* "containers" we encountered

want O(#NZ) storage →

sparse array w/ ordered dense index

☆ ① Θ(#NZ) storage

② indexed access in Θ(log #NZ)

③ iterative access w/ index

④ "fixed" max length

⑤ O(#NZ) insertion/removal → C++ ~~arr~~ (template)

C/C++ dense int array

restricted access: stack (FILO)

① Θ(N) storage (small)

☆ ② indexed (random) access in Θ(1) for R/W

③ iterative (sequential) access w/ pointer or index (Θ(1) per step)

④ fixed max length

⑤ O(N) insertion/removal

want O(1) insertion/removal & dynamic max-length →

doubly linked list

restricted access: deque (stack + queue + ...)

① Θ(N) storage

~~② indexed access in O(N)~~ often not implemented

③ iterative access w/ pointer

☆ ④ dynamic max-length

☆ ⑤ O(1) insertion/removal

+ circular access → circular linked list or circular array

want only O(1) insertion →

singly linked list

① smaller Θ(N) storage

☆ ⑤ O(1) insertion faster

⑤' O(N) removal

③, ④

restricted access ⇓ queue (FIFO)

mixed : 2-D array (e.g. array of array)

other sparse array (e.g. w/ linked list)

actual C++ deque (linked list of arrays)

* abtraction : get the "essence" of what we need

① save implemention efforts (e.g. type abstration by template)

② "easy" change of implemention

singly or doubly ? "same functionality", different underlying implementation

* functionality abstraction (contract)

dense array & sparse array & extendable (dense) array

$\boxed{\text{vector}}$: indexed (random) access

at(i) (random access, R)

set (i, e) (random access, W)

insert (i, e) (insertion)

erase (i) (removal)

* extendable array

if internal array A overflows

grow the array { allocate new array B $O(1)$
copy contents to the new array B $O(n)$ ↑ #elem
remove A $O(1)$
and assign B to A

consider M "pushes" to the array

① size(B) = size(A) + 1

n		size(A)
1		1
	↓ !, allocate, copy(1),	
2		2
	↓ !, allocate, copy(2),	
3		3
	↓ !, allocate, copy(3)	
4		4
⋮		
M	copy(M−1)	

# allocate $= O(M)$ \# copy $= \frac{(M-1)(M)}{2} = O(M^2)$

② size(B) = size(A) * 2

n		size(A)
1		1
	↓ !, allocate, copy(1)	
2		2
	↓ !, allocate, copy(2)	
3		4
	↓	
4		4
	↓ !, allocate, copy(4)	
5		8
	↓	
6		
	↓	
2^k		
	↓ !, allocate, copy(2^k)	
$M=2^k+1$		

# allocate $= k+1 = O(\log M)$

# Copy $= 1+2+\cdots+2^k$
$= 2^{k+1}-1$
$= O(M)$

② better than ①, implemented in stl::vector
(caveats ?) w/ "reserve" functionality

* functionality abstraction
doubly & singly linked list & array

list : iterative (positional) access

insert (p, e)	insert at position
elem (p)	element at position
begin ()	starting position
p != end() ⇐ isEnd (p)	is p the end ?
nextOf (p)	go to next element of p
erase (p)	

iterator (p) : abstraction for
- index in sparse array
- index / pointer in dense array
- pointer in linked list

	sparse array	dense array	list
begin	0	& arr[0]	head
end	n+1	& arr[n+1]	NULL
nextOf (p)	p+1	p++	p→next
elem (p)	at(p)	(*p)	p → value

- iterator < container-type > , "safe" pointer in some sense
- overload "++" to do nextOf , override "*" to do { -- ? / +5 ?
-
```
int sum = 0;
for ( iterator<list<int>>p = c.begin(); p!= c.end(); p++ ){
              can now "freely" change this one
     sum += (*p);
}
```

STL list : doubly linked list

* sequence : vector + list + i ⇔ p