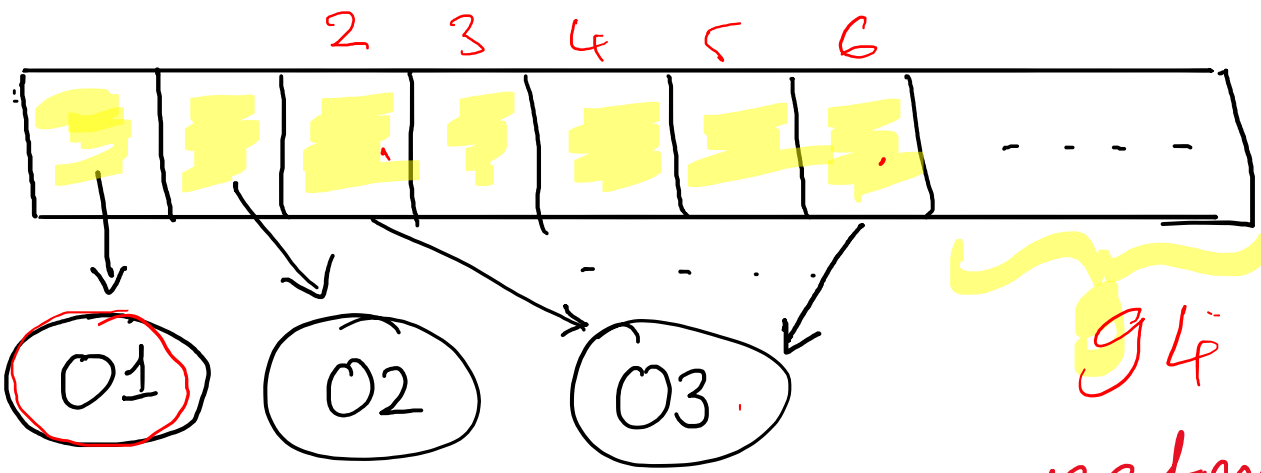
COL106 - Data Structures and Algorithms

Array datastructure in Java

```
Student[] roster = new Student[100];
```



student[2] = student[6]

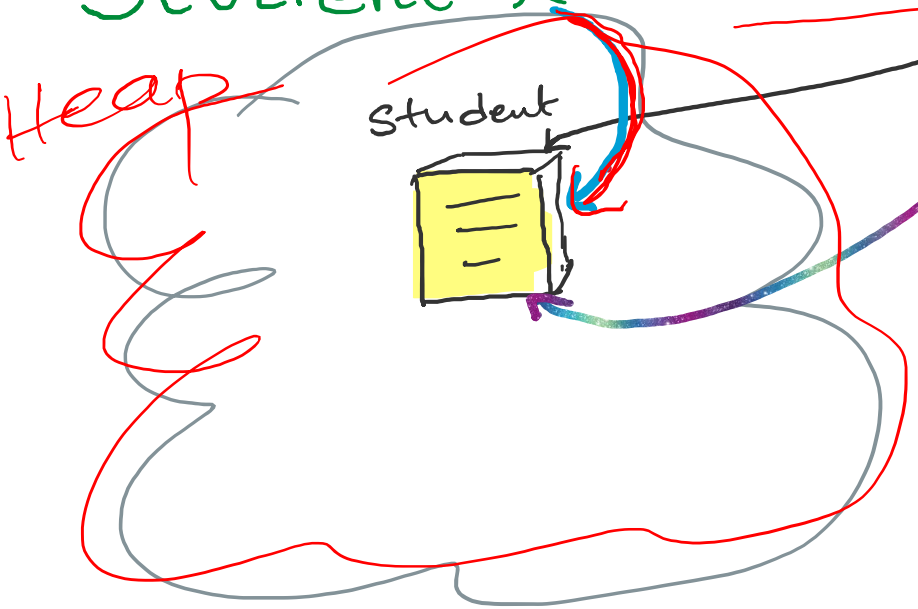
94 blank

Each index contains a reference to an object.

