or multi-set? We restrict to unique keys for now.)
- Often we let key of an item be the item itself, but may want to store more info than just key
- Supports set operations:

Container	`build(X)`	given an iterable `X`, build sequence 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`

- Special case interfaces:

  dictionary | set without the Order operations

- In recitation, you will be asked to implement a Set, given a Sequence data structure.

Array Sequence

- Array is great for static operations! `get_at(i)` and `set_at(i, x)` in $\Theta(1)$ time!
- But not so great at dynamic operations...
- (For consistency, we maintain the invariant that array is full)
- Then inserting and removing items requires:
  - reallocating the array
  - shifting all items after the modified item

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 Sequence

- Pointer data structure (this is **not** related to a Python "list")
- Each item stored in a **node** which contains a pointer to the next node in sequence
- Each `node` has two fields: `node.item` and `node.next`
- Can manipulate nodes simply by relinking pointers!
- Maintain pointers to the first node in sequence (called the head)
- Can now insert and delete from the front in $\Theta(1)$ time! Yay!
- (Inserting/deleting efficiently from back is also possible; you will do this in PS1)
- But now `get_at(i)` and `set_at(i, x)` each take $O(n)$ time... :(
- Can we get the best of both worlds? Yes! (Kind of...)

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)`
Linked List	n	n	1	n	n

Dynamic Array Sequence

- Make an array efficient for **last** dynamic operations
- Python "list" is a dynamic array
- **Idea!** Allocate extra space so reallocation does not occur with every dynamic operation
- **Fill ratio:** $0 \leq r \leq 1$ the ratio of items to space
- Whenever array is full ($r = 1$), allocate $\Theta(n)$ extra space at end to fill ratio r_i (e.g., $1/2$)
- Will have to insert $\Theta(n)$ items before the next reallocation
- A single operation can take $\Theta(n)$ time for reallocation
- However, any sequence of $\Theta(n)$ operations takes $\Theta(n)$ time
- So each operation takes $\Theta(1)$ time "on average"

Amortized Analysis

- Data structure analysis technique to distribute cost over many operations
- Operation has **amortized cost** $T(n)$ if k operations cost at most $\leq kT(n)$
- "$T(n)$ amortized" roughly means $T(n)$ "on average" over many operations
- Inserting into a dynamic array takes $\boldsymbol{\Theta(1)}$ **amortized** time
- More amortization analysis techniques in 6.046!

Dynamic Array Deletion

- Delete from back? $\Theta(1)$ time without effort, yay!
- However, can be very wasteful in space. Want size of data structure to stay $\Theta(n)$
- **Attempt:** if very empty, resize to $r = 1$. Alternating insertion and deletion could be bad...
- **Idea!** When $r < r_d$, resize array to ratio r_i where $r_d < r_i$ (e.g., $r_d = 1/4$, $r_i = 1/2$)
- Then $\Theta(n)$ cheap operations must be made before next expensive resize
- Can limit extra space usage to $(1 + \varepsilon)n$ for any $\varepsilon > 0$ (set $r_d = \frac{1}{1+\varepsilon}, r_i = \frac{r_d+1}{2}$)
- Dynamic arrays only support dynamic **last** operations in $\Theta(1)$ time
- Python List `append` and `pop` are amortized $O(1)$ time, other operations can be $O(n)$!
- (Inserting/deleting efficiently from front is also possible; you will do this in PS1)

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

6.006	Lecture 2	Feb. 6, 2020

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 (our focus today)
- set

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

(see both today)

Static sequence interface:

maintain a sequence of items/objects

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

subject to:

- build(X): make new DS for items in X (iterable with len())
- len(): return n
- iter_seq(): output $x_0, x_1, \ldots, x_{n-1}$ in seq. order
  $\equiv$ get_at(0), get_at(1), ..., get_at(len()-1)
- get_at(i): return object x_i of index i
  - special cases: get_first() $\equiv$ get_at(0)
    get_last() $\equiv$ get_at(len()-1)

logically equivalent
but implementations may differ

- set_at(i, x): change x_i to x

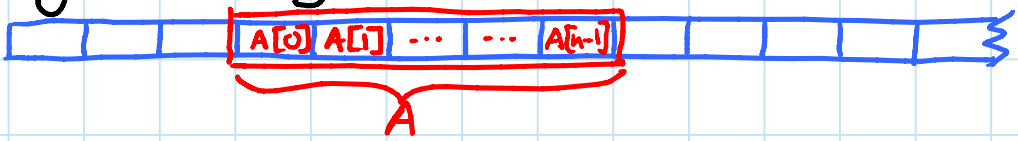
What data structure solves this?

