

## 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:  $(x_0, x_1, x_2, \dots, x_{n-1})$  (zero indexing)
- (use  $n$  to denote the number of items stored in the data structure)
- Supports sequence operations:

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

- Special case interfaces:

<b>stack</b>	insert_last(x) and delete_last()
<b>queue</b>	insert_first(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	<code>build(x)</code> <code>len()</code>	given an iterable $x$ , build sequence from items in $x$ return the number of stored items
Static	<code>find(k)</code>	return the stored item with key $k$
Dynamic	<code>insert(x)</code> <code>delete(k)</code>	add $x$ to set (replace item with key $x.key$ if one already exists) remove and return the stored item with key $k$
Order	<code>iter_ord()</code> <code>find_min()</code> <code>find_max()</code> <code>find_next(k)</code> <code>find_prev(k)</code>	return the stored items one-by-one in key order return the stored item with smallest key return the stored item with largest key return the stored item with smallest key larger than $k$ 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		
	<code>build(x)</code>	<code>get_at(i)</code> <code>set_at(i, x)</code>	<code>insert_first(x)</code> <code>delete_first()</code>	<code>insert_last(x)</code> <code>delete_last()</code>	<code>insert_at(i, x)</code> <code>delete_at(i)</code>
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		
	<code>build(X)</code>	<code>get_at(i)</code> <code>set_at(i, x)</code>	<code>insert_first(x)</code> <code>delete_first()</code>	<code>insert_last(x)</code> <code>delete_last()</code>	<code>insert_at(i, x)</code> <code>delete_at(i)</code>
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  $\Theta(1)$  **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		
	<code>build(x)</code>	<code>get_at(i)</code> <code>set_at(i, x)</code>	<code>insert_first(x)</code> <code>delete_first()</code>	<code>insert_last(x)</code> <code>delete_last()</code>	<code>insert_at(i, x)</code> <code>delete_at(i)</code>
Array	$n$	1	$n$	$n$	$n$
Linked List	$n$	$n$	1	$n$	$n$
Dynamic Array	$n$	1	$n$	$1_{(a)}$	$n$

MIT OpenCourseWare  
<https://ocw.mit.edu>

6.006 Introduction to Algorithms  
Spring 2020

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>