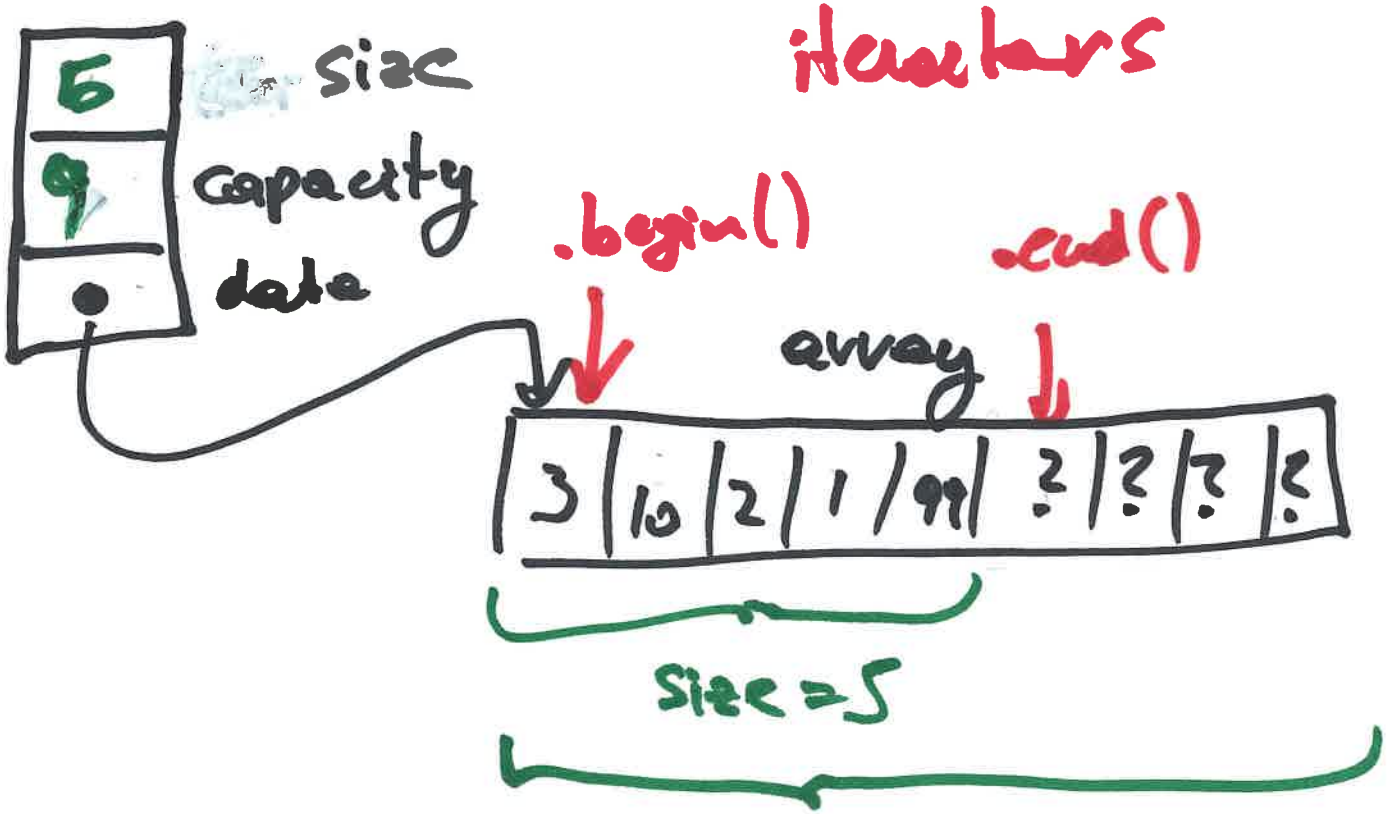
CS 2341

Chapter 3

Lists, Stacks, and Queues

DS Vector (std::vector)

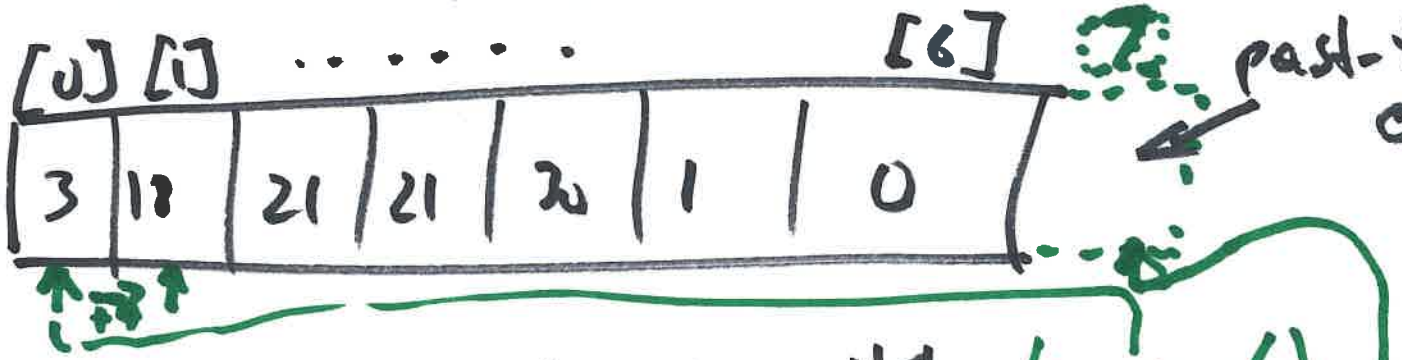


cap = 4 → 8
size = 4 → 5

2x size every time we run out of capacity

Iterators for vectors

```
std::vector<int> v = { 3,18,21,21,34,1,0 };
```




```
std::vector<int>::iterator it1=v.begin()
------------ " ------------ it2=v.end()
```

```
it1++; // go to the next element
```

```
std::cout << *it1 << "\n"; ⇒ 18
```

↑ "dereference" the iterator

```
1   #ifndef MATRIX_H
2   #define MATRIX_H
3
4   #include <vector>
5   using namespace std;
6
7   template <typename Object>
8   class matrix
9   {
10    public:
11      matrix( int rows, int cols ) : array( rows )
12      {
13          for( auto & thisRow : array )
14              thisRow.resize( cols );
15      }
16
17      matrix( vector<vector<Object>> v ) : array{ v }
18        { }
19      matrix( vector<vector<Object>> && v ) : array{ std::move( v ) }
20        { }
21
22      const vector<Object> & operator[]( int row ) const
23        { return array[ row ]; }
24      vector<Object> & operator[]( int row )
25        { return array[ row ]; }
26
27      int numrows( ) const
28        { return array.size( ); }
29      int numcols( ) const
30        { return numrows( ) ? array[ 0 ].size( ) : 0; }
31    private:
32      vector<vector<Object>> array;
33  };
34  #endif
```

