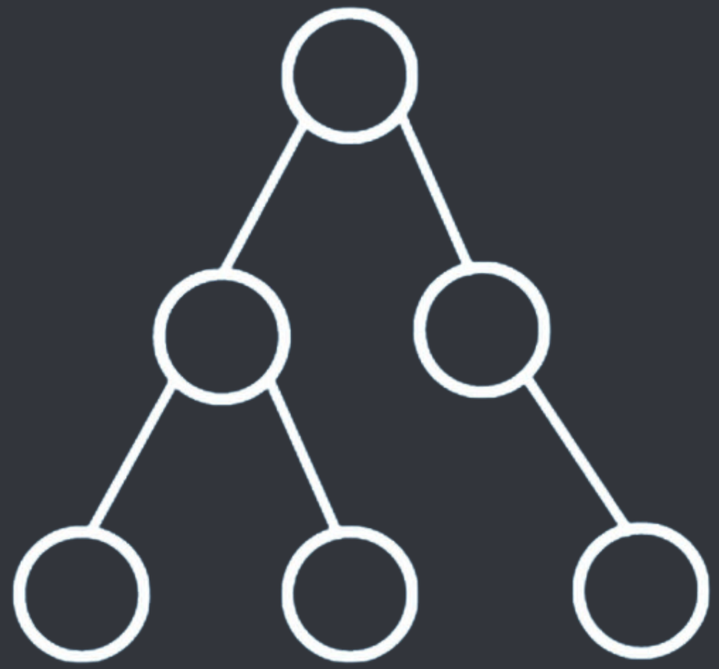
IT Club



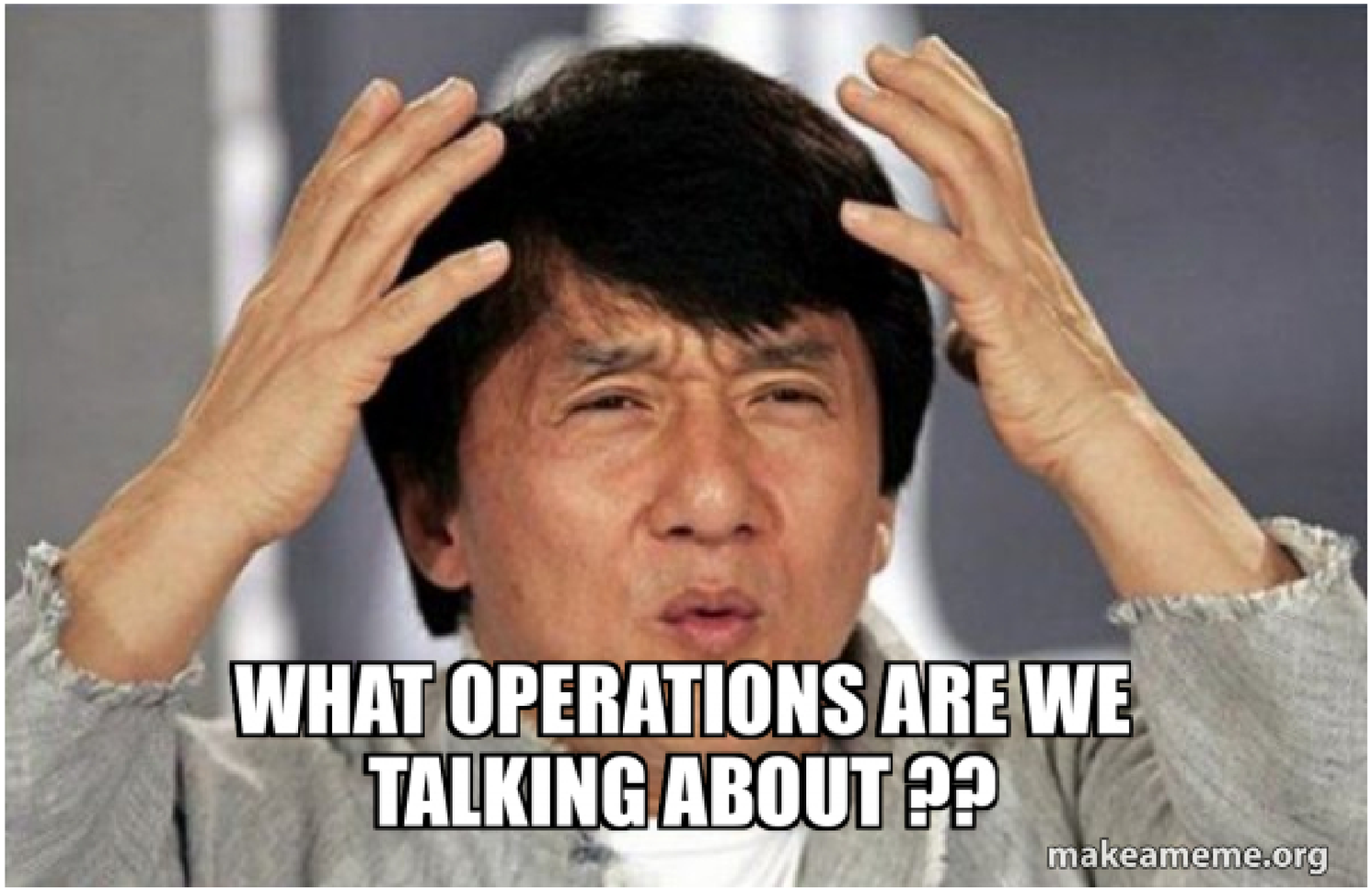
ADVANCED C WORKSHOP

Day 2 and 3
Data Structures and Algorithms

Part 1

What are Data Structure and Algorithms?

A data structure is a way of organizing, storing, and managing data so that operations can be done efficiently.


WHAT OPERATIONS ARE WE TALKING ABOUT ??
makeameme.org

Operations

IT Club

- Insertion
- Deletion
- Traversal
- Searching
- Sorting
- Accessing

Types of Data Structures

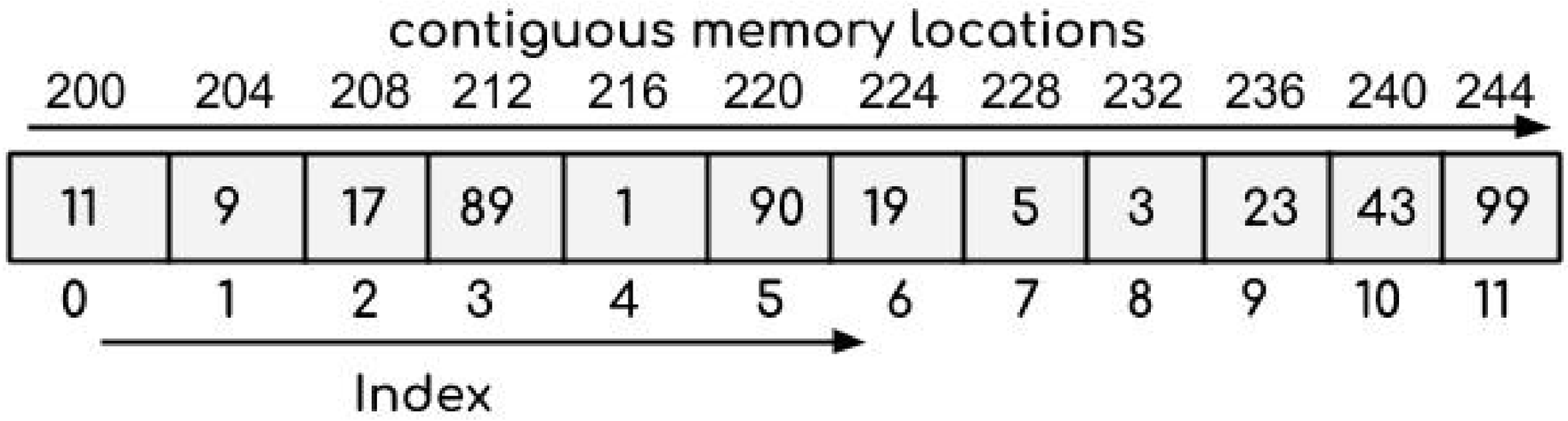
IT Club

1. Linear data structure
2. Non-linear data structure

Linear Data Structure

IT Club

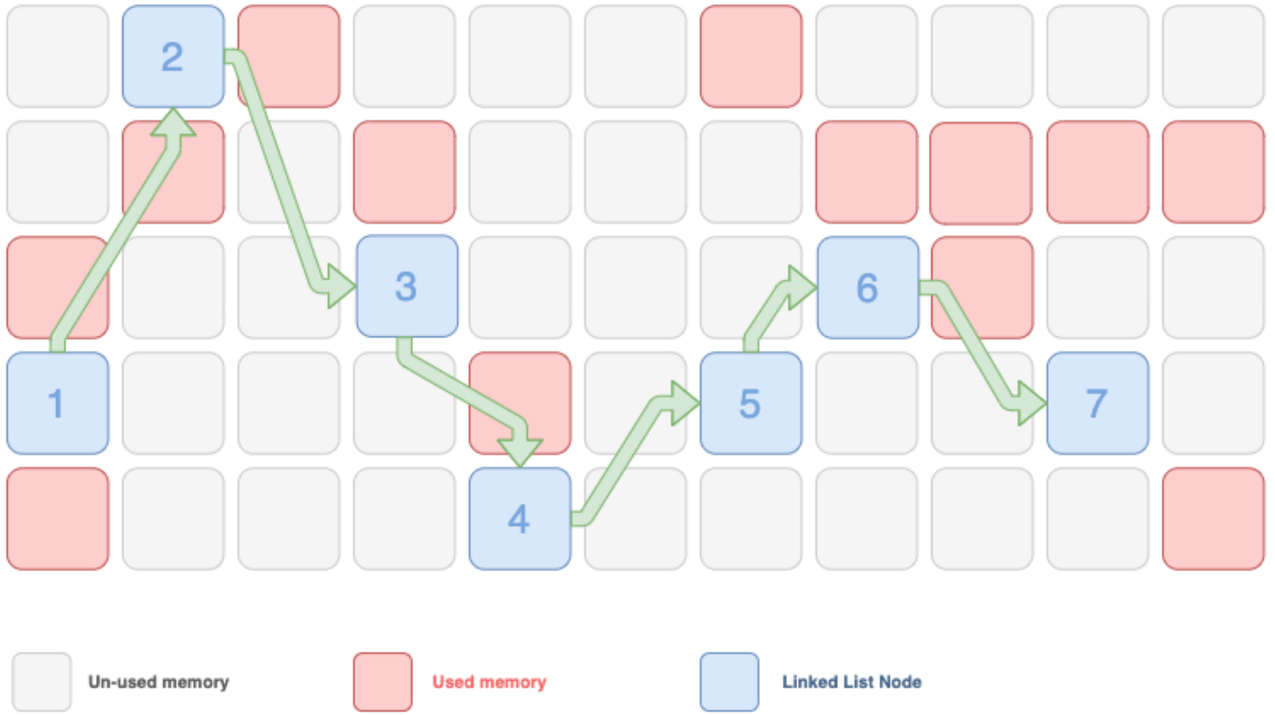
- data elements are *arranged sequentially or linearly* where *each and every element is attached to its previous and next adjacent element*
- Array, linked list, stack, queue
- Elements are *accessed in a sequential order* but it is *not compulsory to store all elements sequentially*.


contiguous memory locations
200
204
208
212
216
220
224
228
232
236
240
244
11
9
17
89
1
90
19
5
3
23
43
99
0
1
2
3
4
5
6
7
8
9
10
11
Index

Array

Linear Data Structure

IT Club



Linked list

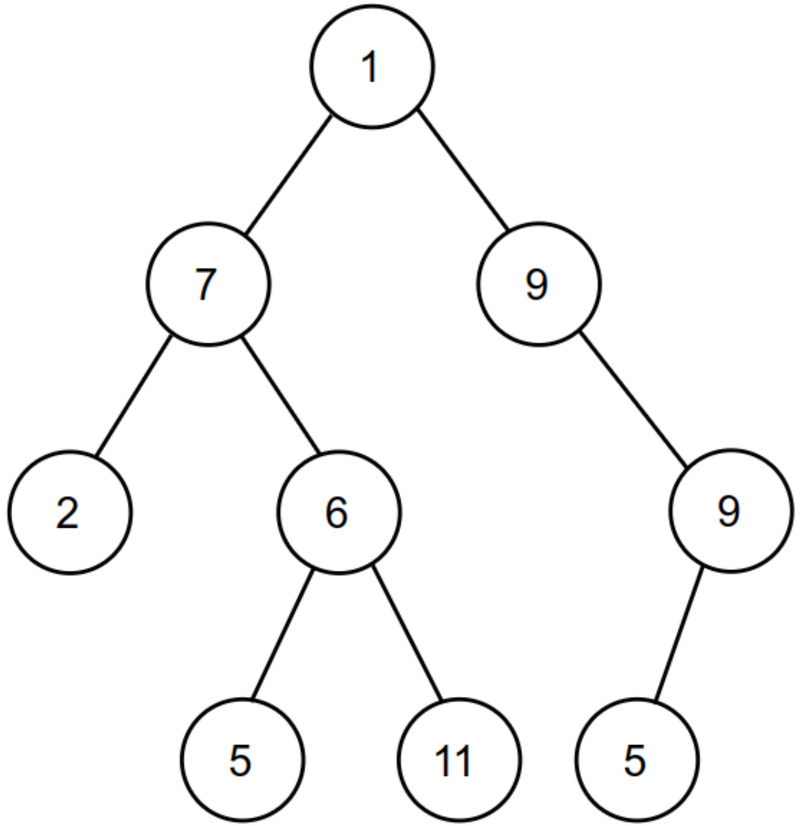
Non-linear Data Structure

IT Club

- data elements are *not* arranged sequentially or linearly
- Elements of this data structure are *stored/accessed in a non-linear order*.
- trees and graphs

Non-linear Data Structure

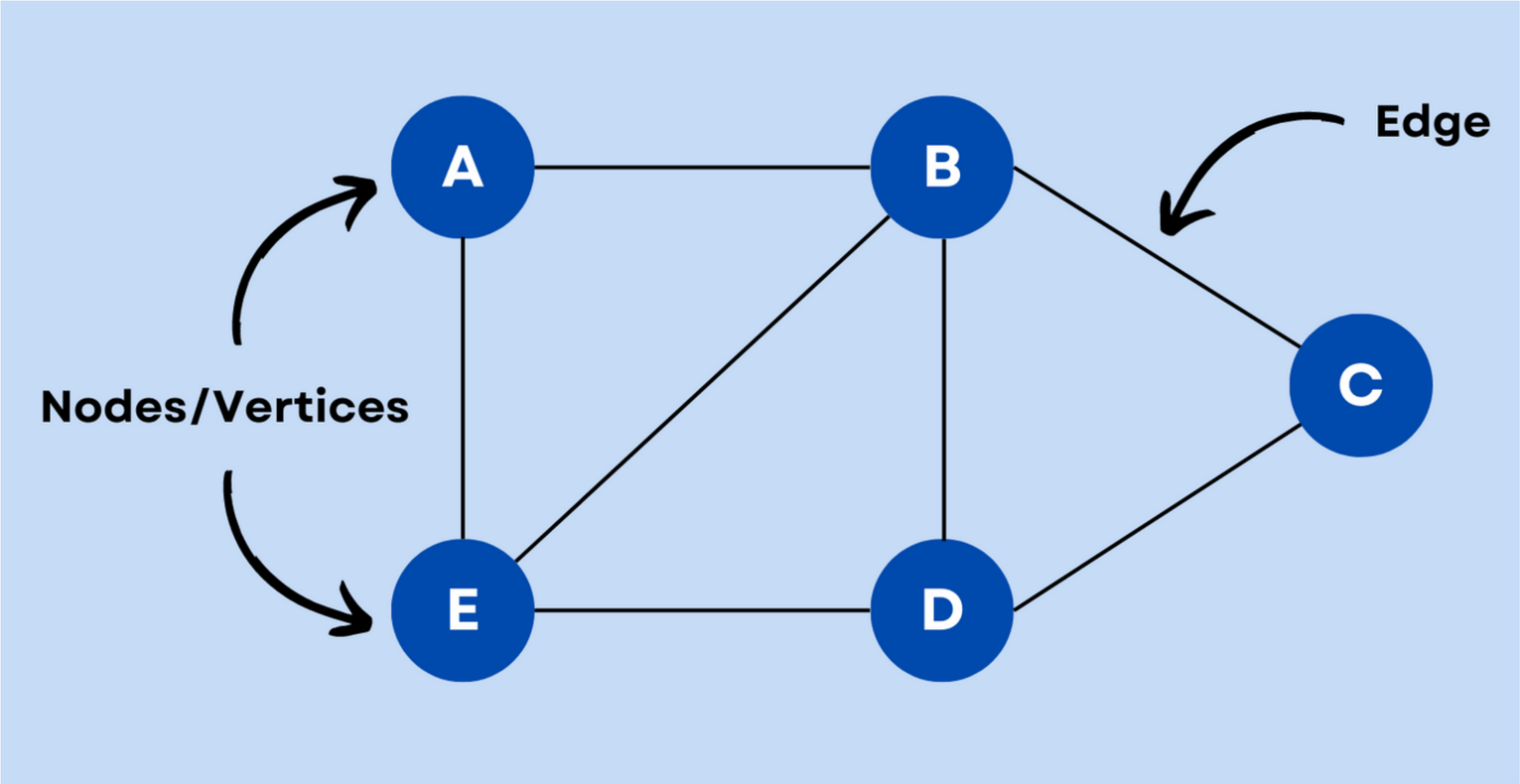
IT Club



Tree

Non-linear Data Structure

IT Club



Graph

Two Pillars of data structures

IT Club

1. How to build one

2. How to use one

An algorithm is the step-by-step unambiguous instructions to solve a given problem.