Reference – the "handle" to an object

Every object in Java can only be accessed through its "reference"

```
Student x = new Student();
```



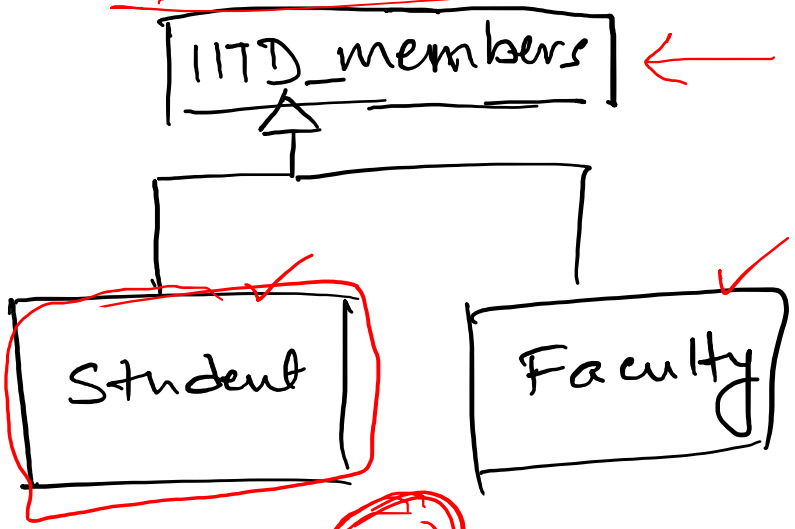

```
y = x
```

x holds the reference to be able to access the object.

x can be assigned to another reference of the same type

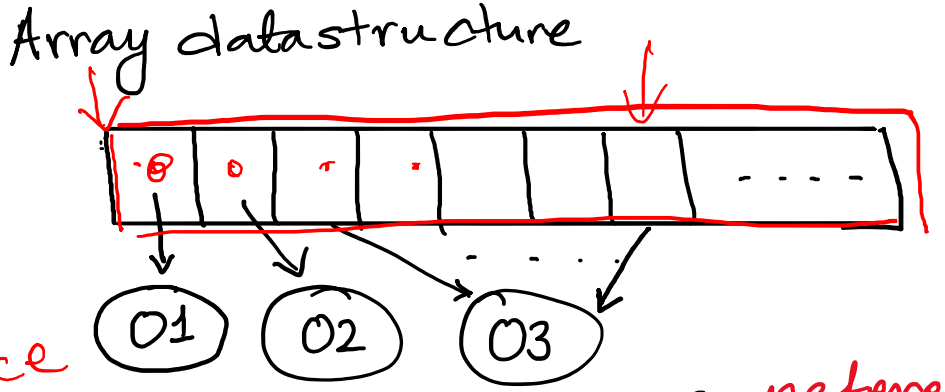
Student x = new Student();

subClasses }
interface }



IITD_member z = x;

Now z will behave as a reference to IITD_member object, ⇐ not as student




```
Student[] roster
    = new Student[100];
```

m
t(i)
reference
size

Each index contains a reference to an object

⇒ Ensures that all references are stored contiguously in memory

⇒ Access to objects through references

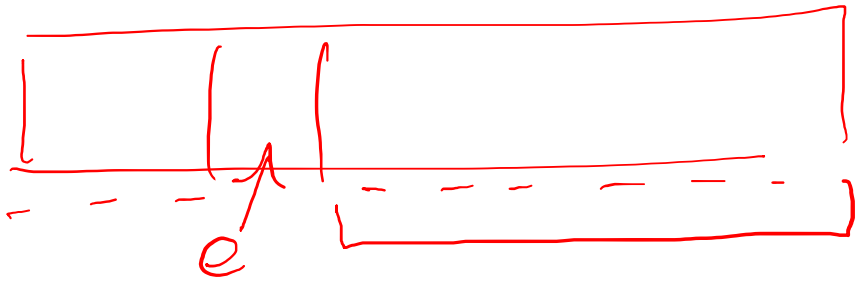
⇒ Has a predefined size which cannot be extended

List ADT

Java offers an interface called java.util.List with the following methods (among more)

- add(i, e): insert a new element e at i
- set(i, e): replace the element at i with e; returns element it replaced.
- get(i): get the element at i
- remove(i): remove and return the element at i

List ADT



`add(i, e)`:

Q1: what is the effect on elements already in the collection?

All subsequent elements are shifted one index later in the list.

Q2: what happens when $i < 0$ or $i >$ size() of the list?