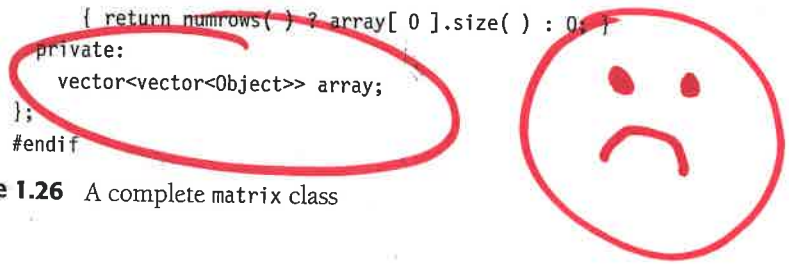
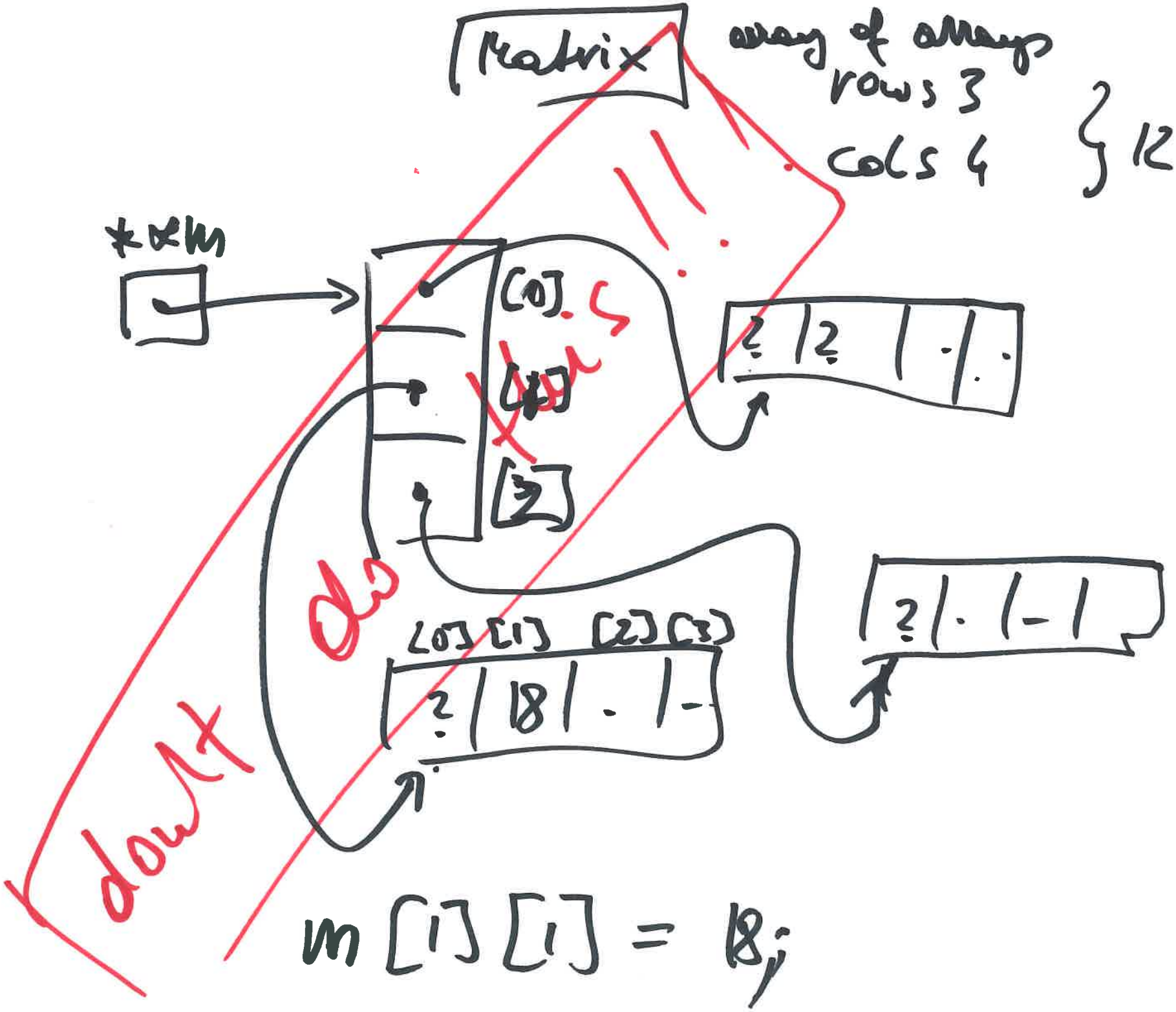


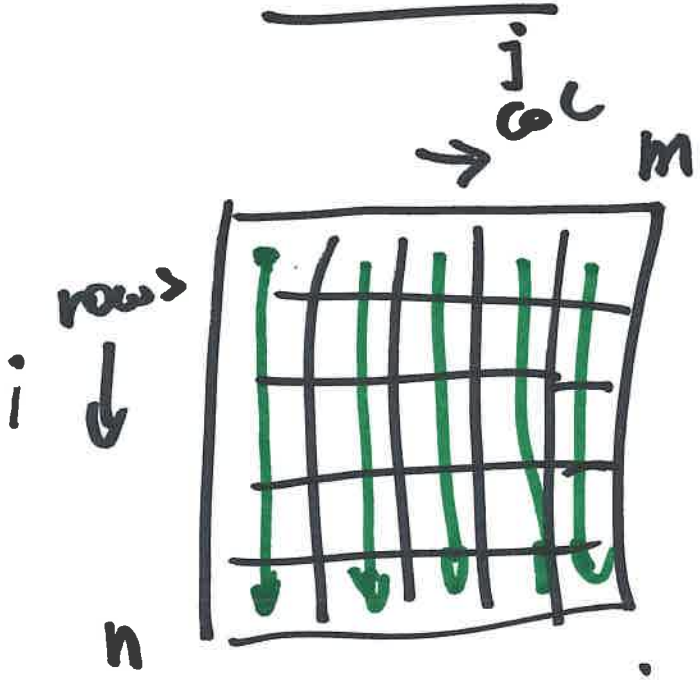
Figure 1.26 A complete `matrix` class

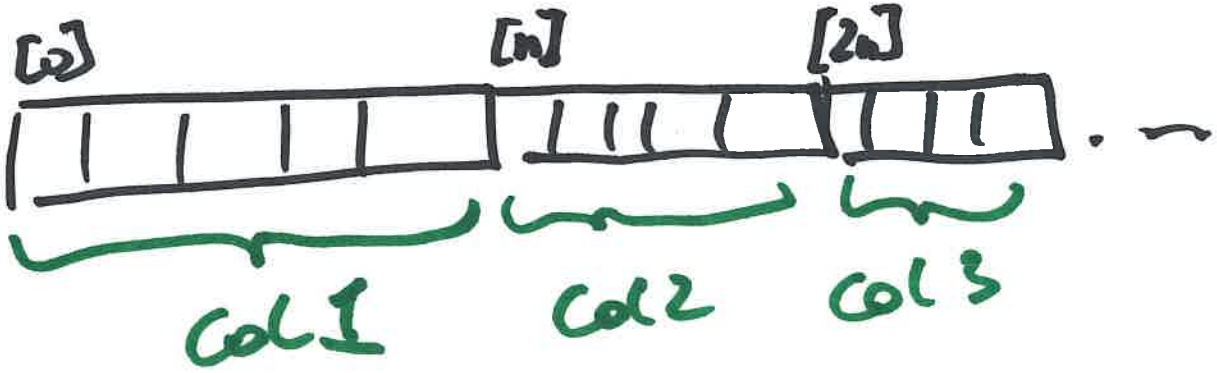
1.7.2 operator[]

The idea of `operator[]` is that if we have a `matrix` `m`, then `m[i]` should return a vector corresponding to row `i` of `matrix` `m`. If this is done, then `m[i][j]` will give the entry in position `j` for vector `m[i]`, using the normal `vector` indexing operator. Thus, the `matrix` `operator[]` returns a `vector<Object>` rather than an `Object`.