Three Pillars of programming

IT Club

1. Readability
2. Memory (Space)
3. Speed (Time)

Time and Space Complexities!

The rate at which the running time increases as a function of input is called rate of growth.

Types of asymptotic notations

IT Club

1. Big-O -> aka worst case analysis
2. Big-Omega (Ω) -> best case analysis
3. Big-Theta (Θ) -> average case analysis

What is worst case?

IT Club

- Defines the input for which the algorithm takes a long time (slowest time to complete).

- Input is the one for which the algorithm runs the slowest.

4 simple rules to determine Big O

IT Club

1. Find the worst case
2. Use different terms for multiple inputs

   (arguments of the function)
3. Remove the constants
4. Drop non dominants (keep only the worst case)

Example

IT Club

```
// Loop through the array
for (int i = 0; i < n; i++) {
    if (nums[i] == 5) {
        // Print a message when 5 is found
        printf("Number 5 is found at index %d.\n", i);
        break; // Exit the loop once 5 is found
    }
}
```

Example

IT Club

```
function printArrayPairs(arrayA, arrayB)
    for elementA in arrayA
        for elementB in arrayB
            print(elementA, elementB)
        end for
    end for
end function
```

Example

IT Club

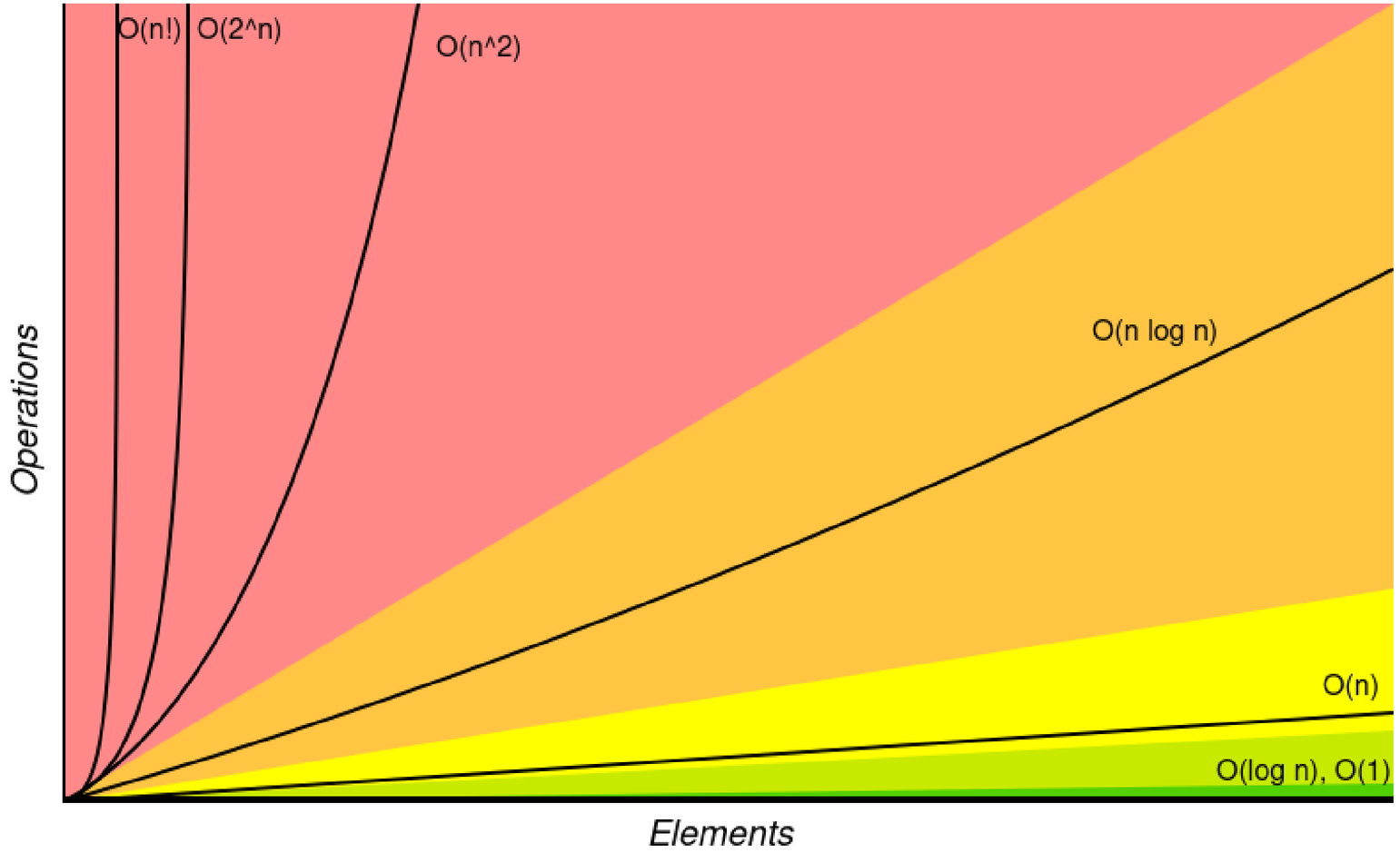
```
// First loop with time complexity O(3n^2)
for (int i = 0; i < 3 * n; i++) {
  for (int j = 0; j < n; j++) {
      // Some constant-time operation
  }
}

// Second loop with time complexity O(2n)
for (int k = 0; k < 2 * n; k++) {
  // Some constant-time operation
}

// Third loop with time complexity O(5)
for (int l = 0; l < 5; l++) {
  // Some constant-time operation
}

// Total time complexity:
// O(3n^2) + O(2n) + O(5) = O(3n^2 + 2n + 5) = O(n^2)
```

Some common time complexities

IT Club

Time Complexity	Name	Example
O(1)	Constant	Accessing an element in an array by index
O(n)	Linear	Iterating through elements in an array
O(log n)	Logarithmic	Binary search in a sorted array
O(n log n)	Linearithmic	Merge sort or quicksort
O(n^2)	Quadratic	Iterating through n x n 2D array
O(2^n)	Exponential	The Tower of Hanoi problem

Comparison



What is the time complexity
for the following?

```
for (int i = 1; i <= n; ++i)
{
    i *= 2;
}
```

For n = 8, it takes 3 iterations

Loop	***i***	i (for next iteration)
1	1	2 * 1 + 1 = 3
2	3	2 * 3 + 1 = 7
3	7	2 * 7 + 1 = 15 (> 8)

For n = 16, it takes 4 iterations

Loop	*i*	i (for next iteration)
1	1	2 * 1 + 1 = 3
2	3	2 * 3 + 1 = 7
3	7	2 * 7 + 1 = 15
4	15	2 * 15 + 1 = 31 (> 16)

Exercise Solution

IT Club

Let for any variable n >= 1, $n = 2^k$
where k is the number of operations taken (iterations)

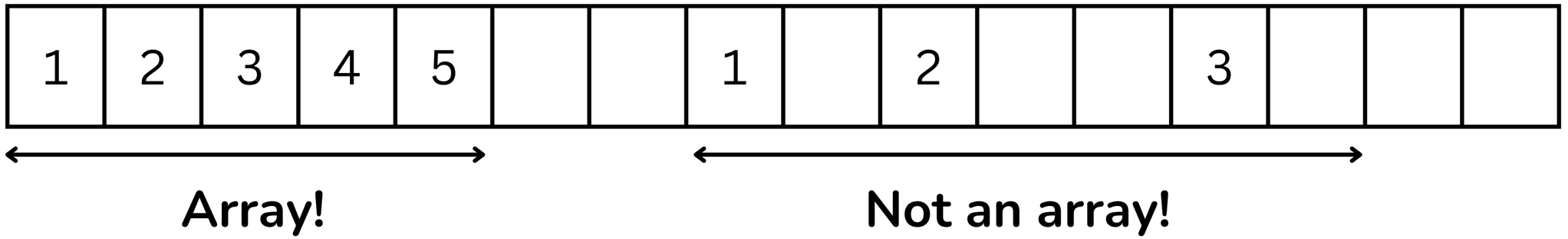
Taking log on both sides,
$\log(n) = \log(2^k)$
$\log(n) = k \log 2 = k \rightarrow k = \log_2(n)$

$\therefore$ Time complexity = O(log n)

Some useful data structures!

Arrays

Arrays are basically a bunch of **similar data** stored together that are **contiguous in memory.**



Notable features:

- O(1) element access
- Locality of reference
- O(n) insertion and deletion
- O(n) search in an unsorted array

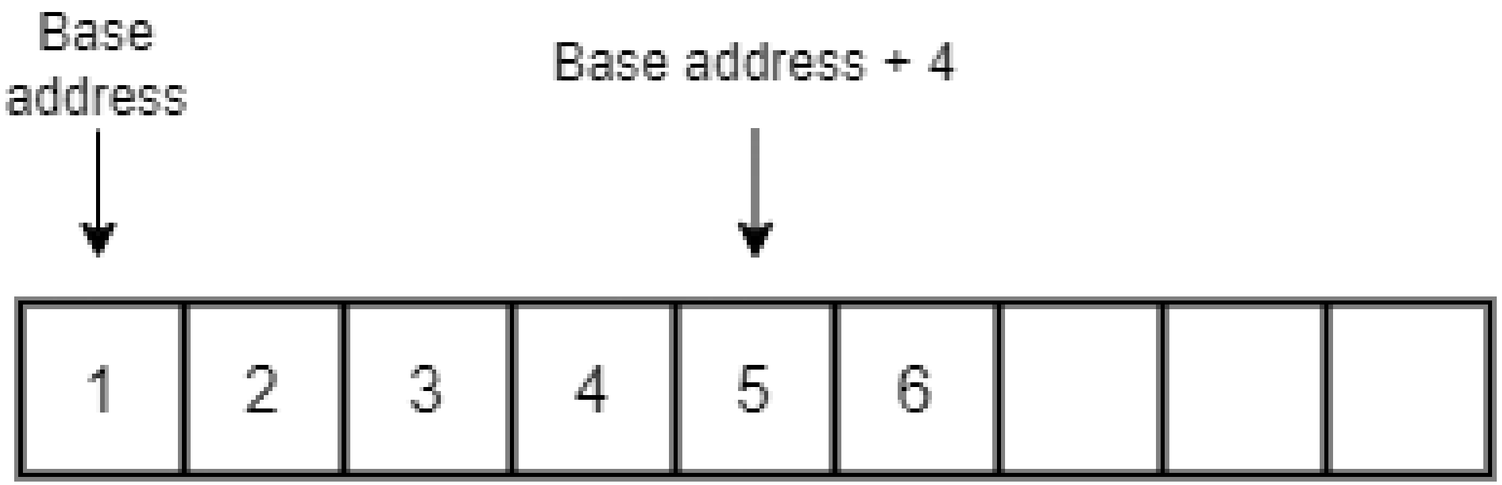
How do you access the i-th element of an array?

array[i] right?

but what actually goes on behind this?

Arrays: Access

- Elements are accessed using simple pointer arithmetic regardless of number of elements



array[i] = *(base address + i)

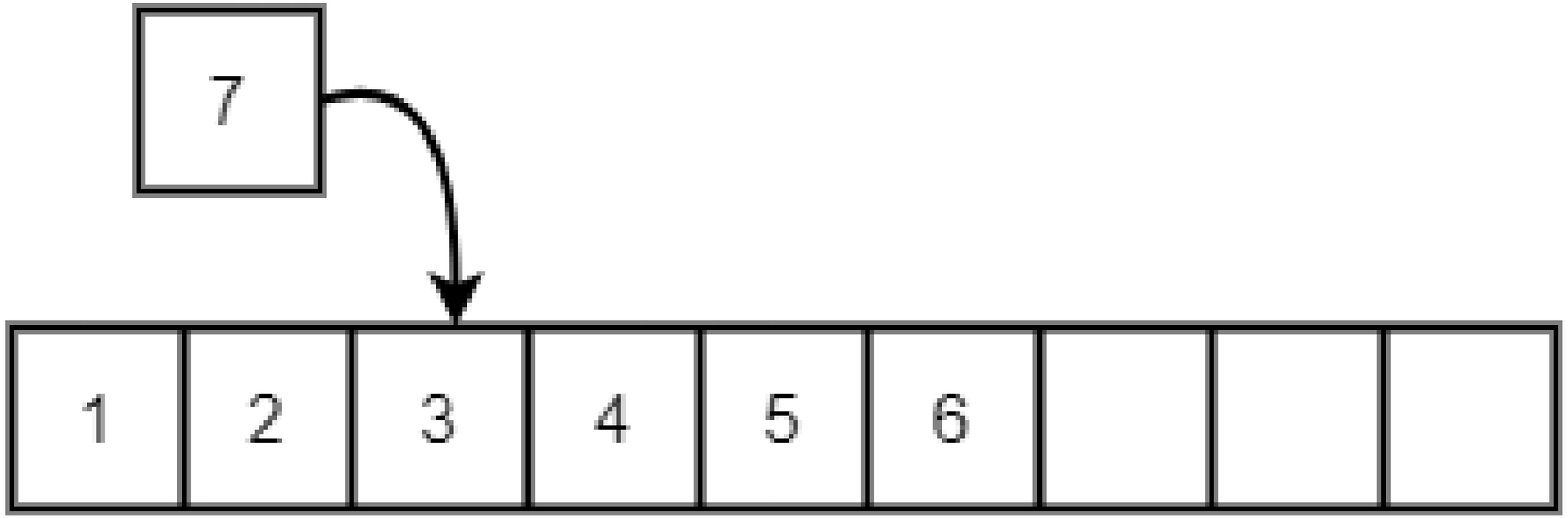
So,
array[i] and **i[array]**
boil down to the same expression
***(array + i)**

meaning..

This is completely valid!

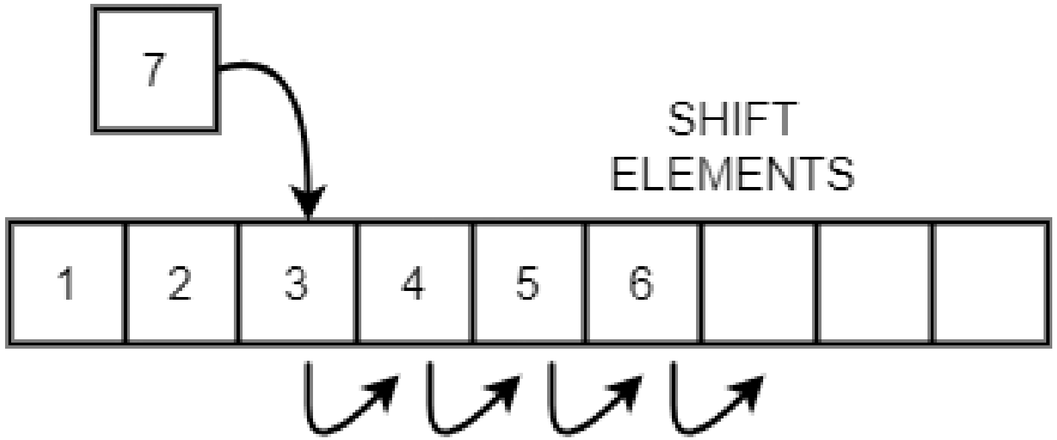
```
int arr[] = {1,2,3,4,5};
printf("%d", 0[arr]);
```

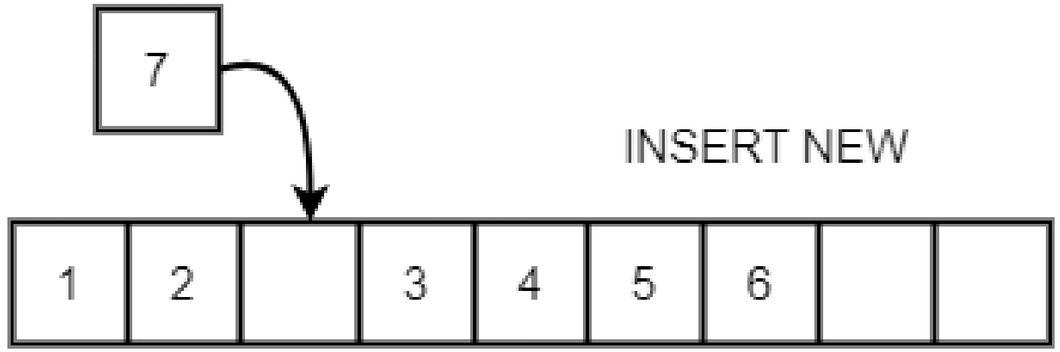
How would you insert a new element in an array?

