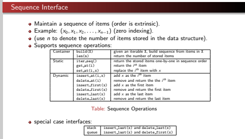


Sequence Interface

- Maintain a sequence of items (order is extrinsic).
- Example: $(x_0, x_1, x_2, \ldots, x_{n-1})$ (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(n)`	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 the i^{th} item
	`insert_first(x)`	add x as the first item
	`delete_first(x)`	remove and return the first item
	`insert_last(x)`	add x as the last item
	`delete_last(x)`	remove and return the last item

Table: Sequence Operations

- special case interfaces:

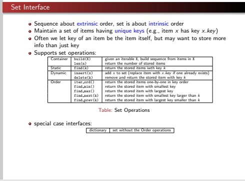
stack	`insert_last(x)` and `delete_last(x)`
queue	`insert_last(x)` and `delete_first(x)`

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(n)`	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 the i^{th} item
	`insert_first(x)`	add x as the first item
	`delete_first(x)`	remove and return the first item
	`insert_last(x)`	add x as the last item
	`delete_last(x)`	remove and return the last item

Table: Sequence Operations

Note that `insert_delete` operations change the rank of all items after the modified item



Set Interface

- Sequence about extrinsic order, set is about intrinsic order
- Maintain a set of items having unique keys (e.g., item x has key $x.key$)
- 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)`
	`len(n)`
		given an iterable `X`, build sequence from items in `X`
		return the number of stored items

Static	`find(k)`	return the stored items iwth key k

Dynamic	`insert(x)`
	`delete(k)`
		add x to set (replace item with $x.key$ if one already exists)
		remove and return the stored item with key k

Order	`iter_ord()`
	`find_min()`
	`find_max()`
	`find_next(k)`
	`find_prev(k)`
		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

Table: Set Operations

- special case interfaces:

dictionary	set without the Order operations

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)`
	`len(n)`
		given an iterable `X`, build sequence from items in `X`
		return the number of stored items

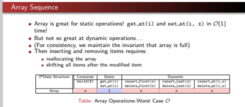
Static	`find(k)`	return the stored items iwth key k

Dynamic	`insert(x)`
	`delete(k)`
		add x to set (replace item with $x.key$ if one already exists)
		remove and return the stored item with key k

Order	`iter_ord()`
	`find_min()`
	`find_max()`
	`find_next(k)`
	`find_prev(k)`
		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

Table: Set Operations

Note that `find` operations return `None` if no qualifying item exists



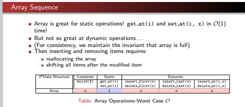
Array Sequence

- Array is great for static operations! `get_at(i)` and `set_at(i, x)` in $\mathcal{O}(1)$ 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

3*Data Structure	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i)`	`insert_first(x)` `delete_first(x)`	`insert_last(x)` `delete_last(x)`	`insert_at(i,x)` `delete_at(i,x)`
Array	n	1	n	n	n

Table: Array Operations-Worst Case $\mathcal{O}$

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.



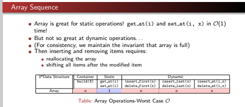
Array Sequence

- Array is great for static operations! `get_at(i)` and `set_at(i, x)` in $\mathcal{O}(1)$ 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

3*Data Structure	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i)`	`insert_first(x)` `delete_first(x)`	`insert_last(x)` `delete_last(x)`	`insert_at(i,x)` `delete_at(i,x)`
Array	n	1	n	n	n

Table: Array Operations-Worst Case $\mathcal{O}$

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.



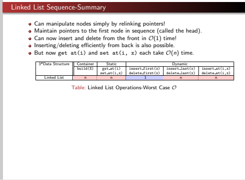
Array Sequence

- Array is great for static operations! `get_at(i)` and `set_at(i, x)` in $\mathcal{O}(1)$ 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

3*Data Structure	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i)`	`insert_first(x)` `delete_first(x)`	`insert_last(x)` `delete_last(x)`	`insert_at(i,x)` `delete_at(i,x)`
Array	n	1	n	n	n

Table: Array Operations-Worst Case $\mathcal{O}$

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 $\mathcal{O}(1)$ 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.