Matrix
array of arrays
rows 3
cols 4
} 12
**m
[0]
[1]
[2]
[0] [1] [2] [3]
m[1][1] = 18;

Matrix n rows x m col

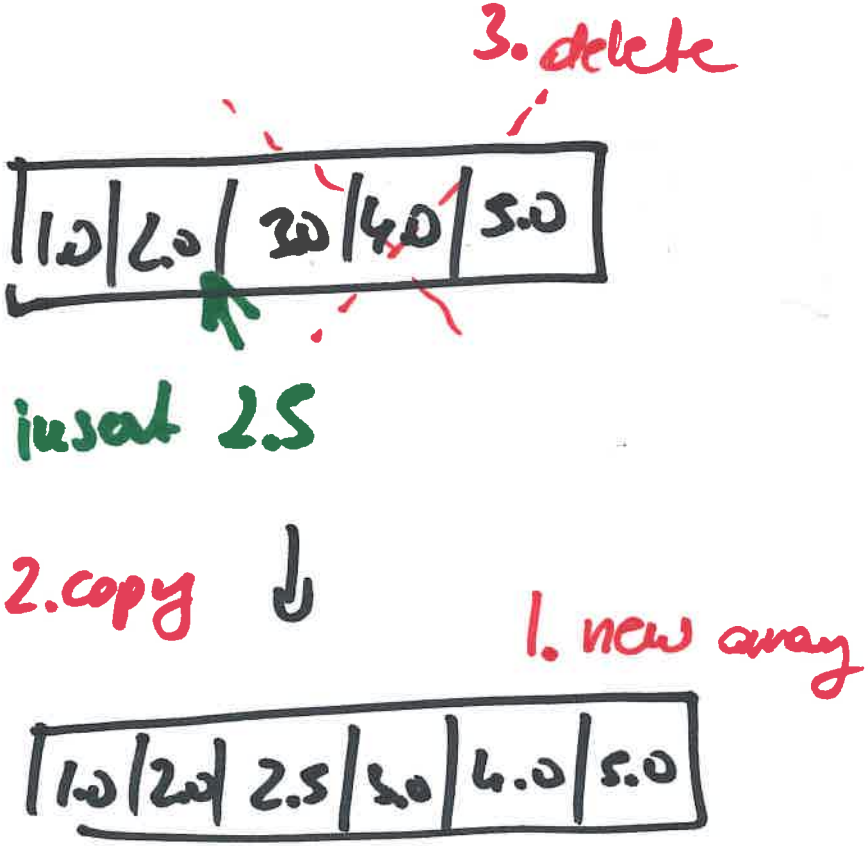




column-major orientation

index into array: i, j ⇒ j x n + i

Insert into a vector (array)



1. allocate memory

O(1)

2. copy + insert

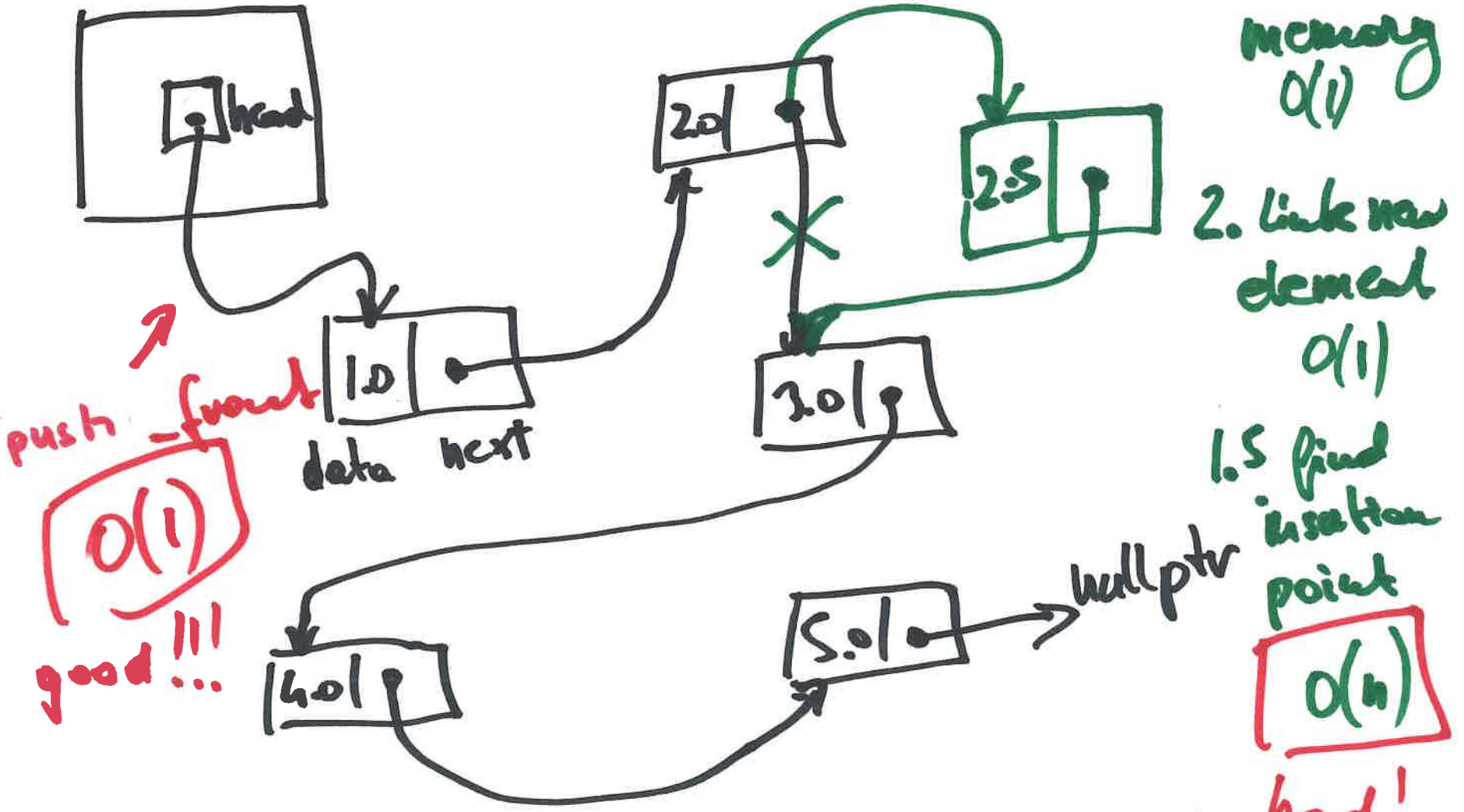
O(n) bad!