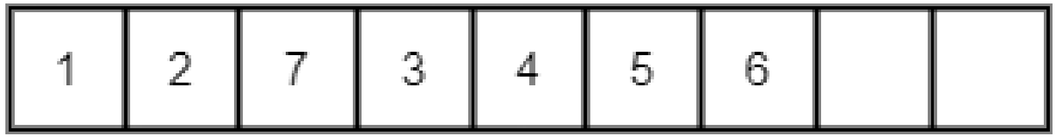


Arrays: Insertion

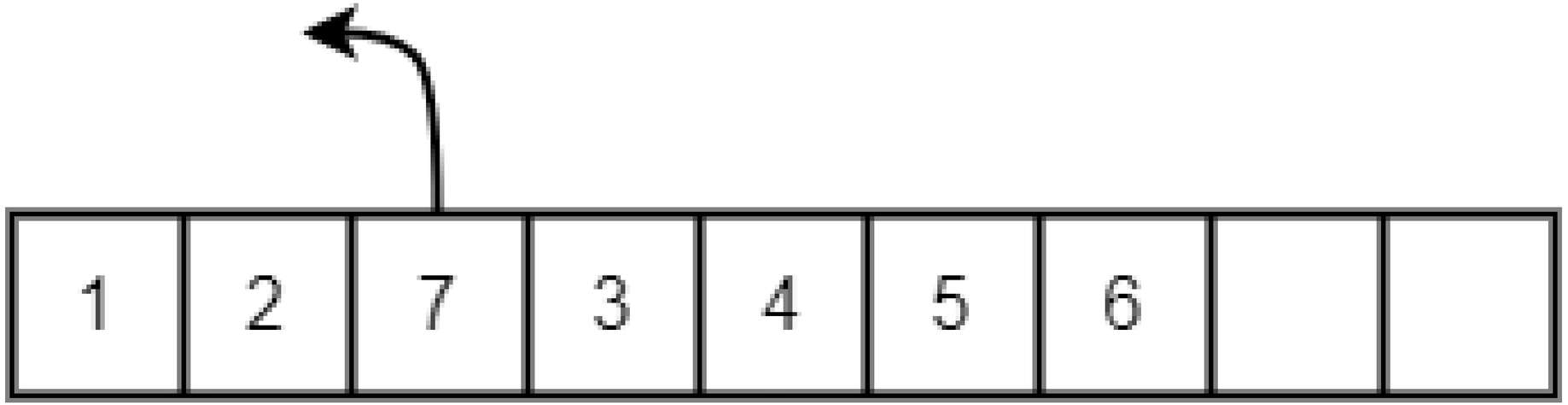
- Requires shifting of elements





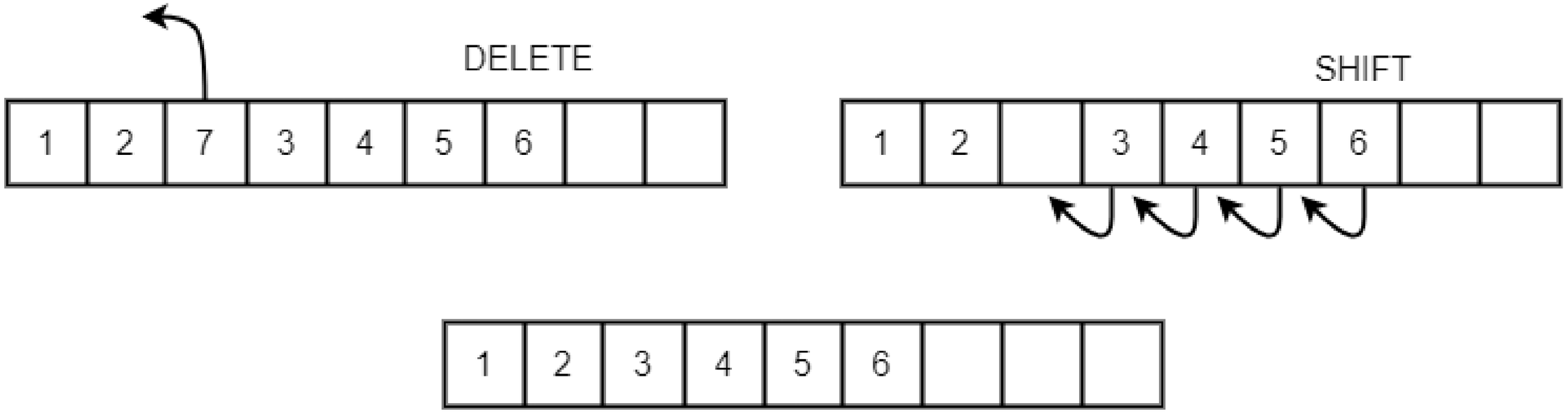


How would you delete an element from an array?



Arrays: Deletion

- Requires shifting of elements



Arrays

Pros:

- simple, fast, cache friendly

Cons:

- not very suitable for a lot of insertions and deletions

Linked lists

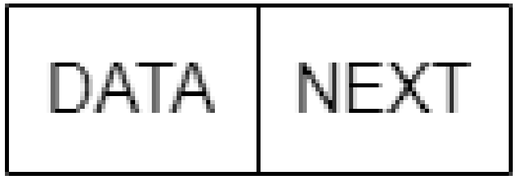
Linked lists

IT Club

Linked lists are basically a bunch of **similar data**
stored as individual chunks called **nodes**
which are **linked** to adjacent nodes

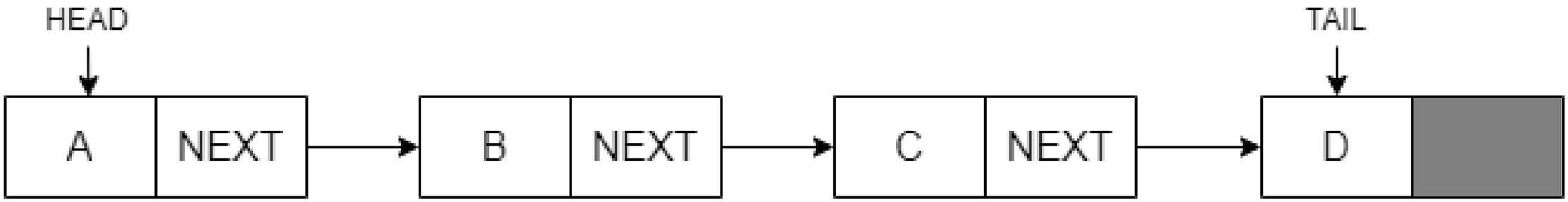
A node consists of..

- data
- links (pointers to the next/previous node in the list)




```c
struct Node{
    int data;
    struct Node *next;
};
```

Linked lists



Linked list is basically a bunch of **linked nodes** that represent a list

Notable features:

- **HEAD:** Points to **start element** of list
- **TAIL:** Points to **last element** of list
- O(n) element access/list traversal
- O(1) insertion and deletion

Linked lists: Traversal

IT Club

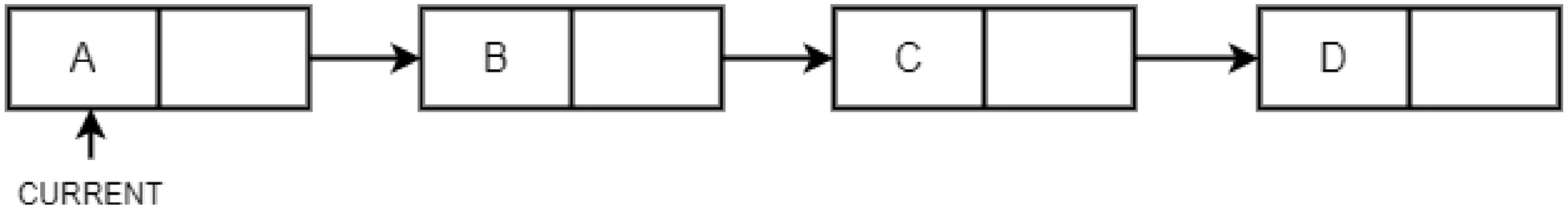
- **Access** requires moving through the list **one node at a time**
- To access the i-th element, travel to next node i times

list[i] = node after traversing the list i times

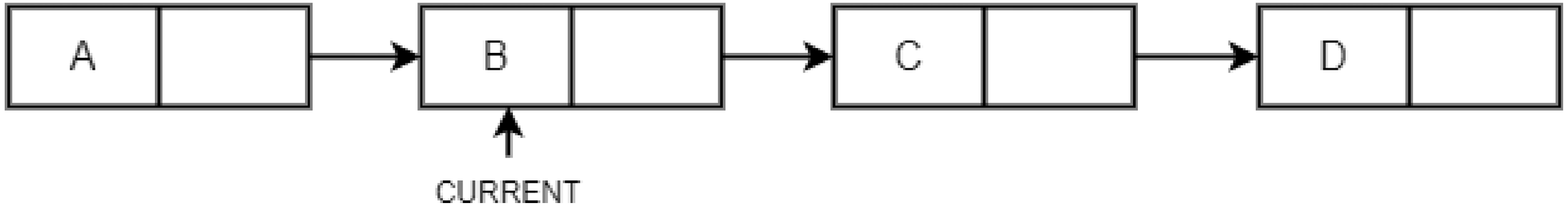
Linked lists

IT Club

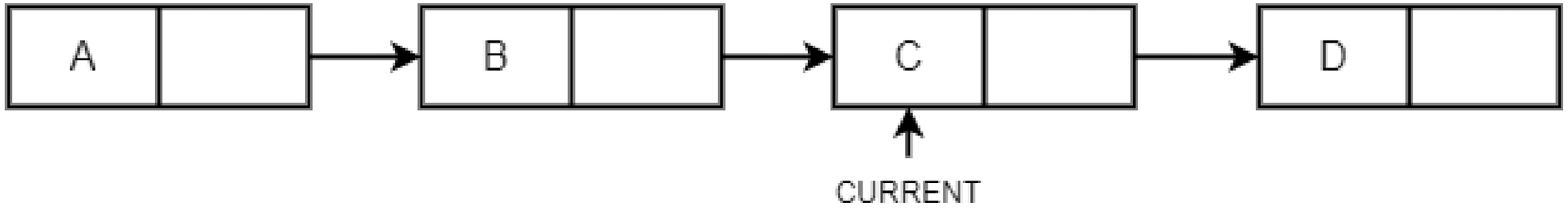
list[0]



list[1]



list[2]

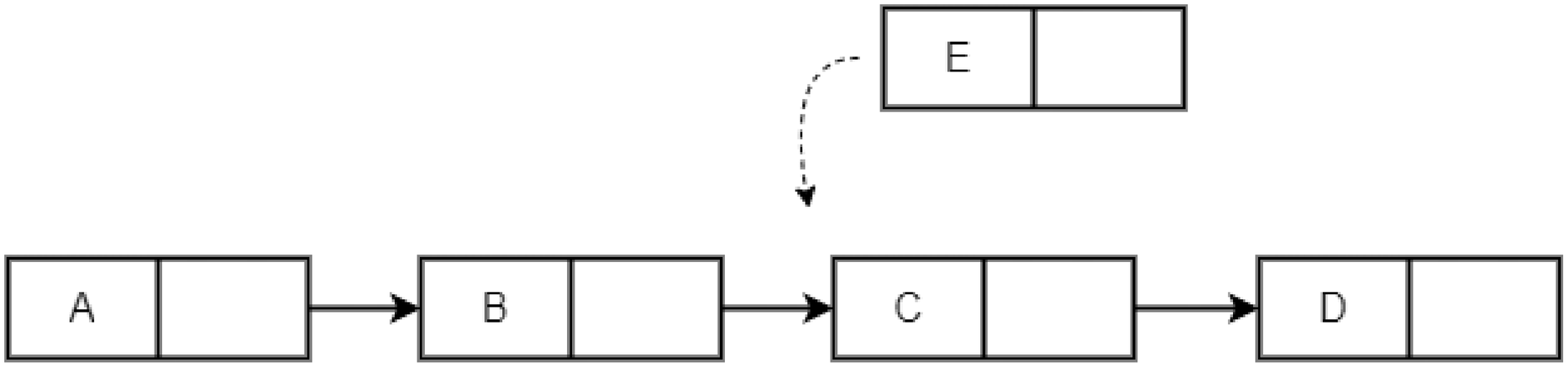


Linked lists

IT Club

```
// loop until current becomes NULL, ie, end of the list
while (current){
    printf("%d, ", current->data);

    // go to the next node
    current = current->next;
}
```

Insertion in a linked list?



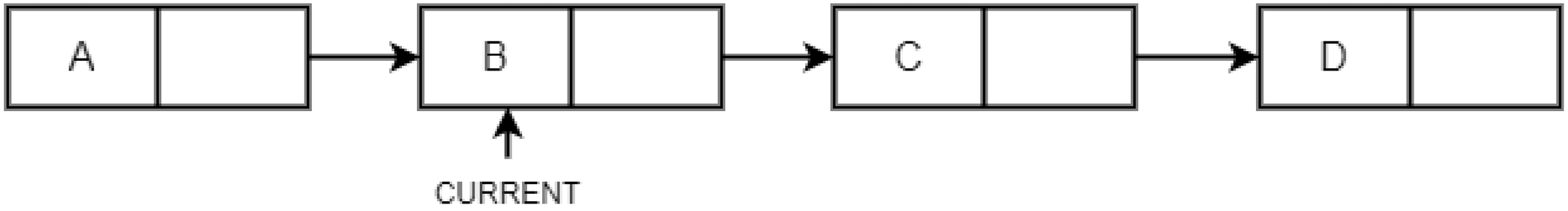
Insertion in a linked list?

- **Traverse** to required position
- **Update new node to point to next node in the list**
- **Update current node to point to new node**