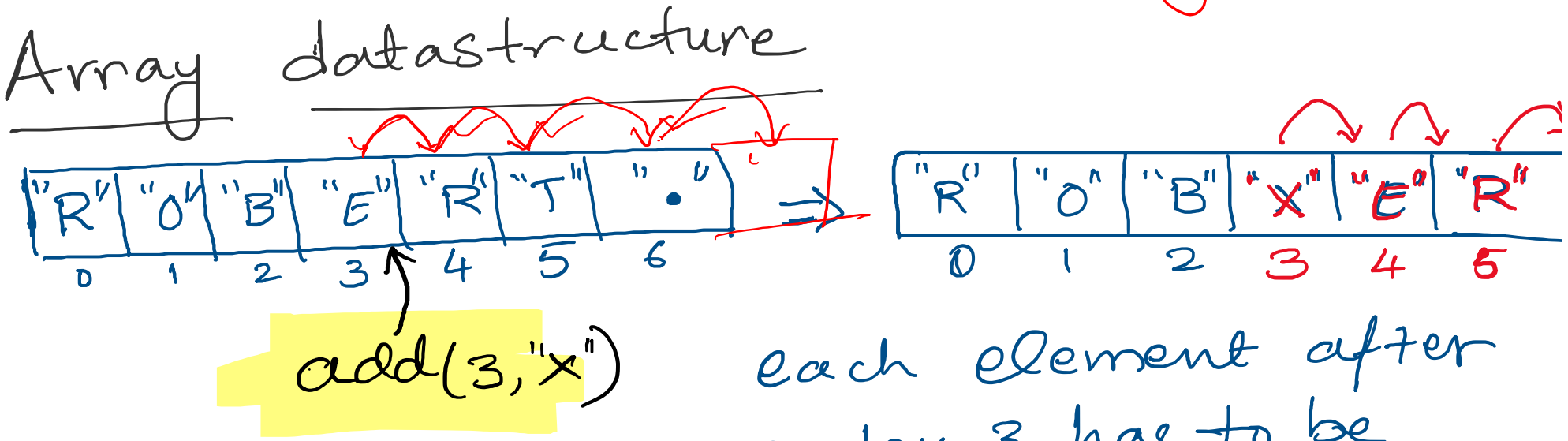
Error !!

A sequence of operations on List

Operation	Return value	State
add(0, A)	—	(A)
add(0, B)	-	(B, A)
get (1)	A	(B, A)
set (2, C)	"error"	(B, A)
add (2, C)	—	(B, A, C)
add (4, D)	"error"	(B, A, C)
set (1, X)	A	(B, X, C)
remove (2)	C	(B, X)

Implementing List ADT

ArrayList

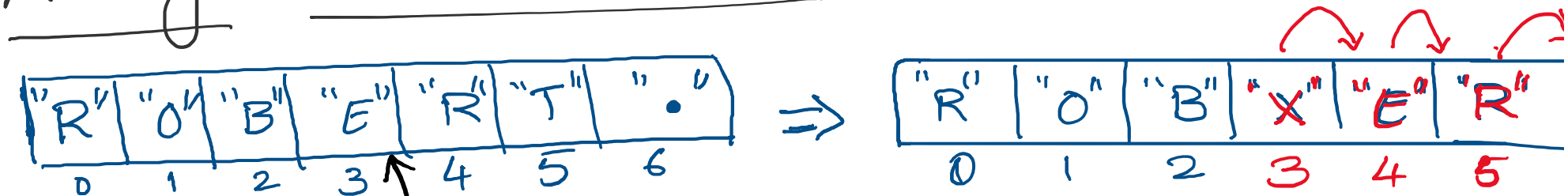


each element after index 3 has to be shifted by one index

Similarly when an element is removed, all elements have to be moved to a lower index.

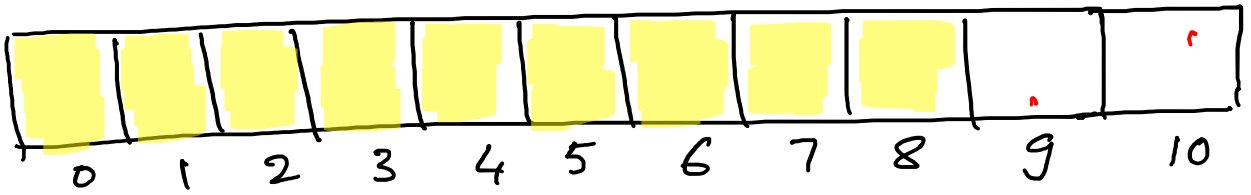
Array Implementation of List ADT (Array List)