Linked List Sequence-Summary

- 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 $\mathcal{O}(1)$ time!
- Inserting/deleting efficiently from back is also possible.
- But now get at(i) and set at(i, x) each take $\mathcal{O}(n)$ time.

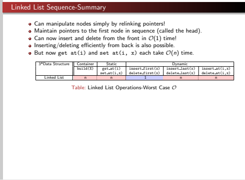
3*Data Structure	Container	Static	Dynamic		
	build(X)	get_at(i) set_at(i,x)	insert_first(x) delete_first(x)	insert_last(x) delete_last(x)	insert_at(i,x) delete_at(i,x)
Linked List	n	n	1	n	n

Table: Linked List Operations-Worst Case $\mathcal{O}$

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):
        self.item = x
        self.next = None
    # O(1)

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



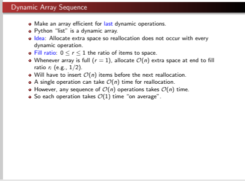
Linked List Sequence-Summary

- 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 $\mathcal{O}(1)$ time!
- Inserting/deleting efficiently from back is also possible.
- But now `get at(i)` and `set at(i, x)` each take $\mathcal{O}(n)$ time.

3*Data Structure	Container	Static	Dynamic		
	`build(X)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first(x)`	`insert_last(x)` `delete_last(x)`	`insert_at(i,x)` `delete_at(i,x)`
Linked List	n	n	1	n	n

Table: Linked List Operations-Worst Case $\mathcal{O}$

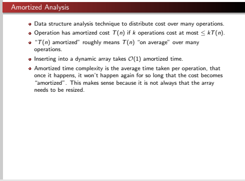
Such data structures are sometimes called pointer-based or linked and are much more flexible than array-based data structures because their constituent items can be stored anywhere in memory. A linked list stores the address of the node storing the first element of the list called the head of the list, 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 $\mathcal{O}(1)$ time. However, the only way to find the ^{ith} 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.



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 $\mathcal{O}(n)$ extra space at end to fill ratio r_i (e.g., 1/2).
- Will have to insert $\mathcal{O}(n)$ items before the next reallocation.
- A single operation can take $\mathcal{O}(n)$ time for reallocation.
- However, any sequence of $\mathcal{O}(n)$ operations takes $\mathcal{O}(n)$ time.
- So each operation takes $\mathcal{O}(1)$ time "on average".

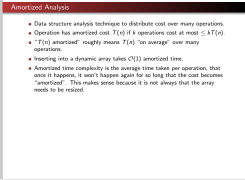
The array's dynamic sequence operations require linear time with respect to the length of array A. Is there another way to add elements to an array without paying a linear overhead transfer cost each time you add an element? One straight-forward way to support faster insertion would be to over-allocate 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.



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 $\mathcal{O}(1)$ amortized time.
- Amortized time complexity is the average time taken per operation, that once it happens, it won't happen again for so long that the cost becomes "amortized". This makes sense because it is not always that the array needs to be resized.

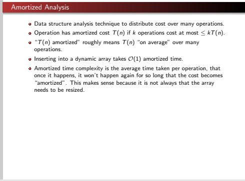
Then how does Python support appending to the end of a length n Python List in worst-case $\mathcal{O}(1)$ time? The answer is simple: it doesn't. Sometimes appending to the end of a Python List requires $\mathcal{O}(n)$ time to transfer the array to a larger allocation in memory, so sometimes appending 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 $\mathcal{O}(n)$ time (i.e. such linear time transfer operations do not occur often), so insertion will take $\mathcal{O}(1)$ 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.



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 $\mathcal{O}(1)$ amortized time.
- Amortized time complexity is the average time taken per operation, that once it happens, it won't happen again for so long that the cost becomes "amortized". This makes sense because it is not always that the array needs to be resized.

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 $\mathcal{O}(n)$ 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.



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 $\mathcal{O}(1)$ amortized time.
- Amortized time complexity is the average time taken per operation, that once it happens, it won't happen again for so long that the cost becomes "amortized". This makes sense because it is not always that the array needs to be resized.

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.