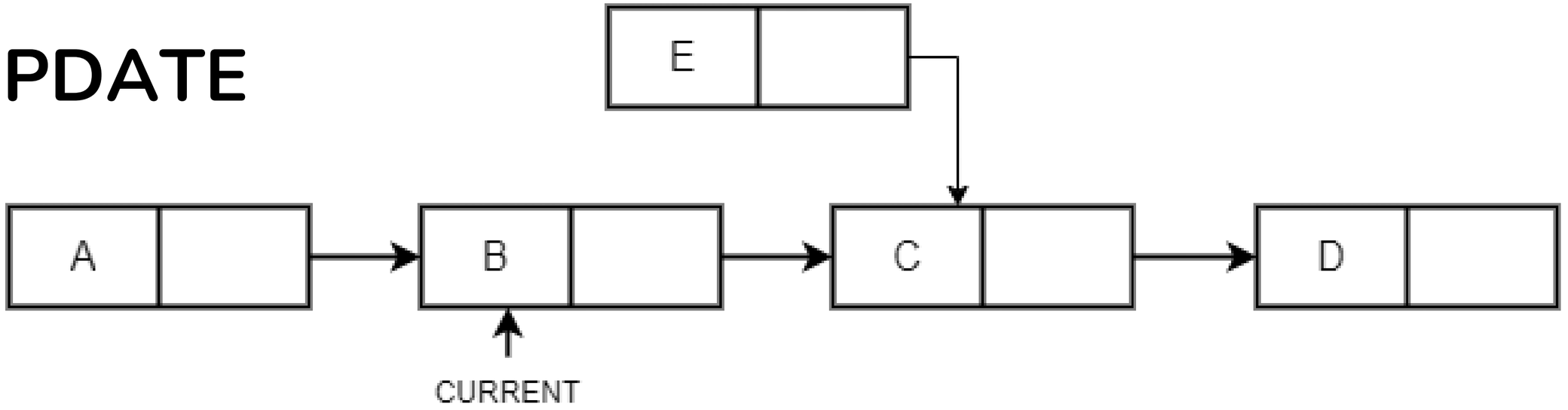
Linked lists: Insertion

IT Club

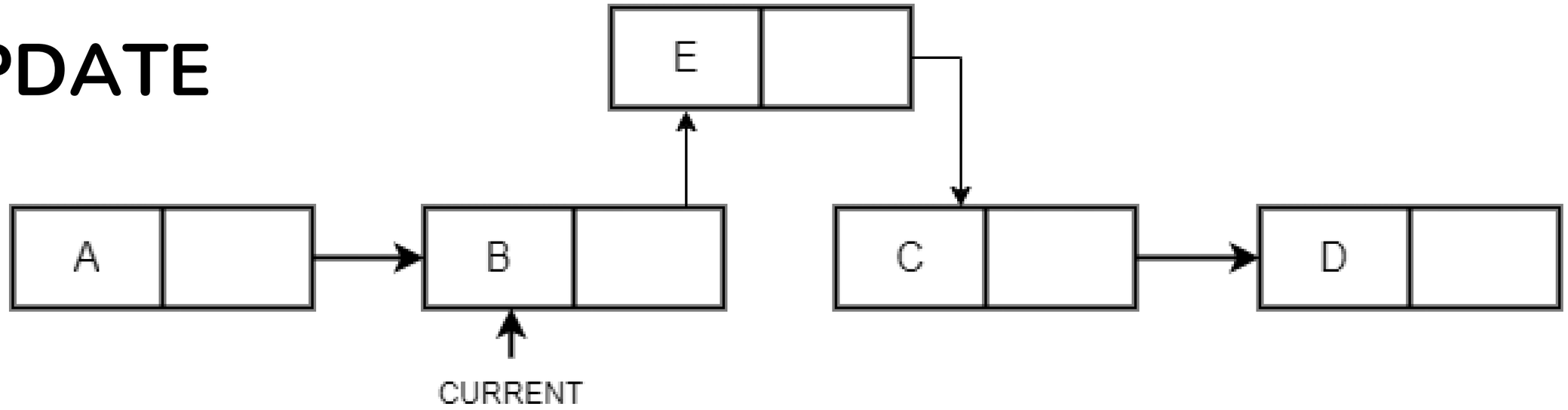
TRAVERSE

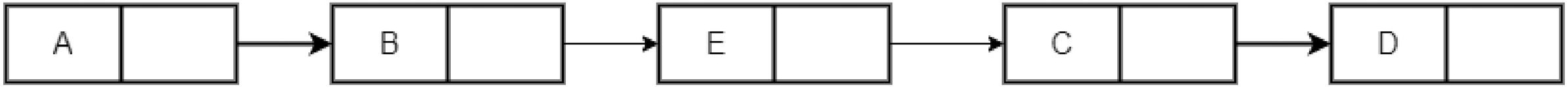


UPDATE

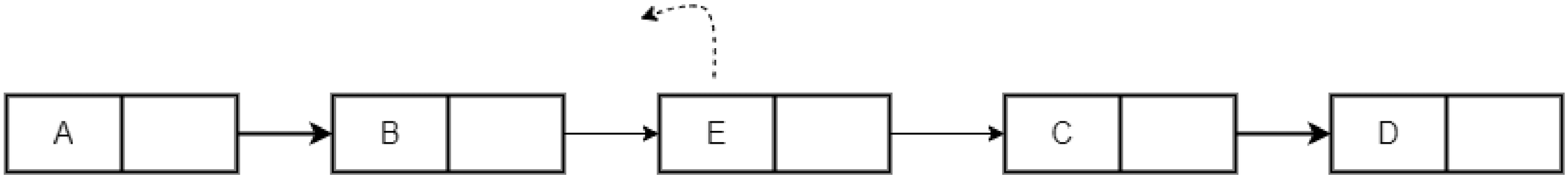


UPDATE





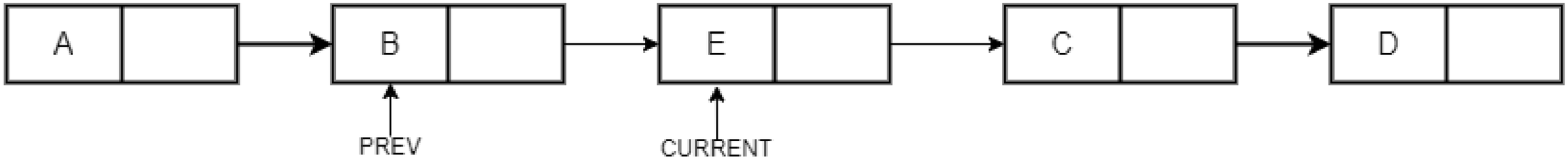
Deleting a node from a linked list?



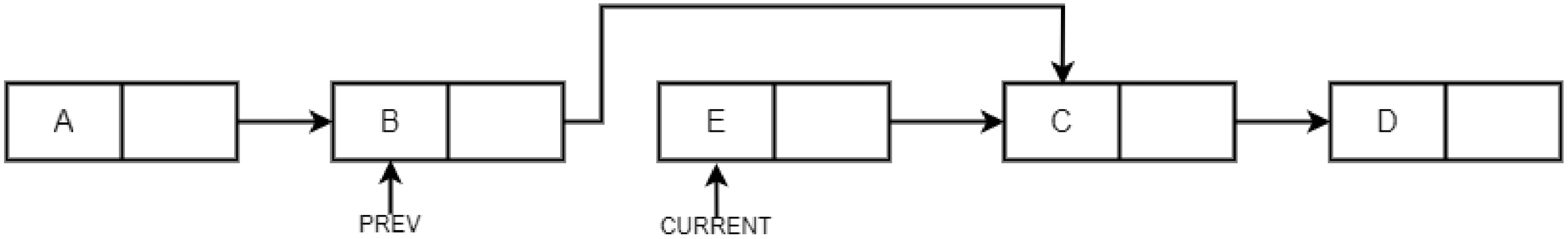
Deleting a node from a linked list?

- **Traverse** to required position
- **Update prev node** to **point** to **next node** in the list
- **Delete** node

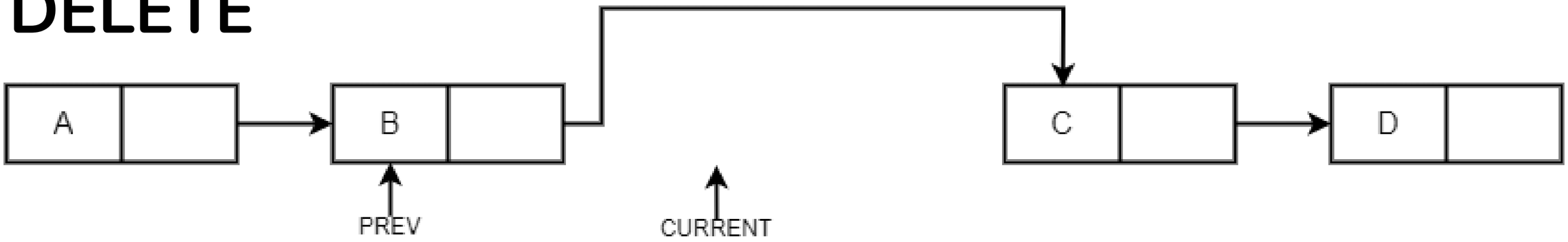
TRAVERSE



UPDATE



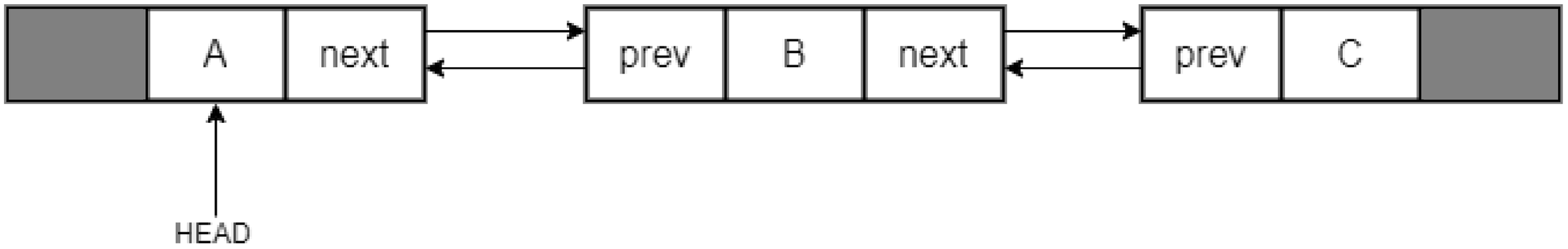
DELETE



Doubly linked lists

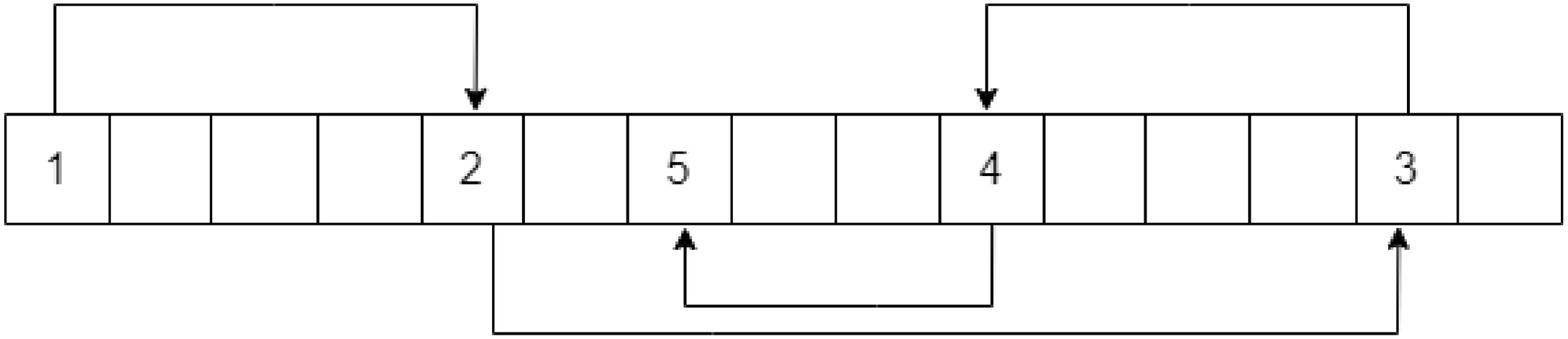
- node also contains a **pointer** to previous node
- list can be traversed both **forwards** and **backwards**

Linked lists



Why linked lists?

- **Flexibility for dynamic allocations (nodes can be anywhere in memory)**



- **Insertions and deletions don’t require shifting or reallocations**
  - only selected nodes need to be updated
  - can be used efficiently with free list allocators

Cons:

- cache misses
- chances of losing the pointer to a node, making it inaccessible

Stack

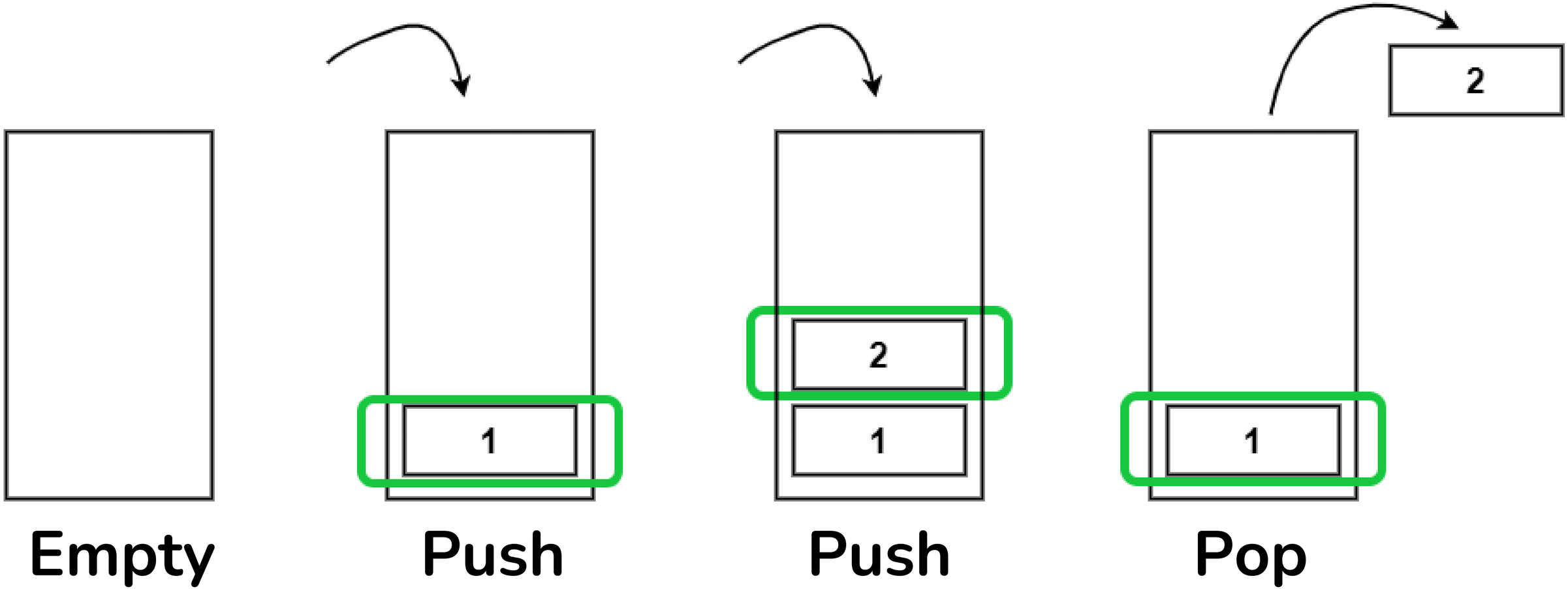
Stacks are a data structure that store and manipulate data in the **Last In First Out (LIFO) order.**

Basically, imagine a stack of books/plates

Features of stack:

- **TOP:** Keeps track of the top of the stack
- **PUSH(val):** Add *val* on top of the stack
- **POP():** Removes the top element

- O(1) insert (**PUSH**) and delete (**POP**)


1
2
1
2
1
Empty
Push
Push
Pop

Both **PUSH** and **POP** need only two operations

- insert at the top/remove from the top
- update the top of stack

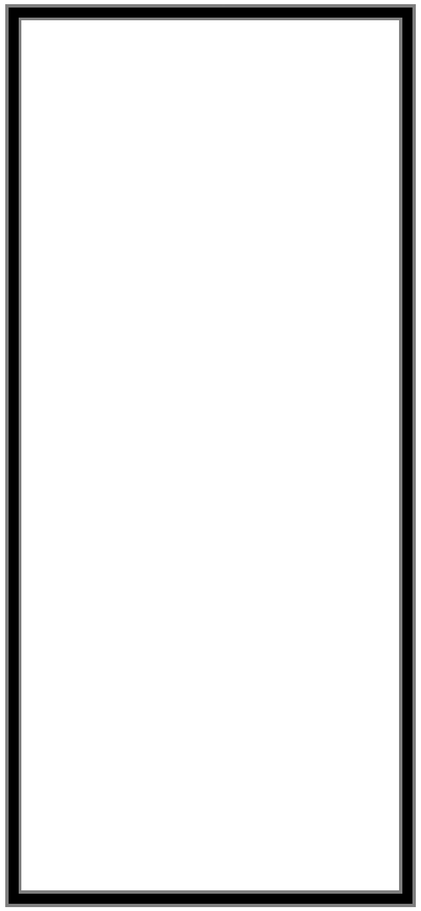
Hence, both operations are **O(1)**

Uses:

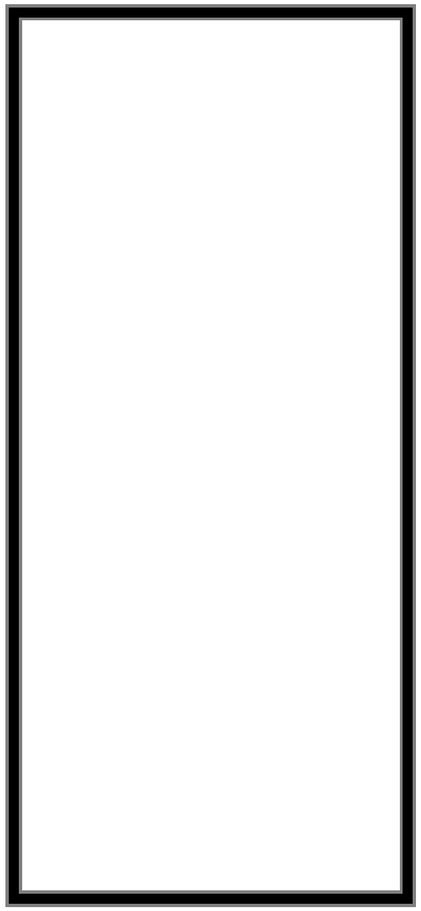
- **Undo-Redo!**
- **DMs list on social media**
- **Call stacks**

and many more!