3. delete old array from memory

O(1)

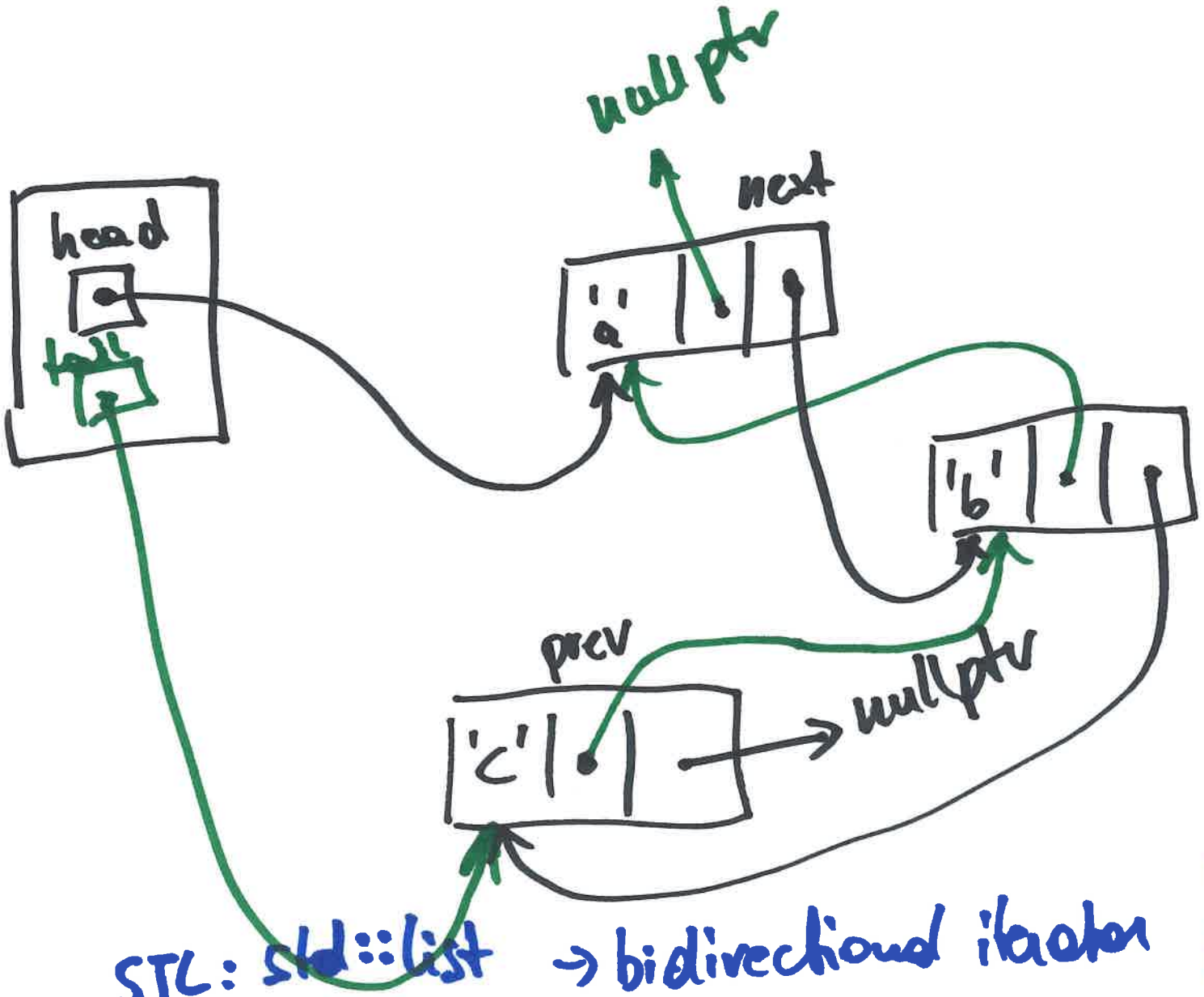
Linked List (singly-linked list)

insert (2.5 after 2)



STL: forward_list with forward iterator (only ++)

Doubly-linked List



STL: std::list → bidirectional iterator it++ it--

Time Complexity

Each element is allocated in a linked list individually, so we do not need contiguous memory and we do not need to copy/move data for insertion and deletion. What is the time complexity for the following operations using lists or arrays (Big-O notation)?

Operation	Array	STL vector	Singly Linked List	Doubly Linked List
insert a new element at a random location	O(n)	O(n)	O(n)	O(n)
insert a new element (front)	O(n)	O(n)	O(1)	O(1)
insert a new element (back)	O(n)	O(1)/O(n)*	O(n)	O(1)
access an element (with index)	O(1)	O(1)	O(n)	O(n)
access an element (with iterator)	—	O(1)	O(1)	O(1)
find an element by value	O(n)	O(n)	O(n)	O(n)
delete an element	O(n)	O(n)***	O(n)***	O(n)***
delete the whole data structure	O(1)	O(1)	O(n)	O(n)