Array datastructure



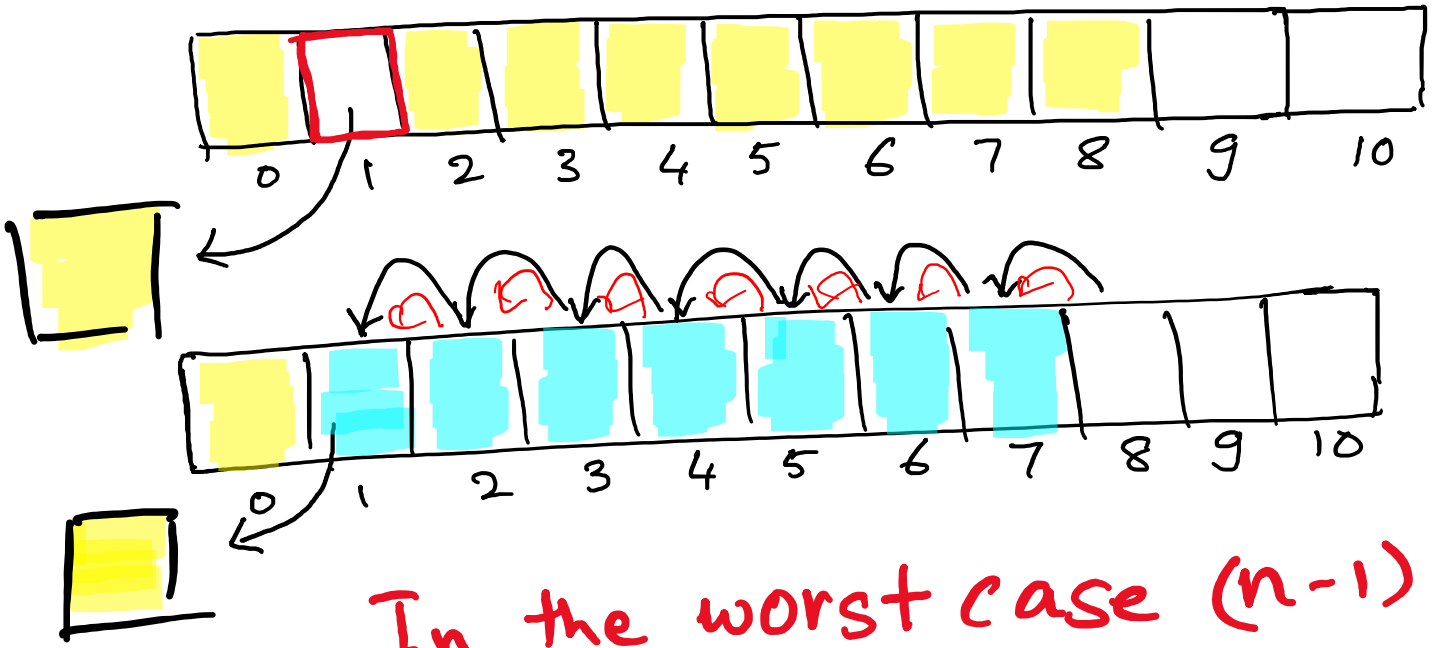
each element after index 3 has to be shifted by one index

Similarly when an element is removed, all elements have to be moved to a lower index.



A (1 size)