Stack: Undo-Redo

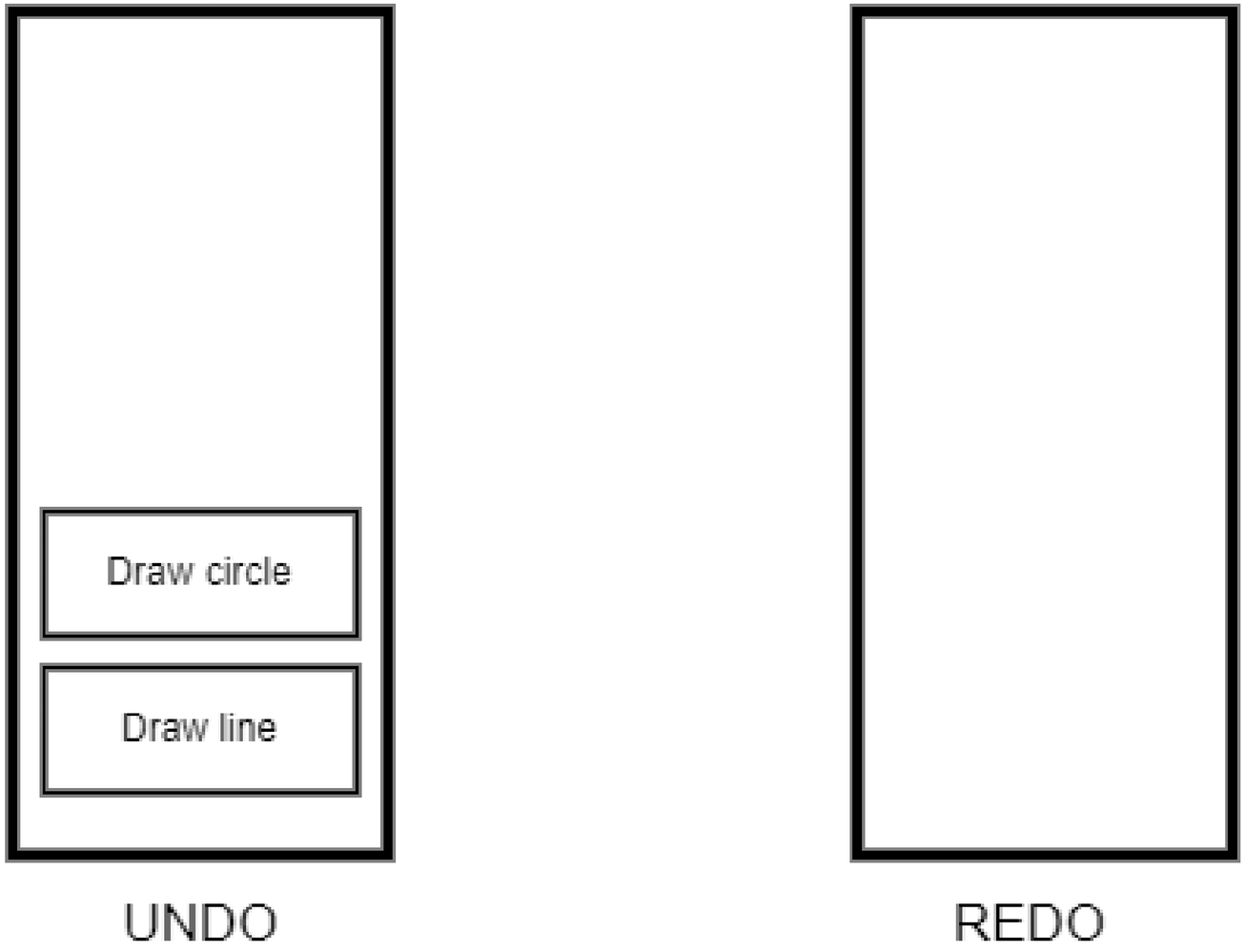


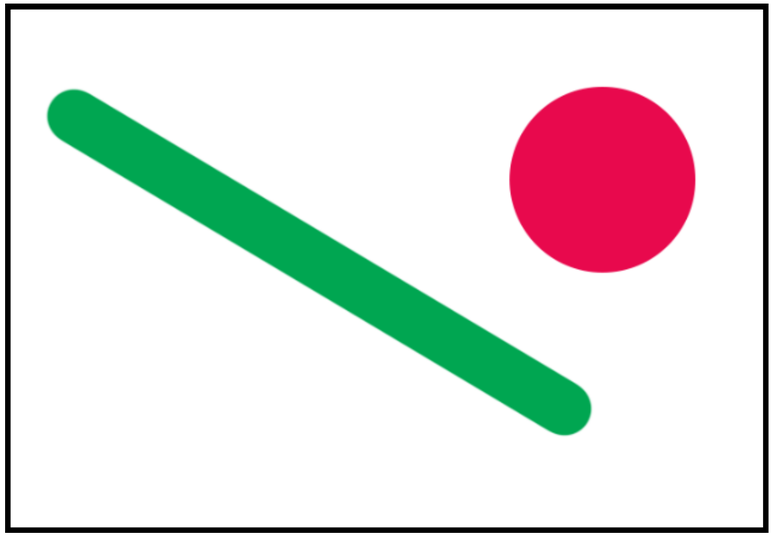
UNDO



REDO

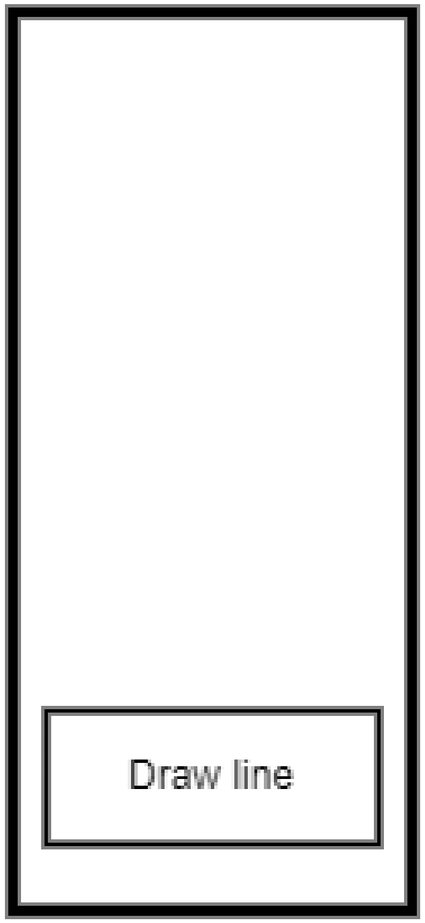
Stack: Undo-Redo



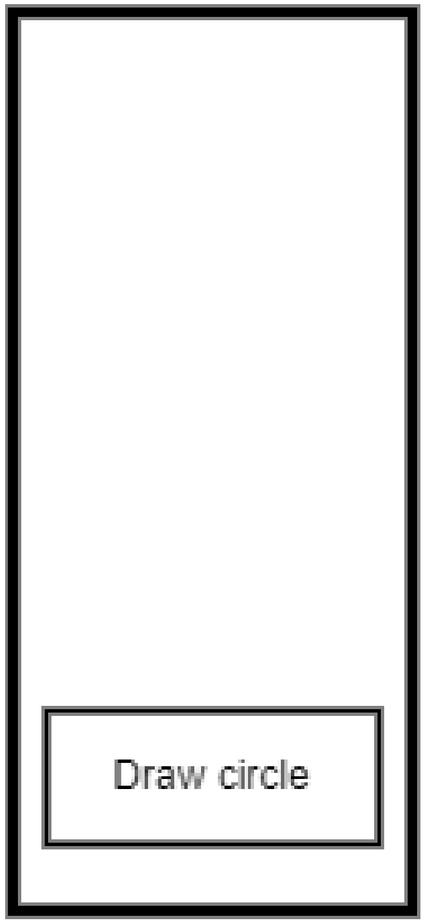


Push all actions into UNDO stack

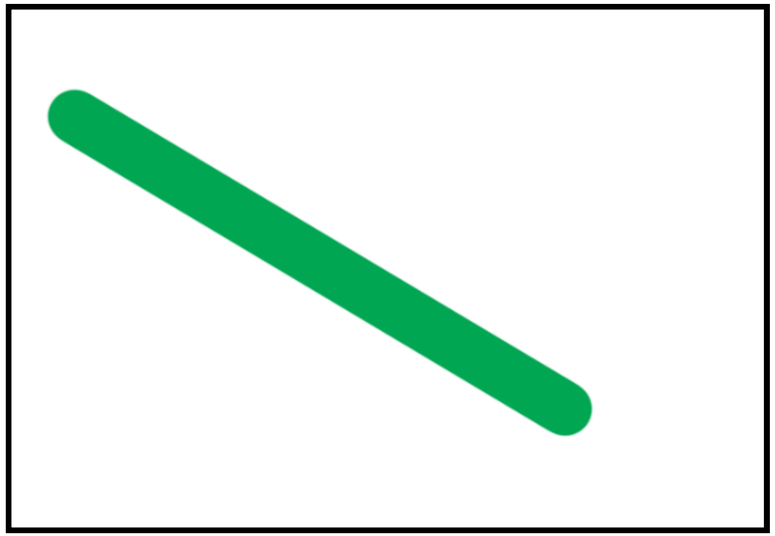
Stack: Undo-Redo



UNDO

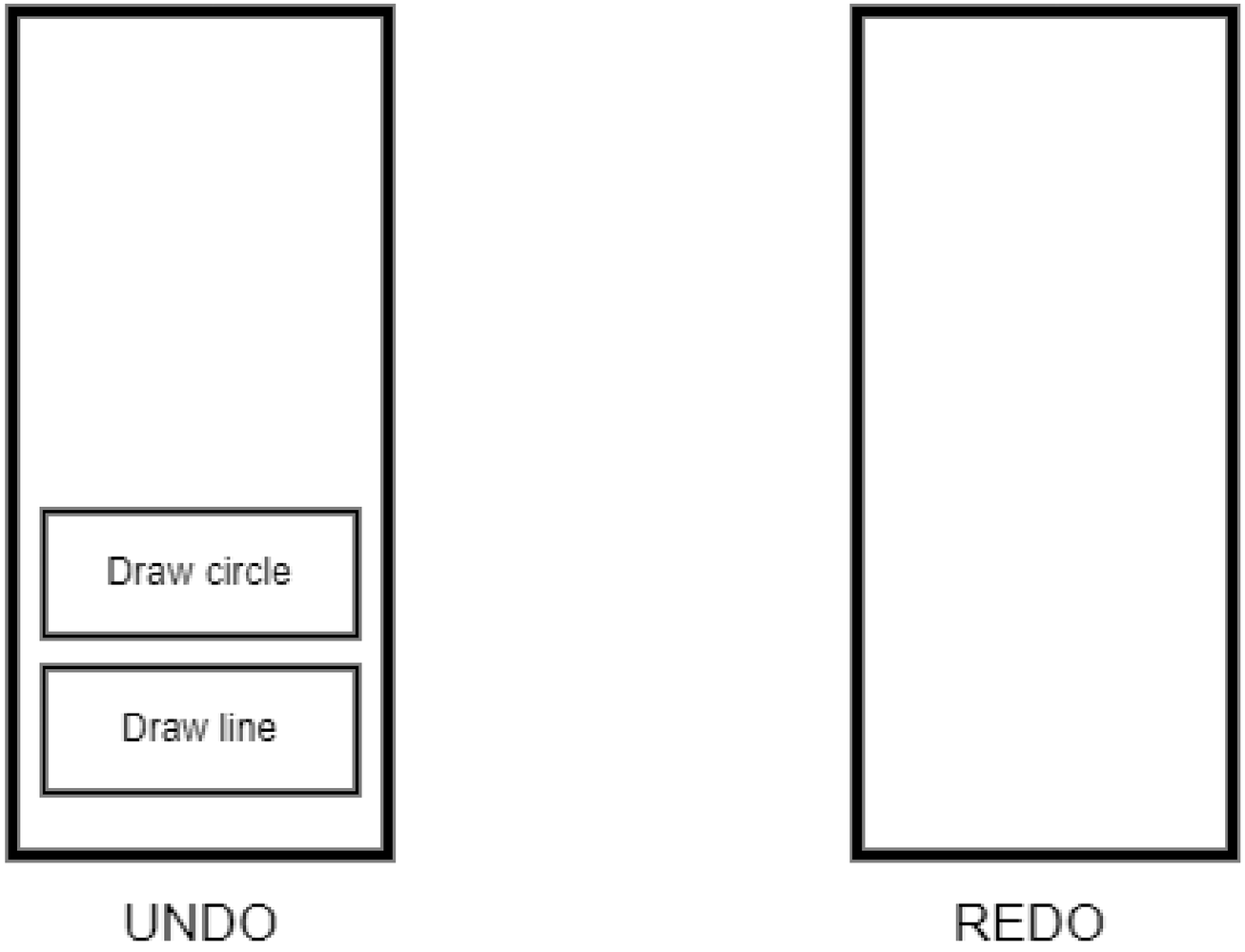


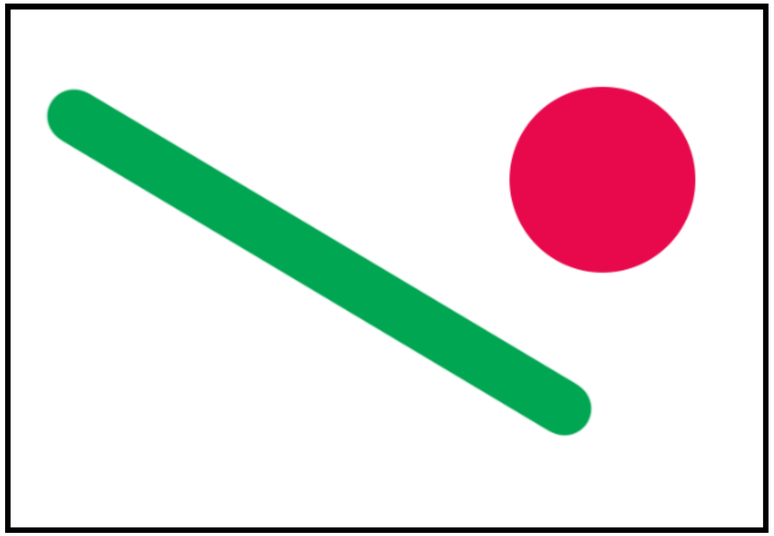
REDO



On undo, pop from UNDO and push to REDO

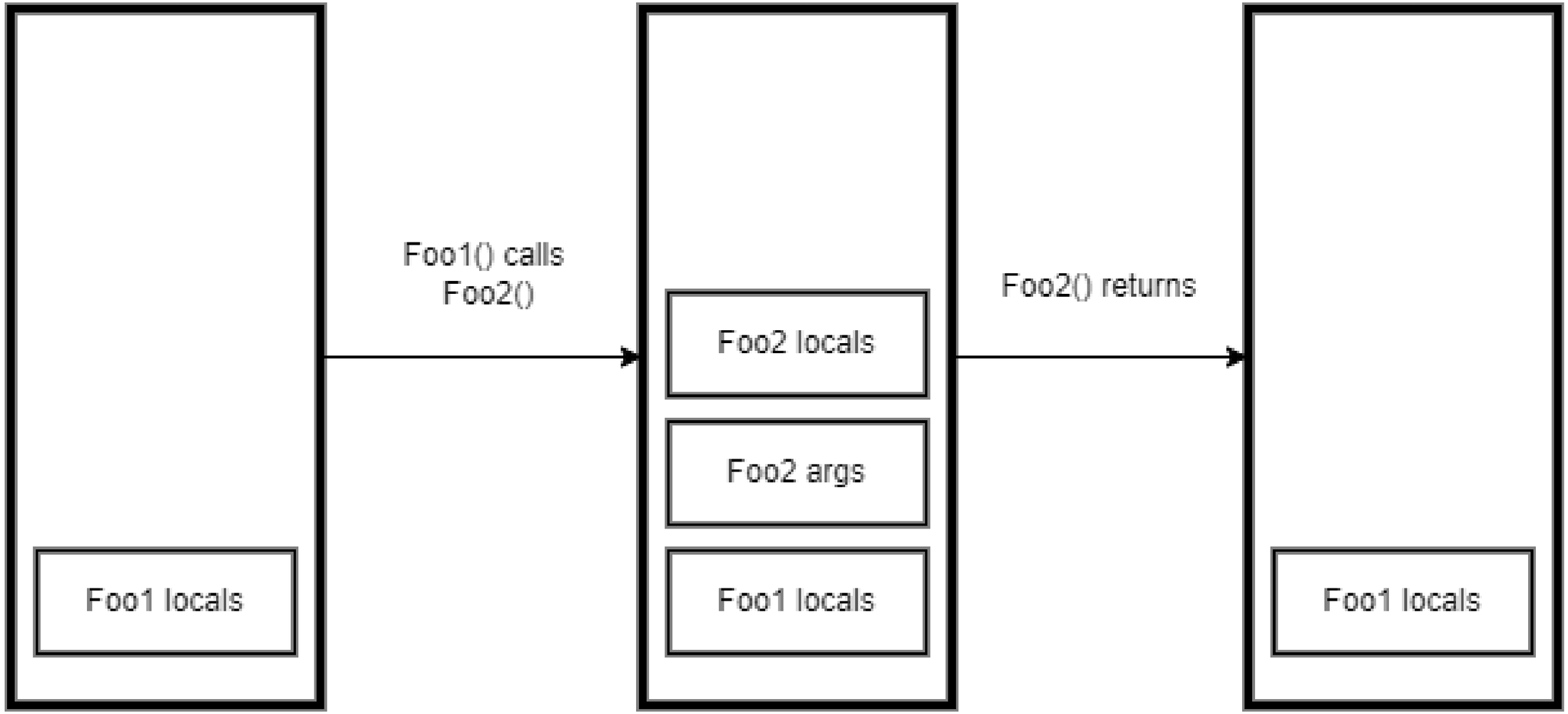
Stack: Undo-Redo





On redo, pop from REDO and push to UNDO

Stack: Call stack



Queues

Queues are a data structure that store and manipulate data in the **First In First Out (FIFO) order.**