* no capacity left

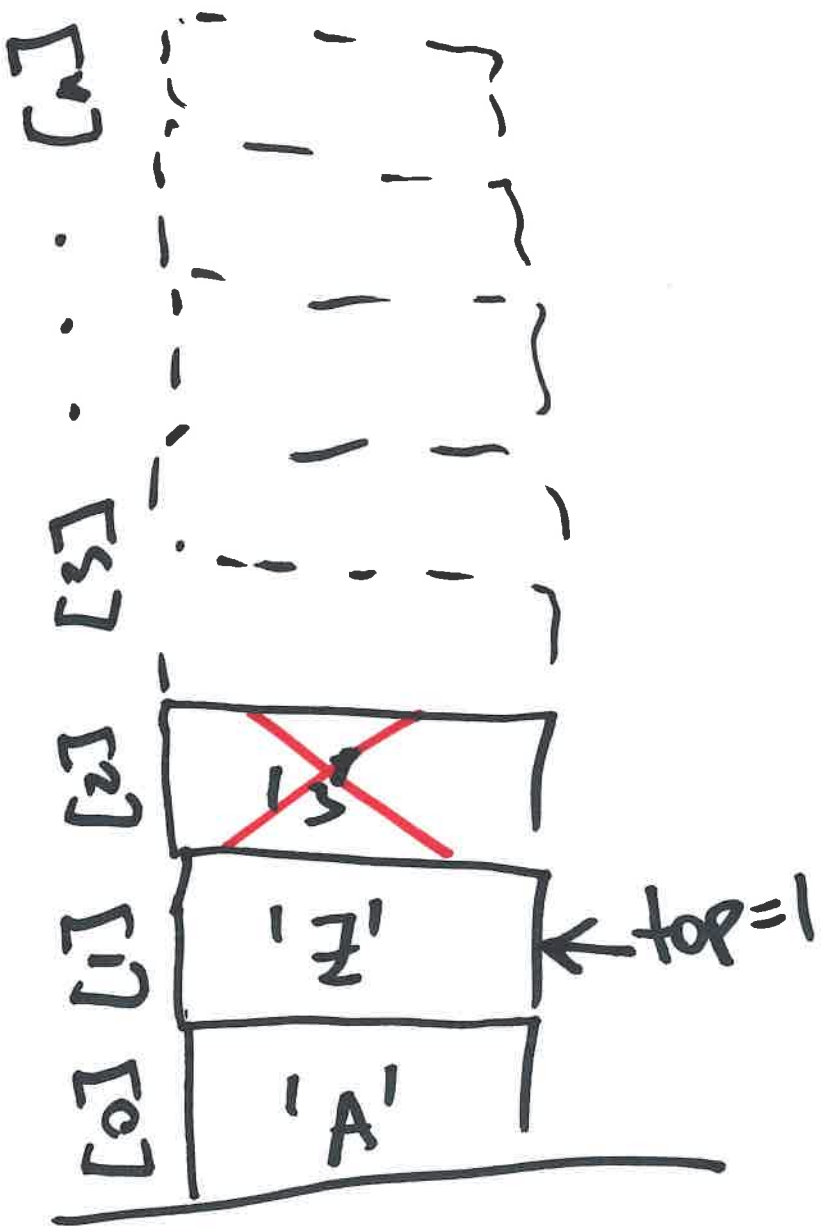
** if sorted → binary search → O(log n)

*** last element → O(1)

**** first element O(1)

Stack

LIFO (last-in first-out)



push('A')
push('Z')
push('3')
} O(1)

pop() → '3' O(1)

Implement as array

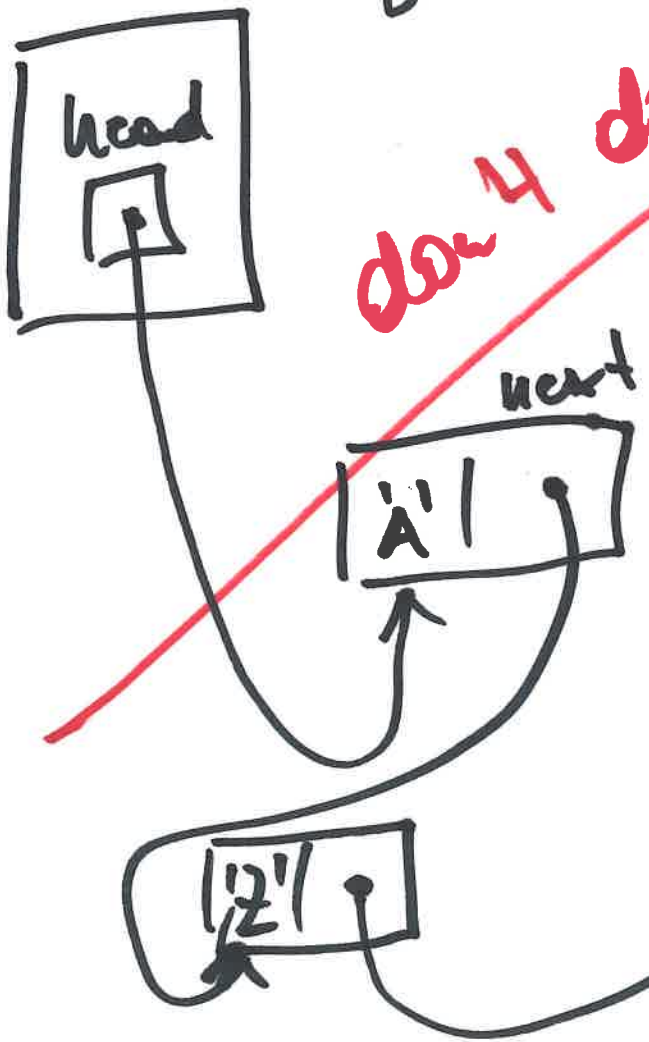
empty stack top = -1

size = top + 1

Stack List implementation

using push_back

don't do this!!