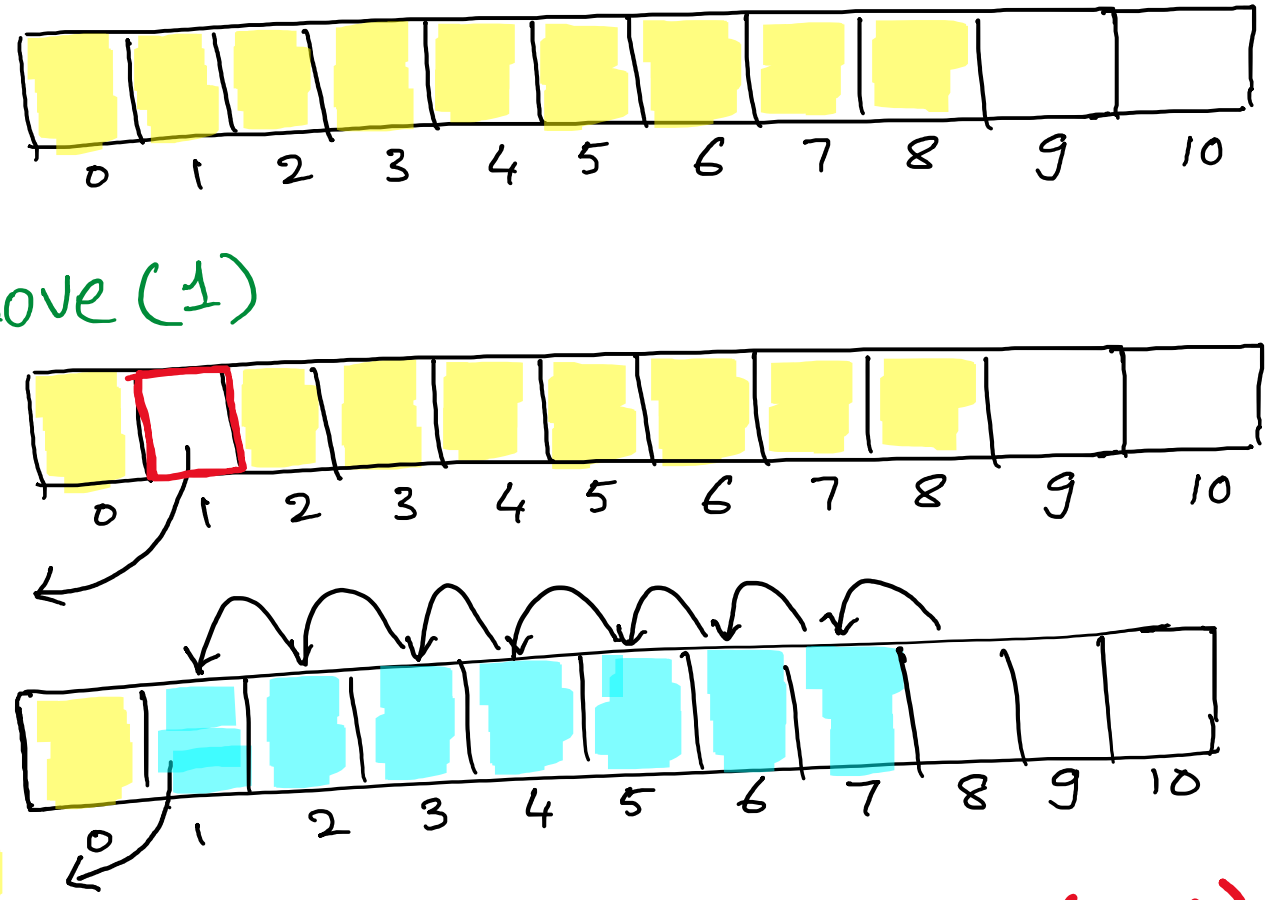
remove(1);



A

A

In the worst case (n-1) elements need to be copied!



Arrays can not have gaps in between

In the worst case (n-1) elements need to be copied!

Array Implementation of List ADT

Arrays have predefined size

```
Student[] roster = new Student[100];
```

⇒ what happens when there are very few?

⇒ What happens when there are too many?

101

copy 100 elements

Growable Array

Let us denote by push(o) insert at end

push is same as adding after last non-empty slot in the array

We will run out of space

1. Create a larger array A'
2. Copy all elements of A to A'
   in order, and assign A to A'
3. put the element o at its right index.

Growable Array

Let us denote by push(o) insert at end

push is same as adding after last non-empty slot in the array

We will run out of space!

1. Create a larger array A'
2. Copy all elements of A to A' in order, and assign A to A'
3. put the element o at its right index.

How large should A' be?

Strategies to Grow

How to allocate the larger A' ?

Strategy 1:

Let the size of A' be c larger than A

```
Student[] A_dash = new Student[A.length + c];
for(int k=0; k< A.length; k++) {
    A_dash[k] = A[k];
}
A = A_dash;
```

} copying

Strategies to Grow

How to allocate the larger A' ?

Strategy 2:

Let the size of A' be twice that of A

```
Student[] A_dash = new Student[A.length * 2];
    for(int k=0; k< A.length; k++) {
        A_dash[k] = A[k];
    }
    A = A_dash;
```

Work Involved to Grow

we start with an array with capacity 1

Double the size each time we run out

Total time for n push operations is $O(n)$

Effectively, each push operation is Constant cost!

We replace the array $k = \log_2 n$ times

Total time $T(n)$ over n push operations is

$$T(n) = n + 1 + 2 + 4 + 8 + \cdots + 2^k$$

$$= n + 2^{k+1} - 1$$

$$= 3n - 1 \Rightarrow \boxed{T(n) = O(n)}$$

Array Implementation of List

⇒ Locating an element at an index i

Constant cost operation O(1)

⇒ Space used by the list

as much as largest it can be

⇒ add / remove operations

can be as bad as requiring full array copy O(n)

What drives the choice of implementation?

1. Efficiency of the algorithm

   ⇒ will the implementation choice help in speeding up the algorithm.

   ⇒ add new methods, improve state / data "structure", storage and access.

2. Resource usage (usually memory).