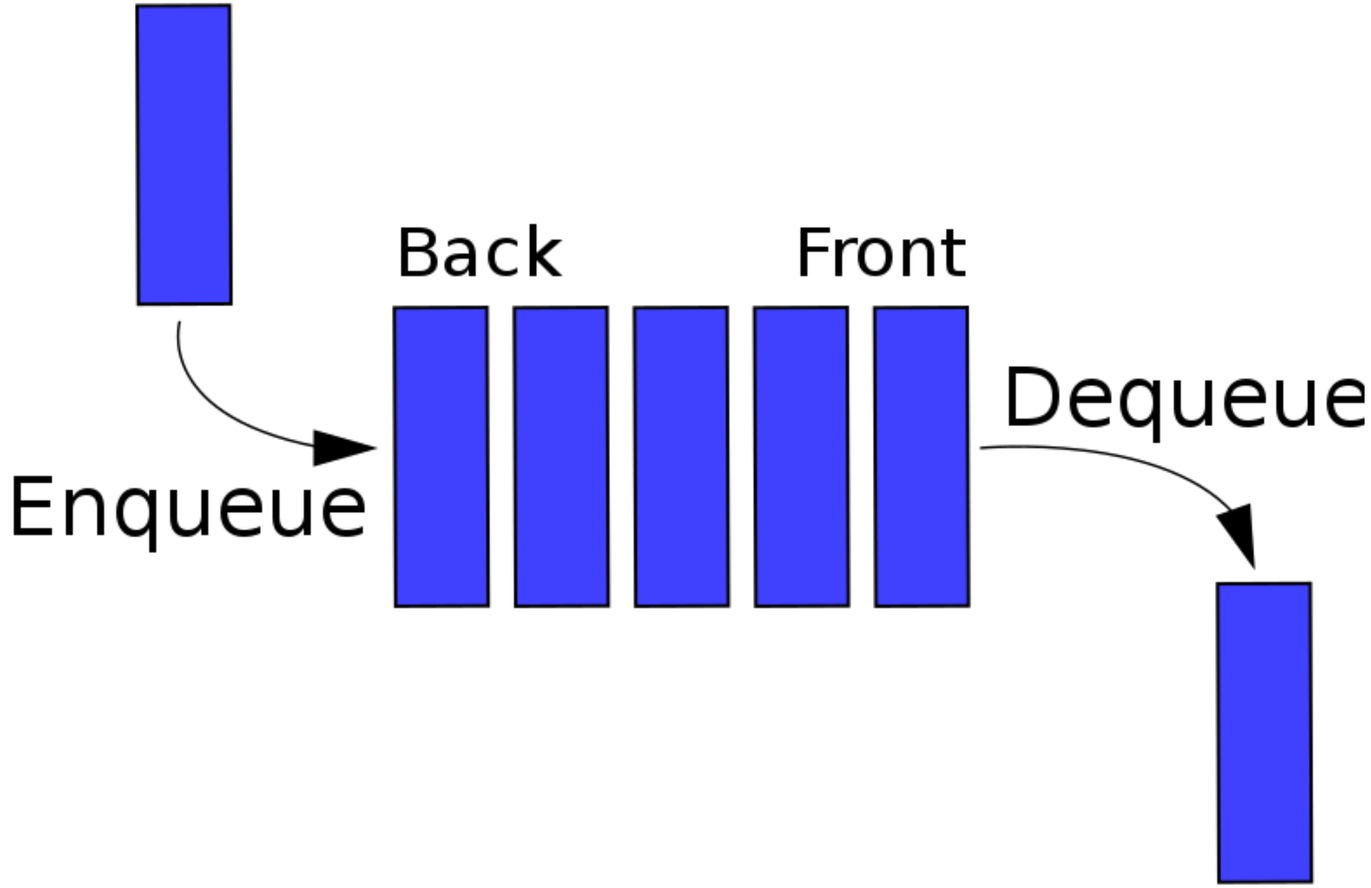
Basically, imagine a queue of people (no nepotism)

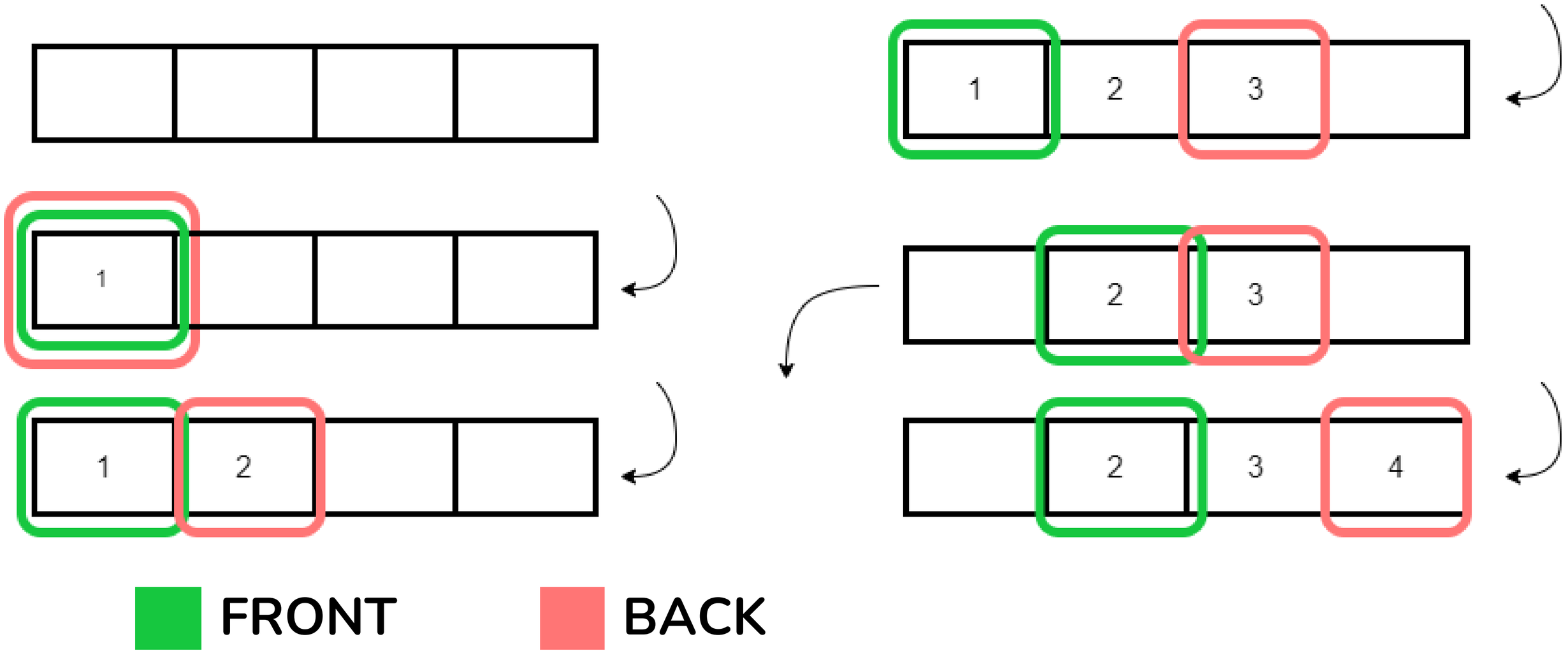
Features of queue:

- **FRONT:** Keeps track of the first added value
- **BACK:** Keeps track of last added value
- **ENQUEUE(val):** Add *val* at the back of queue
- **DEQUEUE():** Removes the front element

- O(1) insert (**ENQUEUE**) and delete (**DEQUEUE**)

Queue




1
1
2
1
2
3
2
3
2
3
4
FRONT
BACK

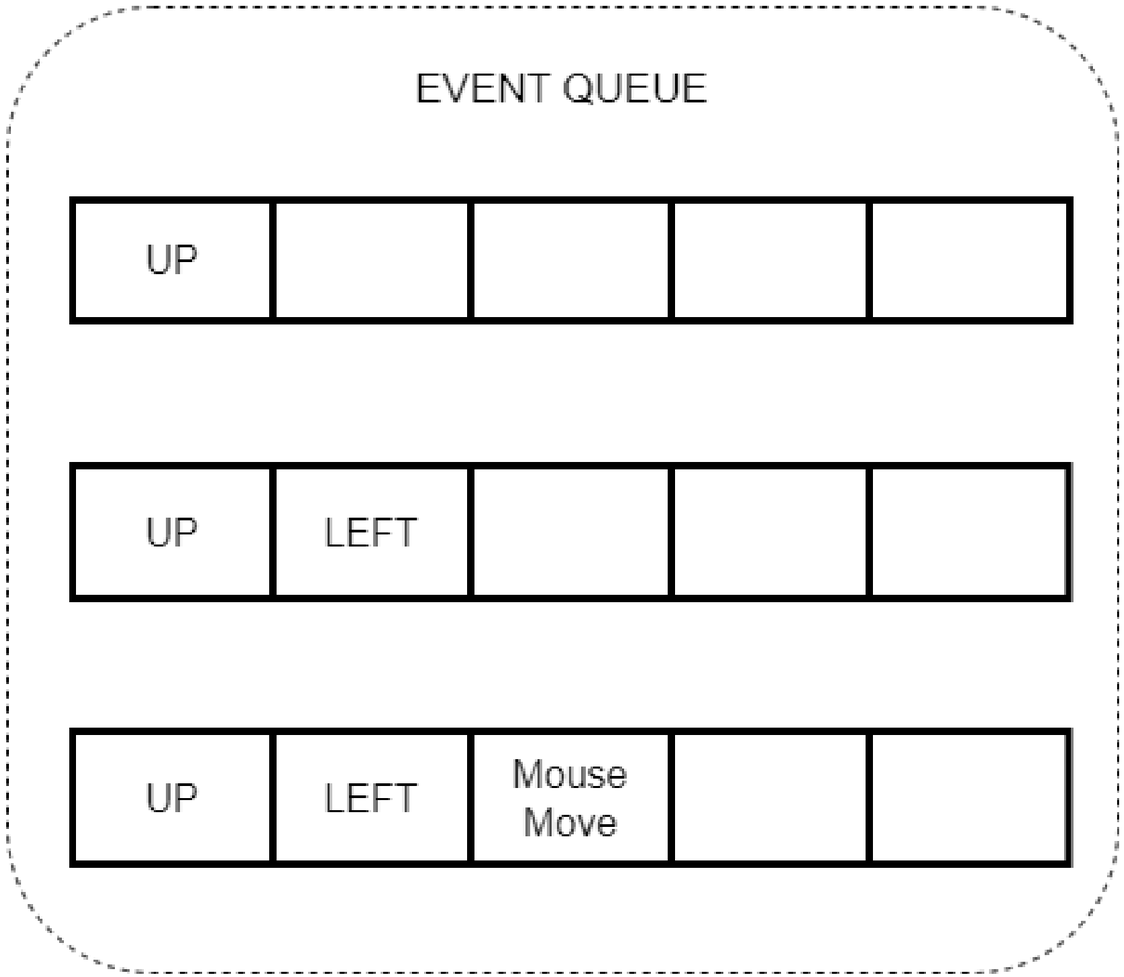
Uses:

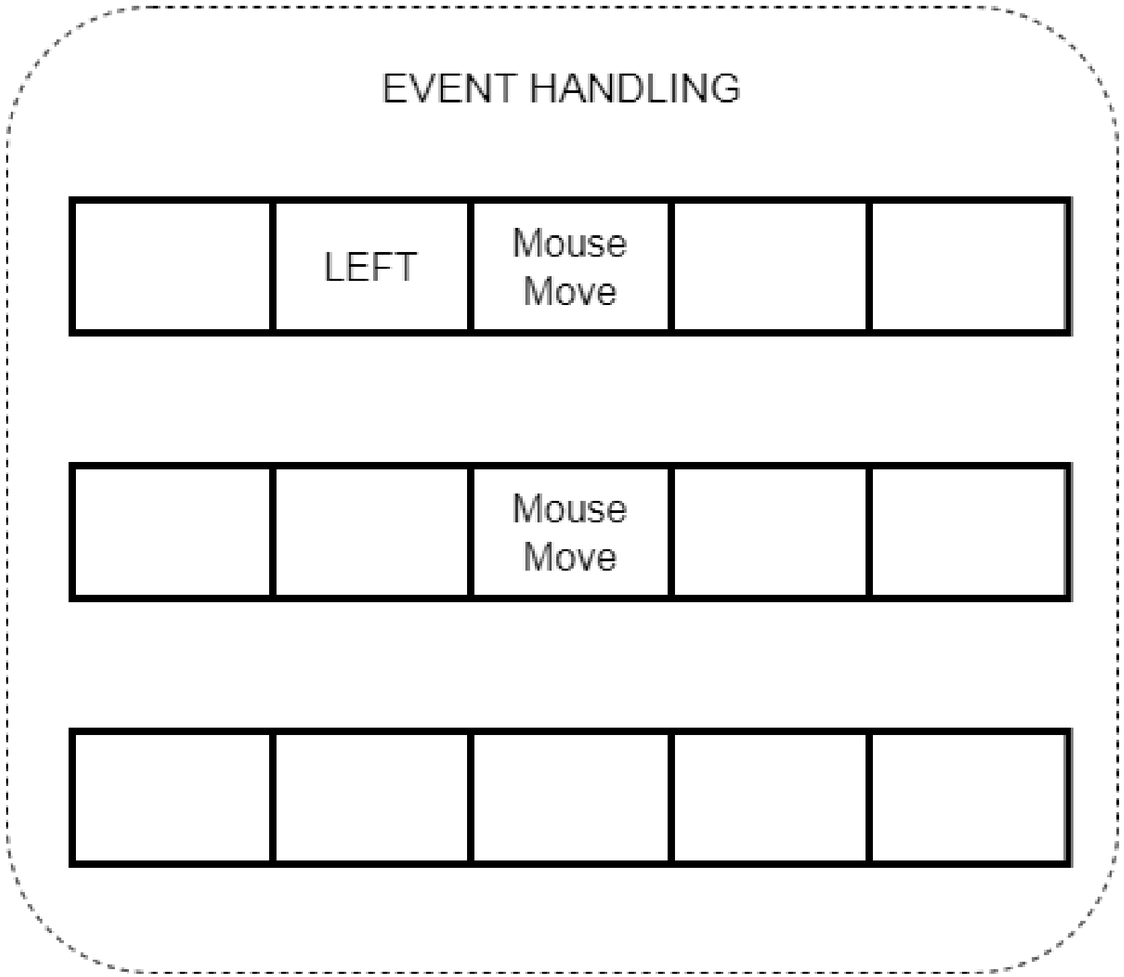
- **Event handling in applications**
- **Resource and process management in operating systems**
- **Handling client requests to servers**

and many more!

Queue: Event Handling

IT Club





Implementation: Stacks and Queues

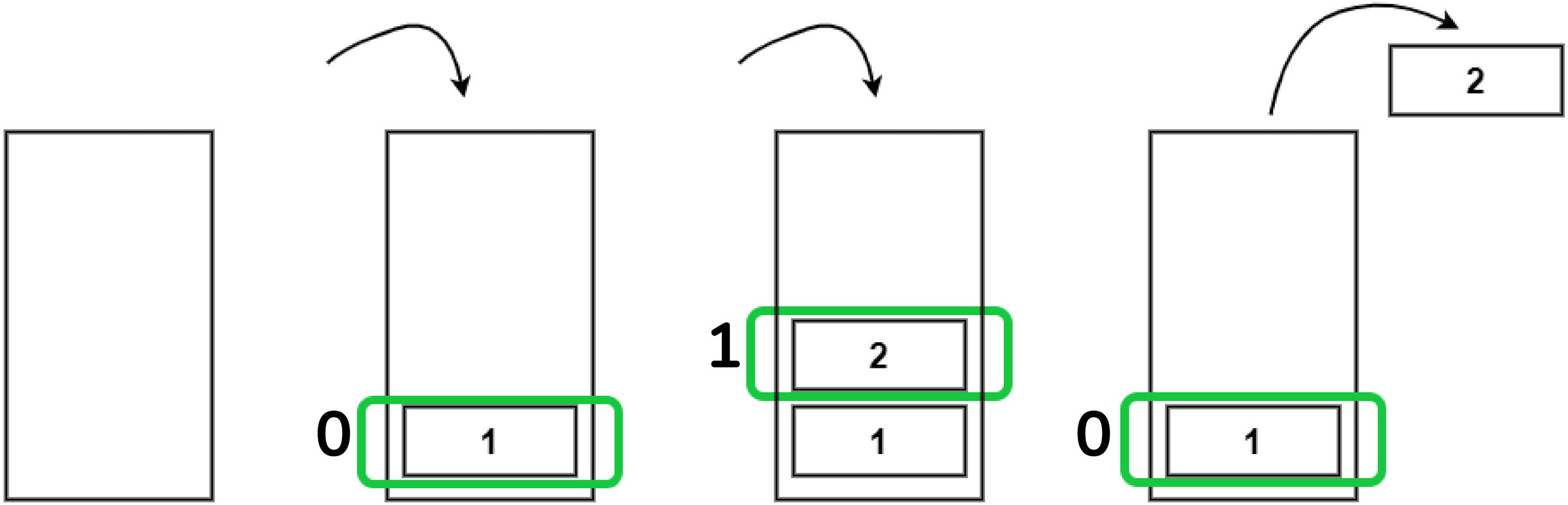
You can use **either an array** or **a linked list** to represent a stack/queue.

- **Array:** Generally **static** implementation
- **Linked list:** **Dynamic** implementation

Array implementation

- static stack/queue **(fixed size)**
- **simpler**
- can be used for **smaller number of data** or when the max number is known
- **overflow when completely full**

Implementation



TOP represents the index of top element

PUSH: Increase TOP by one before pushing

POP: Decrease TOP by one after popping

Linked list implementation

- dynamic stack/queue **(variable size)**
- **a bit more complex**
- used when **number of data is unknown**
- doesnt overflow since **new nodes can be added almost indefinitely**

An array implementation is given with the resources.
A linked list implementation is left as an exercise.