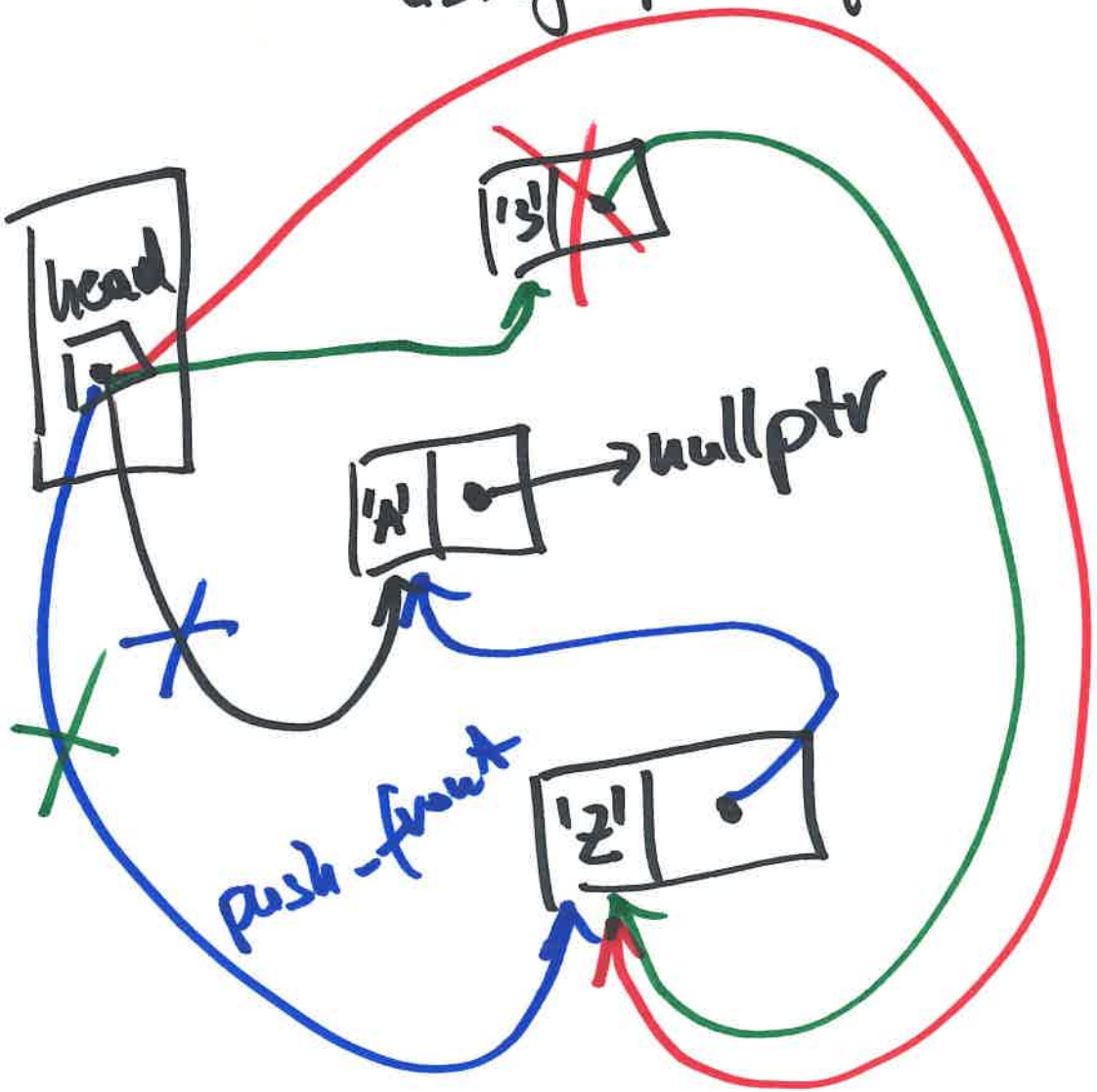


push('A')
push('Z') O(n)
push('3')

pop() -> 3 O(n)

Stack List

using push_front



push('A') O(1)

push('Z')

push('3')

pop() → 3 O(1)

Check for balanced brackets

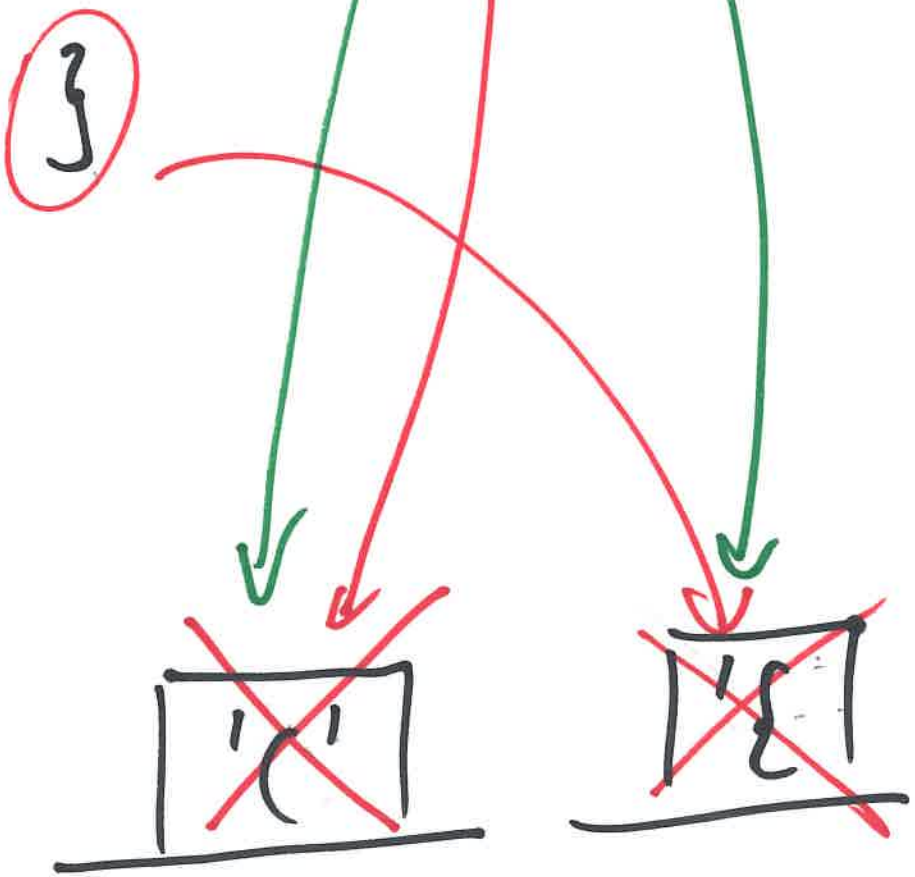
```
#include <iostream>

int main() {

    std::cout << "Hello World!" << "\n";

    return 0;

}
```



