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) *← Abstract Data type*

- 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**

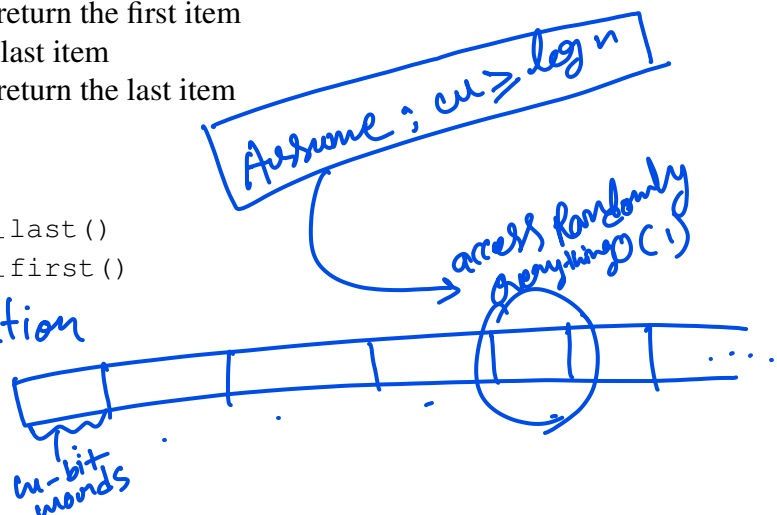## Sequence Interface (L02, L07)

- 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 ith item
	`set_at(i, x)`	replace the ith item with x
Dynamic	`insert_at(i, x)`	add x as the ith item
	`delete_at(i)`	remove and return the ith 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()`

Assume : $w \geq \log n$

access randomly → everything O(1)

Key : word RAM model of computation
— memory = array of w- bit words

w-bit words

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 ~~foor naw Hems Insert~~ → *In Static array: copying ele Size n to Size n+1 will take $O(n)$ time*
  - shifting all items after the modified item

		Operation, Worst Case $O(\cdot)$				
Data	Container	Static	Dynamic			
Structure	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

Cant do inseert last(x) in const time in Static Array, even though seems like it, as Size constant, [diagram] memory
x y , even though in Array no Shifting i.e In memory Shifting.

Linked List Sequence — *for Dynamic sequence operations*

- 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...)

insert last
fixable in LL *can improve* *(tail pointer) to last ele*
delete last (doubly LL)

Data Structure	Container	Static	Operation, Worst Case $O(\cdot)$		
				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 *doesn't change !!!*

original reallocation every +1 step *nehe +5 step*

- **Fill ratio:** $0 \le r \le 1$ the ratio of items to space

+5, +1 *— wont work* *as inc by constant factor*

- 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

Size

len

- However, any sequence of $\Theta(n)$ operations takes $\Theta(n)$ time

- So each operation takes $\Theta(1)$ time "on average"
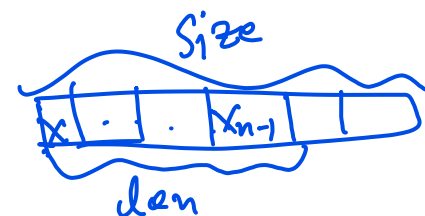
(len = size) × space

resize - new array allocation & copying cost *allocate new Array (+½ size) prev Array*

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 $\Theta(1)$ **amortized** time

- More amortization analysis techniques in 6.046!

(Geometric Series: dominated around last term

ej: O(last term of G.S))

theta(n) amortized means O(1) amortized
time over whole operation Individual
resizing at end may take O(n) time, but on
average to insert each element takes O(1)
time, (averaging over the operation
sequence)

Dynamic Array Deletion

don't have to resize
decrease length

- Delete from back? $\Theta(1)$ time without effort, yay!

can have n=0
but array size θ(n)

- 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	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

Operation, Worst Case $O(\cdot)$