Any questions?

Thank you!

Part 2.. continued

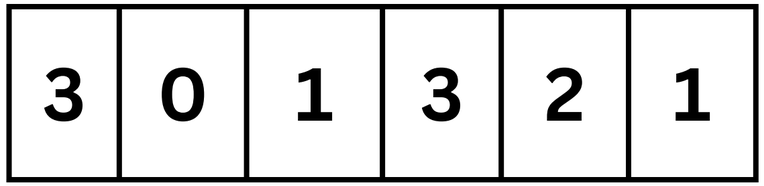
Sorting!

ordering elements in
ascending or descending manner

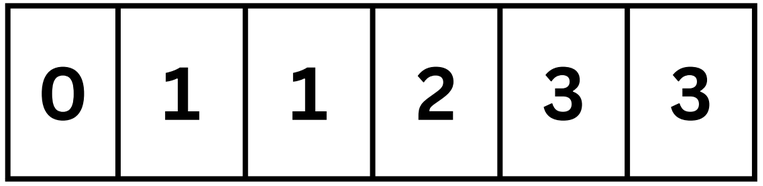
Sorting examples

Ascending manner

Before sorting

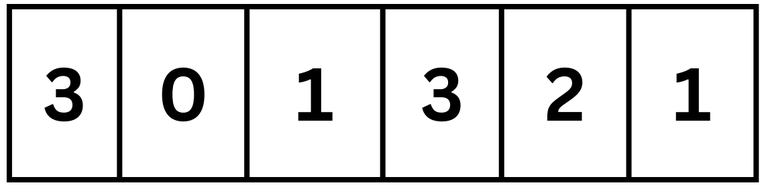


After sorting

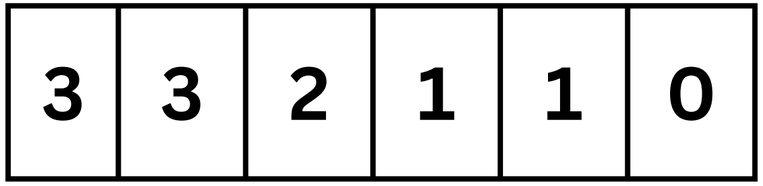


Descending manner

Before sorting



After sorting



Sort based on the first letter

Before sorting

mango	peach	apple	pineapple

After sorting

apple	mango	peach	pineapple

In-place sort

- An in-place sorting algorithm is one that operates directly on the given input array without requiring additional memory.
- O(1) auxiliary space complexity

Stable sort

- If a sorting algorithm is stable then it will retain the original order of the data after sorting is completed.
- If there are duplicates of data then the duplicate piece of data that was on the left will remain on the left and the right will remain to the right after sorting is done.

Sort based on the first letter

Before sorting

mango	**peach**	apple	*pineapple*

After sorting

apple	mango	**peach**	*pineapple*

Stable sort

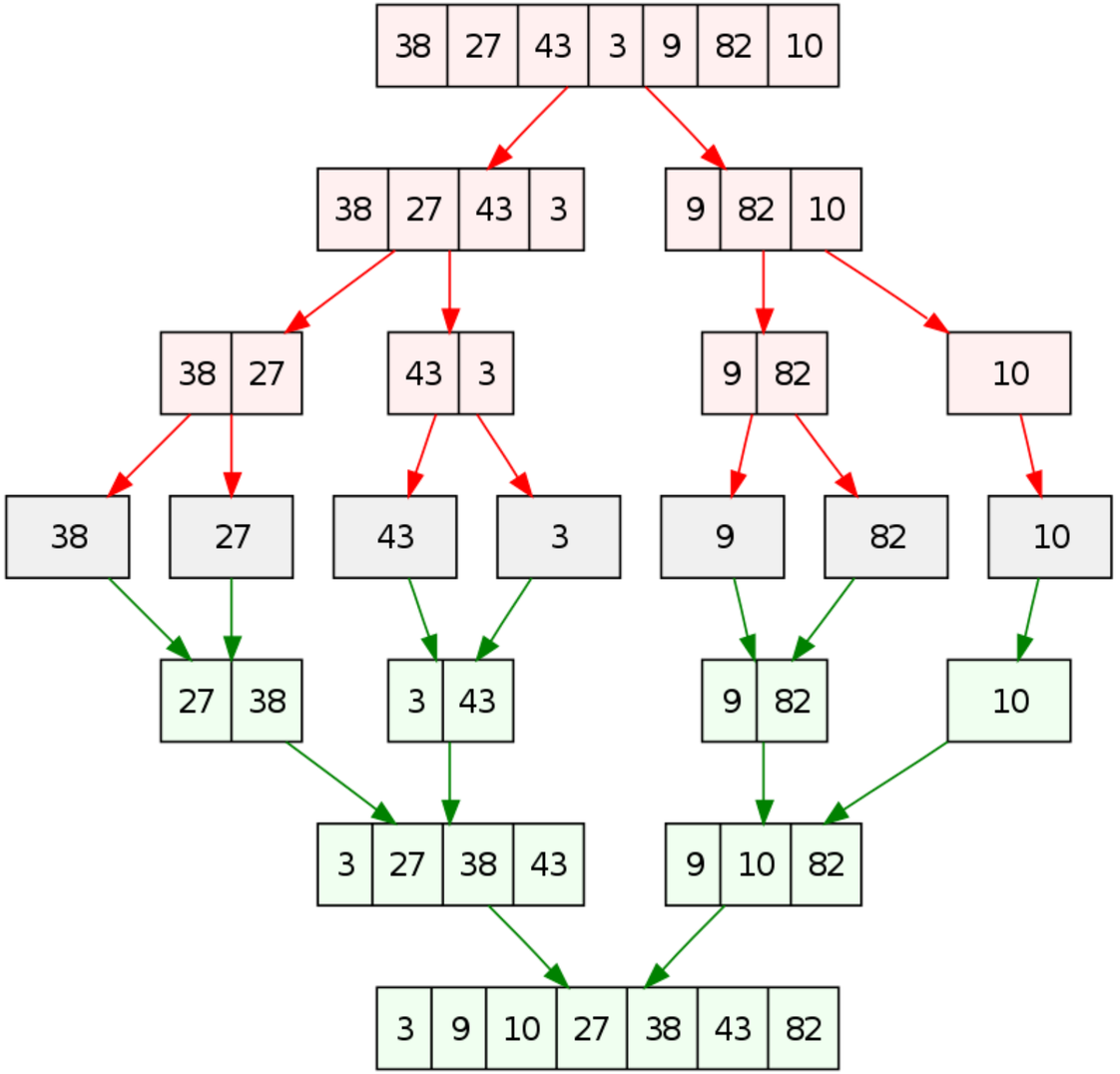
31 23 27 31 57

A case in point: Merge sort

Algorithm

1. Divide the unsorted list into *n* sublists, each containing one element (a list of one element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

Visualize Merge Sort



Calling sort on the list

```c
int main()
{
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int size = sizeof(arr) / sizeof(arr[0]);

    merge_sort(arr, 0, size - 1);

    printf("\nSorted array: ");
    for (int i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Dividing Recursively

```
void merge_sort(int arr[], int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;

        // Sort first and second halves
        merge_sort(arr, left, mid);
        merge_sort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}
```

Merge

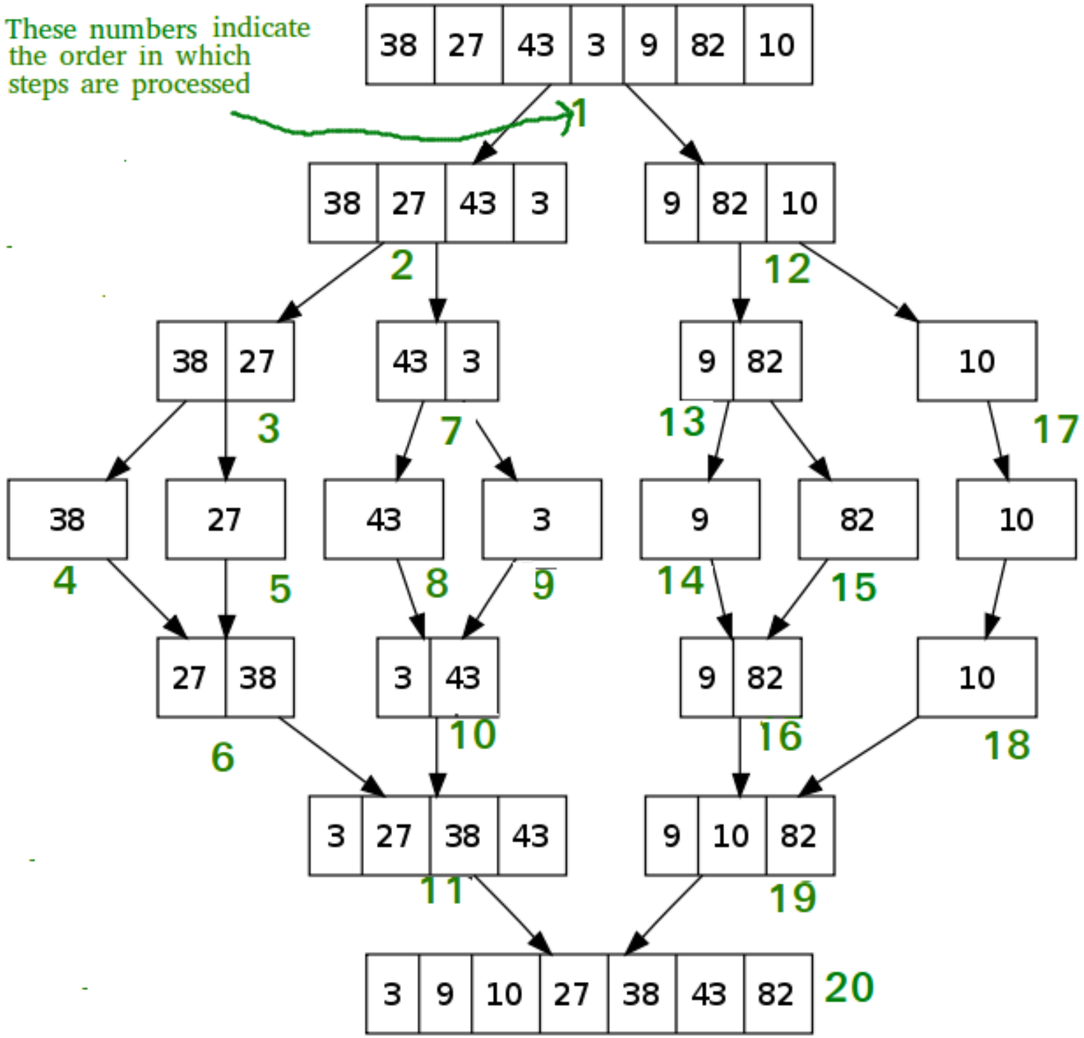
```
void merge(int arr[], int left, int mid, int right)
{
    // Find sizes of two subarrays to be merged
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary arrays
    int left_arr[n1];
    int right_arr[n2];

    // Copy data to temporary arrays left_arr[] and right_arr[]
    for (int i = 0; i < n1; i++)
    {
        left_arr[i] = arr[left + i];
    }

    for (int j = 0; j < n2; j++)
    {
        right_arr[j] = arr[mid + 1 + j];
    }
```

Implementation

IT Club

```
    // Merge the temporary arrays back into arr[left..right]
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2)
    {
        if (left_arr[i] <= right_arr[j])
        {
            arr[k] = left_arr[i++];
        }
        else
        {
            arr[k] = right_arr[j++];
        }

        k++;
    }

    // Copy the remaining elements of left_arr[], if there are any
    while (i < n1)
    {
        arr[k++] = left_arr[i++];
    }