Empty stack means brackets are balanced

Postfix notation

arg 1	arg 2	operator	
5	9	+	⇒ 14

prefix notation

operator	arg 1	arg 2	
+	5	9	⇒ 14

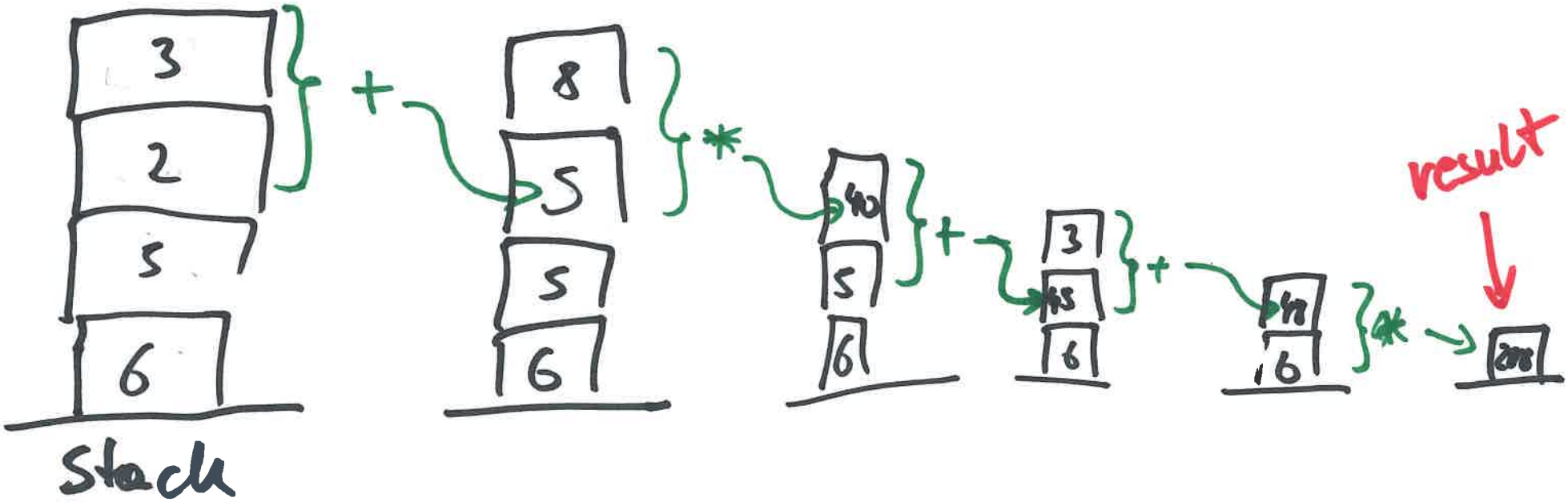
Infix notation

arg 1	operator	arg 2	
5	+	9	⇒ 14

Evaluate a Postfix Expression

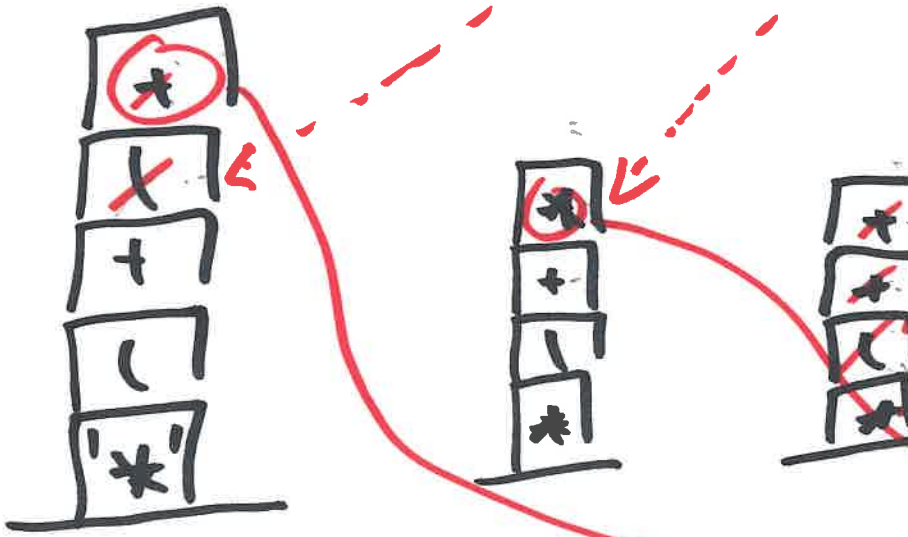
6 5 2 3 + 8 * + 3 + * = ?

- read left-to-right
- numbers → stack
- operators pop the top 2 elements from the stack and push the result back on the stack



Infix to Postfix

Input : 6 * (5 + (2 + 3) * 8 + 3)

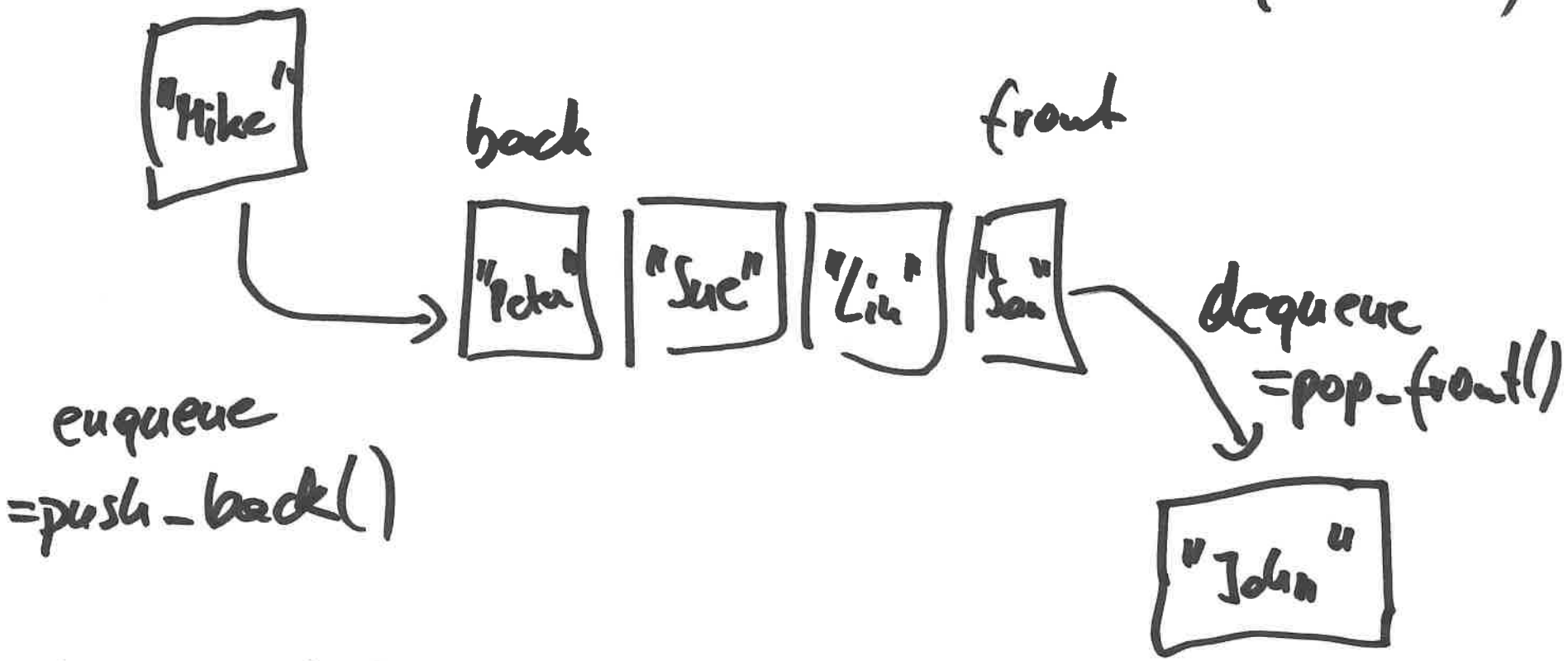


- operands go to output
- operators and parentheses go to stack (be careful with the order of operations)