static array A[0..n-1]

- A[i] stores x_i
- O(1) time per get_at/set_at/len
- O(n) time per build/iter_seq

Key: word RAM model of computation [L1]

- memory = array of w-bit words



- store A consecutively in memory

⇒ A[i] ≡ memory[address(A) + i]

(random access; starting index)

- if items we're storing don't fit in a word, store a pointer to them

(↑ memory address/index)

- assume address/pointer fits in a word so that memory[···] access is O(1) time

⇒ assume $w \geq \lg n$

(↑ problem size)

- need to be able to address entire problem of size n

⇒ word size is a parameter that grows with n ~ not constant (as problems get larger, so must machines)

- also generally assume that individual inputs (e.g. integers) fit in a word

Model for memory allocation:

can allocate an array of n words in $\Theta(n)$ time ⇒ space = O(time)

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_first & insert_last
- **delete_at**(i): shift $x_i \leftarrow x_{i+1} \leftarrow \cdots \leftarrow x_{n'-1} \leftarrow x_{n-1}$
  - special cases: delete_first & delete_last

$\Rightarrow$ set_at(i, x) $\equiv$ delete_at(i); insert_at(i, x)
(but set-at may be more efficient)

Special cases of interest:

- **stack** = get_last(), insert_last(), delete_last()
  top push pop
- **queue** = insert_last(), delete_first()
  enqueue dequeue
- **deque** = insert/delete_first/last()
  ↳ "double-ended queue"

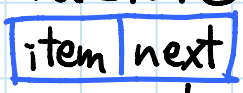
How well do static arrays do?

- $\Theta(n)$ for insert_at & delete_at ☹
- 2 linear costs:
  - shifting indices
  - resizing the array

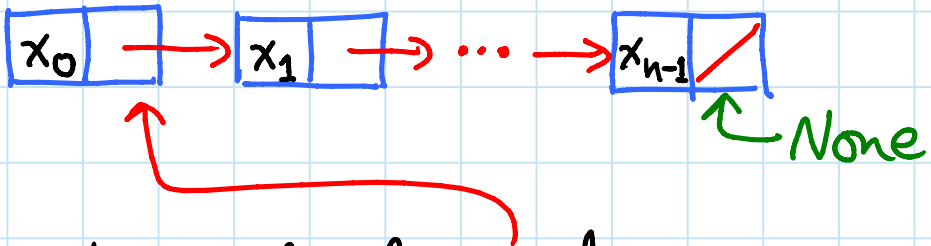
  $\Rightarrow$ even insert/delete_last slow! ☹

unrelated to Python's "list"

Linked lists: pointer-based data structure

- allocate 1 item at a time: 

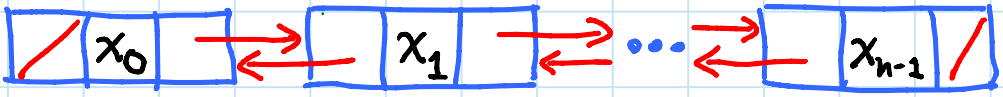
- connect items together with pointers:



- maintain global **head** pointer
  ⇒ fast get_first()
- insert/delete_first(x) also O(1) time:

⇒ solve (reversed) stack in O(1) time/op.!

Exercise: doubly linked list solves deque



- but get_at(i) is slow: O(i) by linear scan

Can we get the best of both worlds?

- get/set_at() performance of arrays
- insert/delete_first() 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: ↑ seq. length
  double size! size *= 2

- worst-case insert_last() still $\Theta(n)$
- but resize only when $n = 2^i$
- ⇒ n insert_last()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 ≤ k · T(n)
- "T(n) amortized" roughly means T(n) "on average", but averaged over all ops.
- e.g. insert_last() on a dynamic array takes O(1) amortized time

Dynamic array delete_last:

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

$\Rightarrow O(1)$ amortized for both insert/delete_last
- 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.

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.

- build(X): make new DS for items in iterable X
- len(): return $|S|$
- find(k): find item with key k (if exists)

Dynamic set interface: ≈ set/dict

- insert(x): add x to S (overwriting colliding key)
- delete(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(∞)
- iter_ord(): output items in sorted order by key

Dynamic ordered set interface:

- delete_min() $\equiv$ delete(find_min().key)
- delete_max() $\equiv$ delete(find_max().key)

Priority queue interface: special D.O.S. studied in L8

- insert(x)
- [find/delete_min() OR find/delete_max()]