Output : 6 5 2 3 + 8 * 3 + *

Queue

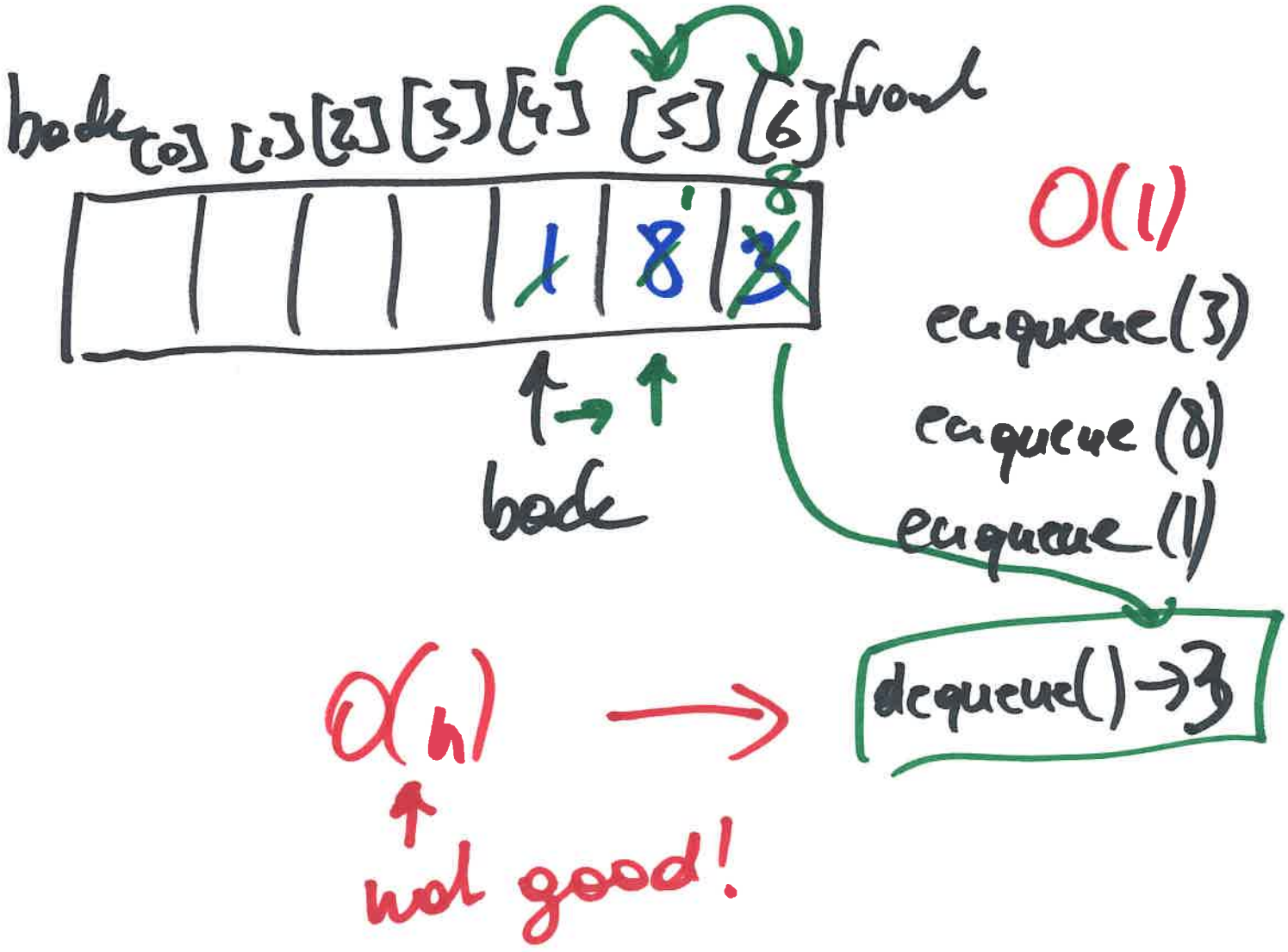
FIFO (first in first out)



Implementations:

- Vector / array → circular buffer
- doubly linked list
- std::deque

Queue Array


back
[0] [1] [2] [3] [4] [5] [6]
front
1
8
3
back
O(1)
enqueue(3)
enqueue(8)
enqueue(1)
dequeue()→3
O(n)
not good!

Queue

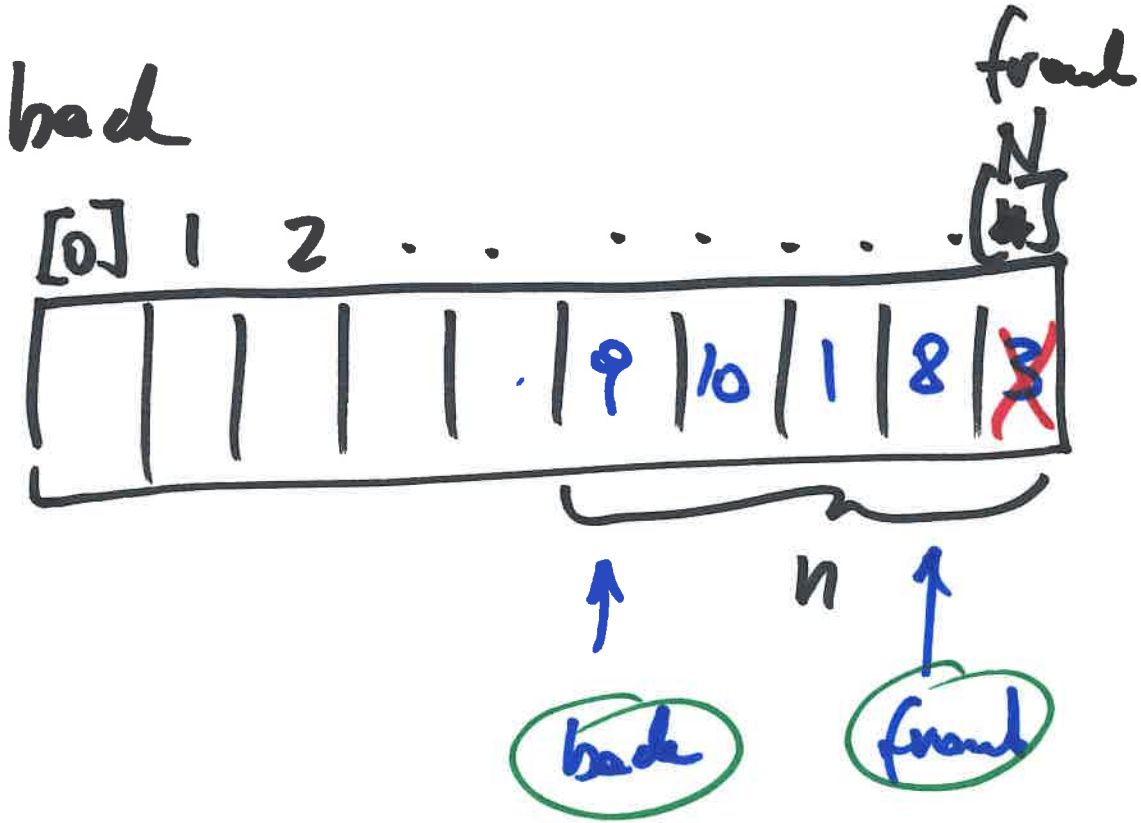
Array (circular buffer)

O(1)

enqueue(3)

enqueue(8)

moves "back"



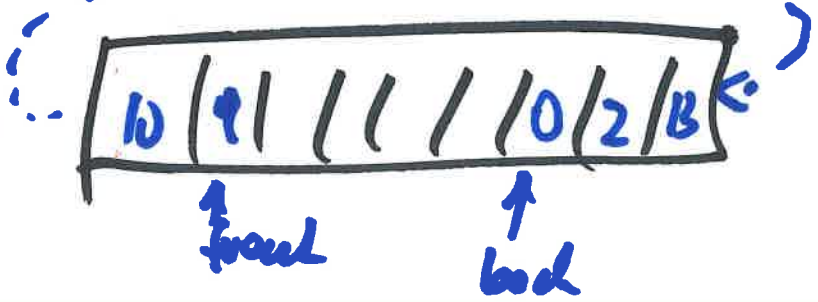
O(1)

dequeue()

moves "front"

N

→ circular buffer



Queue doubly-linked list