    // Copy the remaining elements of right_arr[], if there are any
    while (j < n2)
    {
        arr[k++] = right_arr[j++];
    }
}
```

Visualize Merge Sort



Properties

IT Club

- Divide-and-conquer algorithm
- Stable sort
- Time complexity O(n log n) for worst, average, and best case.
- Space complexity O(n) so *not* in-place sorting algorithm
- Best algorithm to sort linked list

Searching

So you need to find a specific value from a whole bunch of data.
How do you go about it?

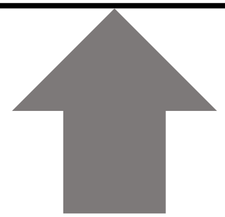
For example, you have bunch of data of people

Name	Ram	Shyam	Hari	Sita	Keanu	Old
Age	30	25	45	23	23	123
Address	KTM	KTM	PKR	PTN	BKT	PKR

and you want to find the people who live in **Pokhara**

Look through all the data **sequentially** and find out, right?

Name	Ram	Shyam	Hari	Sita	Keanu	Old
Age	30	25	45	23	23	123
Address	KTM	KTM	PKR	PTN	BKT	PKR



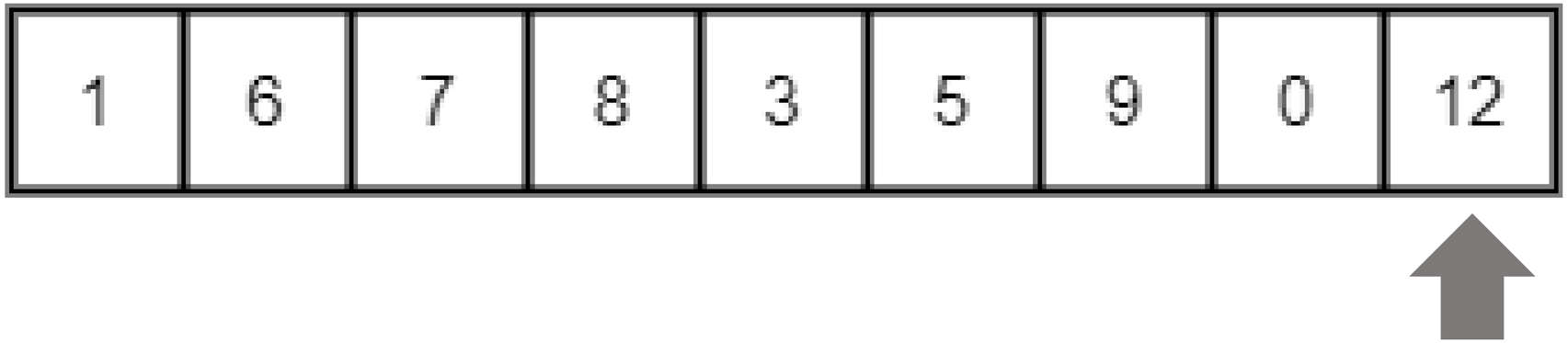
Well, that does work!

This is called **linear search!**

However, is it a good way for searching in general?

What is its time complexity?

The algorithm has to go through **n elements** of the array in the worst case.



For example, searching for 12.

So, if you said linear time complexity

O(n)

you're correct!

So, what if there is a **HUGE** amount of data?
Like **thousands or even more?**

Name														....	
Age														....	
Address														....	

Then searching through the whole data sequentially **isn’t the best idea for performance.**

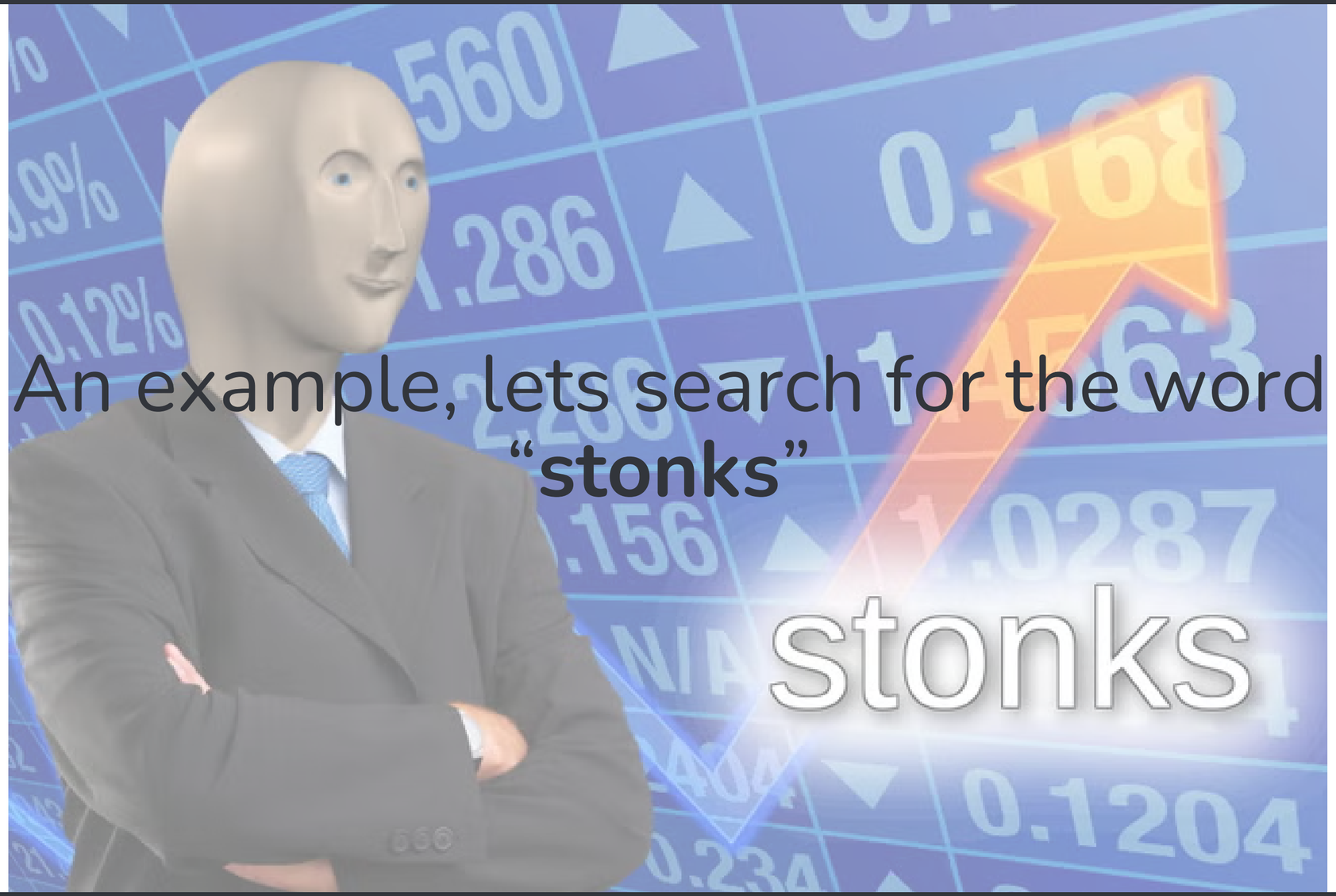
It will get the job done, but it isn’t efficient at all.

A better option:
Binary Search

Well, if you have a **sorted array,**
then there is a better option.

Binary search

Binary search is essentially what you do when you **look for a word in a dictionary**.


An example, lets search for the word
“stonks”
stonks

At first you don't know where
"S" is.
You only know that it is between the start **"A"** and end **"Z"**

You go to a random page between **A** and **Z**, **say J.**

You compare S with J, and conclude that
S comes after J.
So you can effectively **rule out the region A-J**.

Now you go to a page between J and Z,
say T.

Comparing S with T, we see that **S comes before T**.
So, you can safely rule out T-Z.

Repeating this a few more times, we effectively get to S in a short amount of steps.

Chosen page **R**

S comes after R
Possible range R-T

Chosen page **S.**
Done!

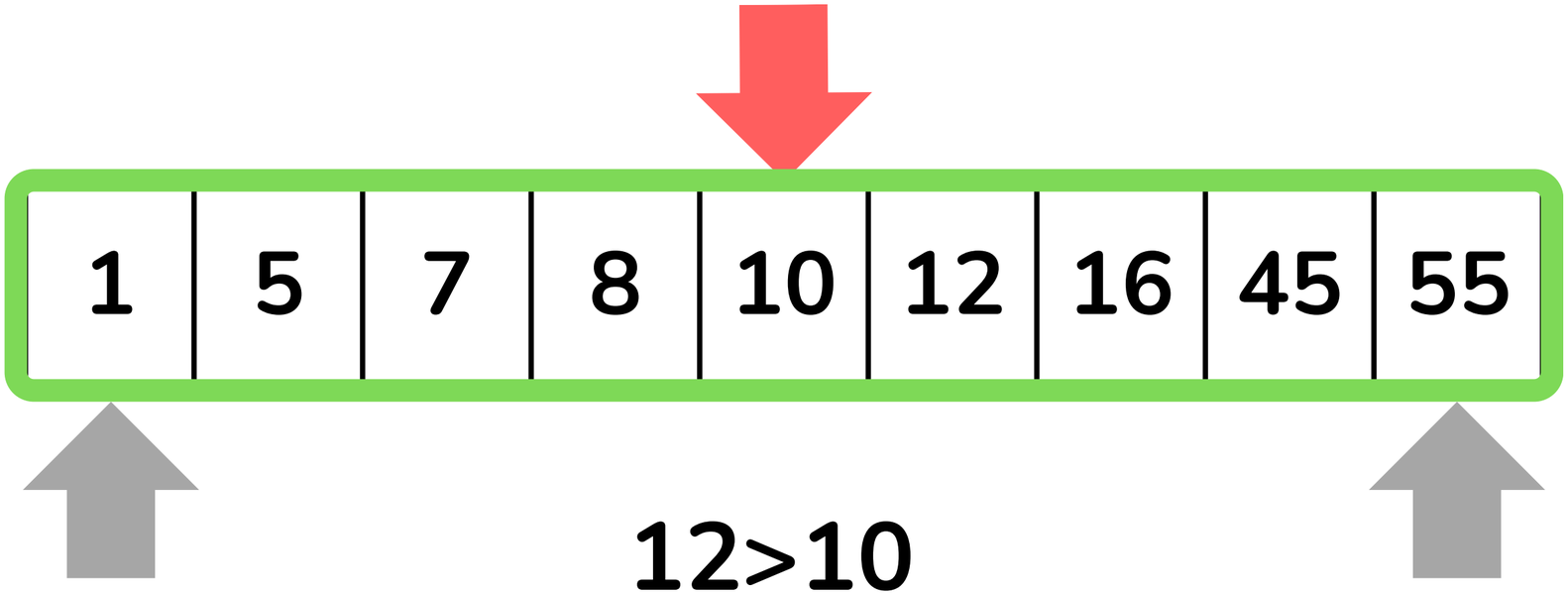
It is apparent that this is way more efficient than having to check each page sequentially to find S.

Binary search is just this, except that you choose **the mid point of the current possible range** on each step.

Lets take an example:
search for 12 in this sorted array.

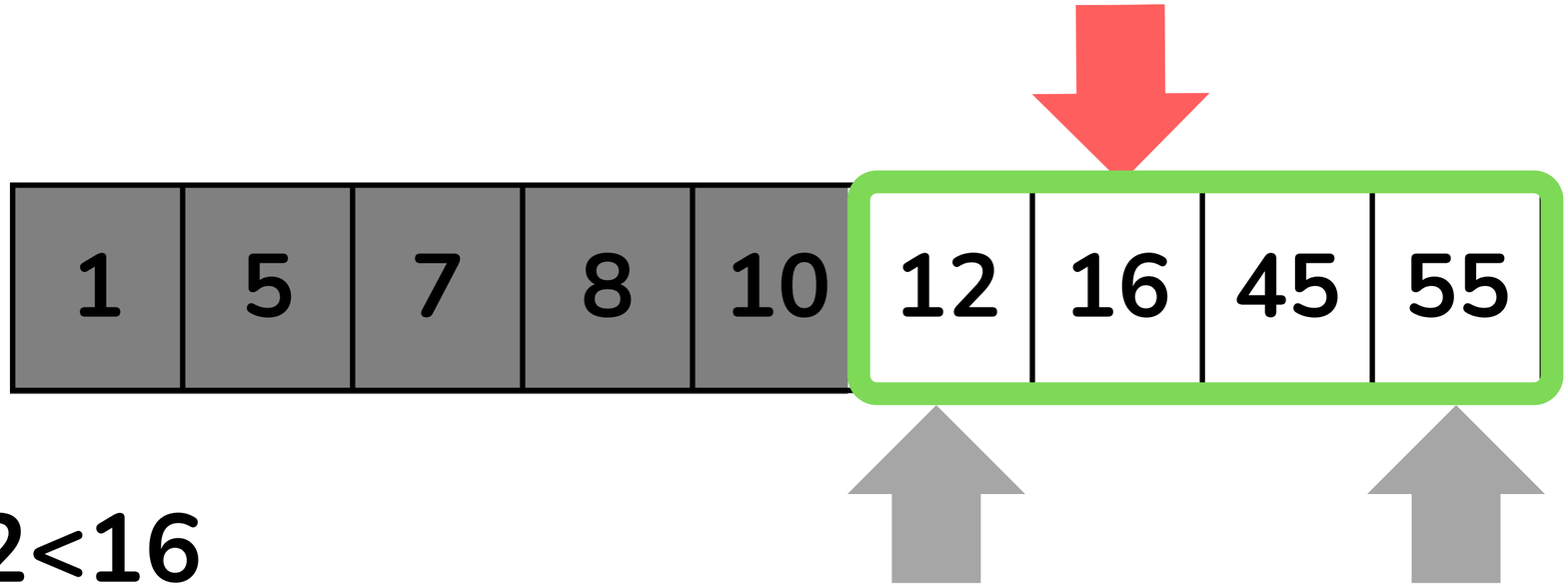
1	5	7	8	10	12	16	45	55

Mid index = (0+8)/2 = 4



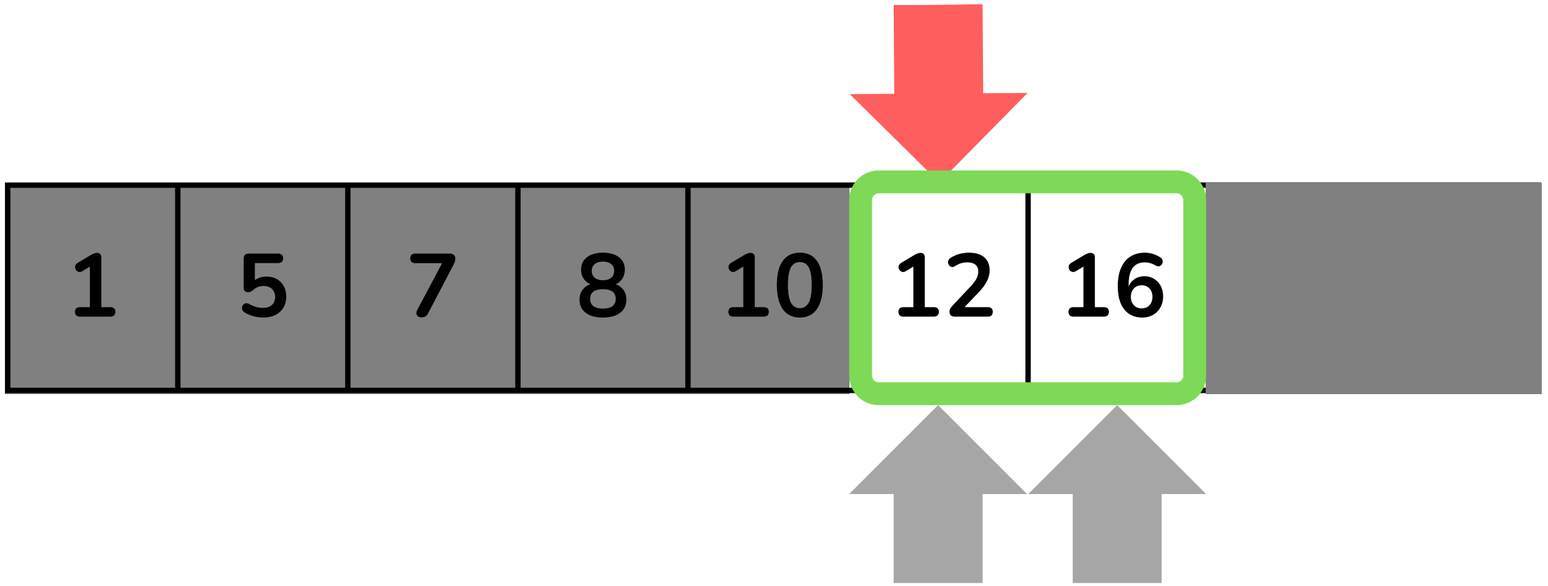
12>10
so it lies to the right

Mid index = (5+8)/2 = 6.5 -> 6



12<16
so it lies to the left

Mid index = (5+6)/2 = 5.5 -> 5



12=12
Done!

Essentially, you're halving the possible range each step!

So, the search in an n-element array goes until

$$\frac{n}{2^x} = 1$$

So, $x = \log_2 n$ is the number of steps required.

Hence, the **time complexity** of binary search is **O(log n)**

Binary Search

IT Club

```c
int binarySearch(int arr[], int low, int high, int target) {
    while (low < high) {
        int mid = low + (high - low) / 2;

        // Check if the target is present at the middle
        if (arr[mid] == target)
            return mid;

        // If the target is greater, ignore the left half
        else if (arr[mid] < target)
            low = mid + 1;

        // If the target is smaller, ignore the right half
        else
            high = mid;
    }
    return -1;
}
```

Binary Search

IT Club

Now, binary search is good for sorted arrays,
but all arrays aren’t sorted!

Enter.. HASHTABLES!

Hashtables are a data structure that map a value (called the **key**)
to
an index in an array where the data is stored (called the **value**).

Keys and values

“PKR”: “Ram”

key value

Different key and value

10: 10

key value

Same key and value

Hash function

Hash tables use a **hash function** to convert a key into a **unique number/hash** which is used as an index in the hash table.

A hash function, in essence, converts the bytes of the key, to a unique number.

An example,
the **adler32 hash function.**

Don’t worry! You don’t need to understand this.

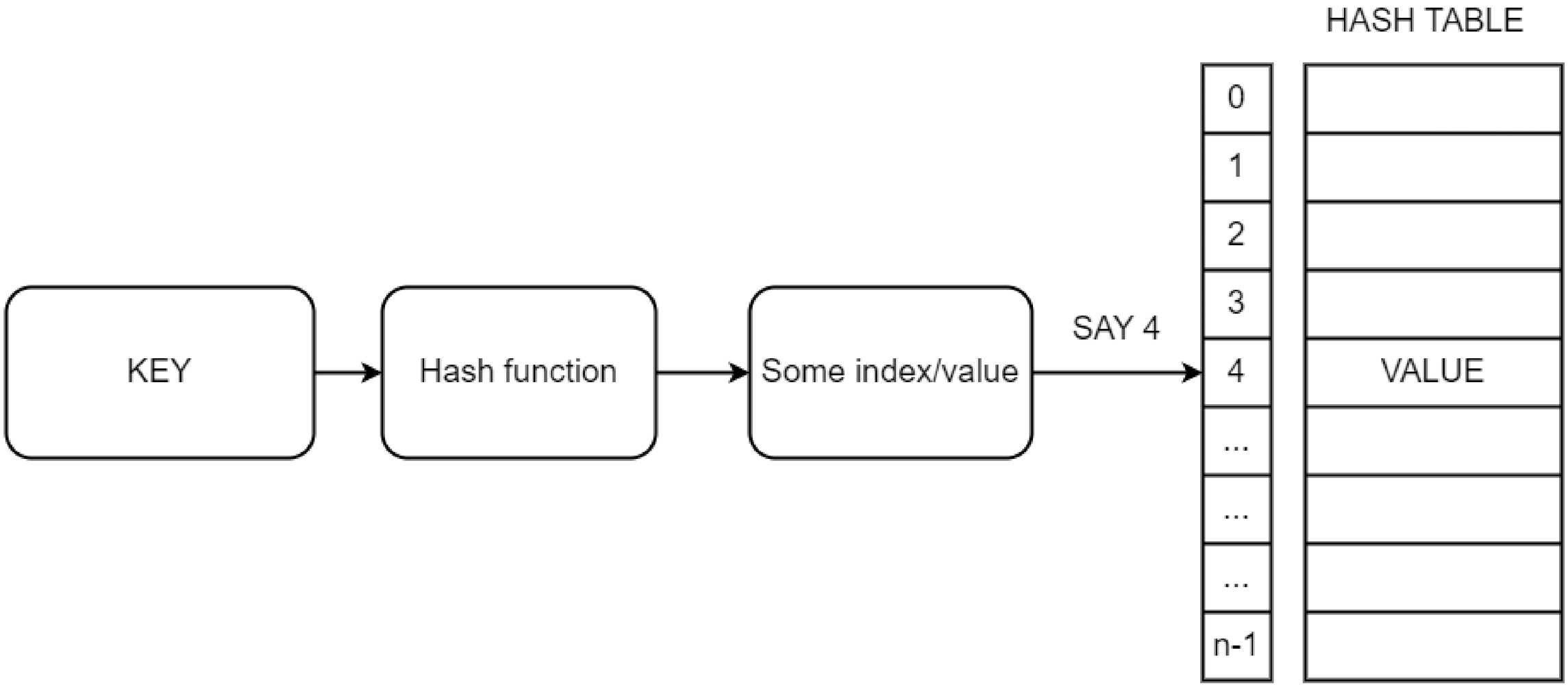
```c
const uint32_t MOD_ADLER = 65521;

uint32_t adler32(unsigned char *data, size_t len)
{
    uint32_t a = 1, b = 0;
    size_t index;

    for (index = 0; index < len; ++index)
    {
        a = (a + data[index]) % MOD_ADLER;
        b = (b + a) % MOD_ADLER;
    }

    return (b << 16) | a;
}
```

Hashtables are usually **fixed length arrays,** meaning that the **generated index needs to be within the array bounds.**

So, **modulo division** is widely used with a prime number fixed size.

actual index = hash value % hash table size

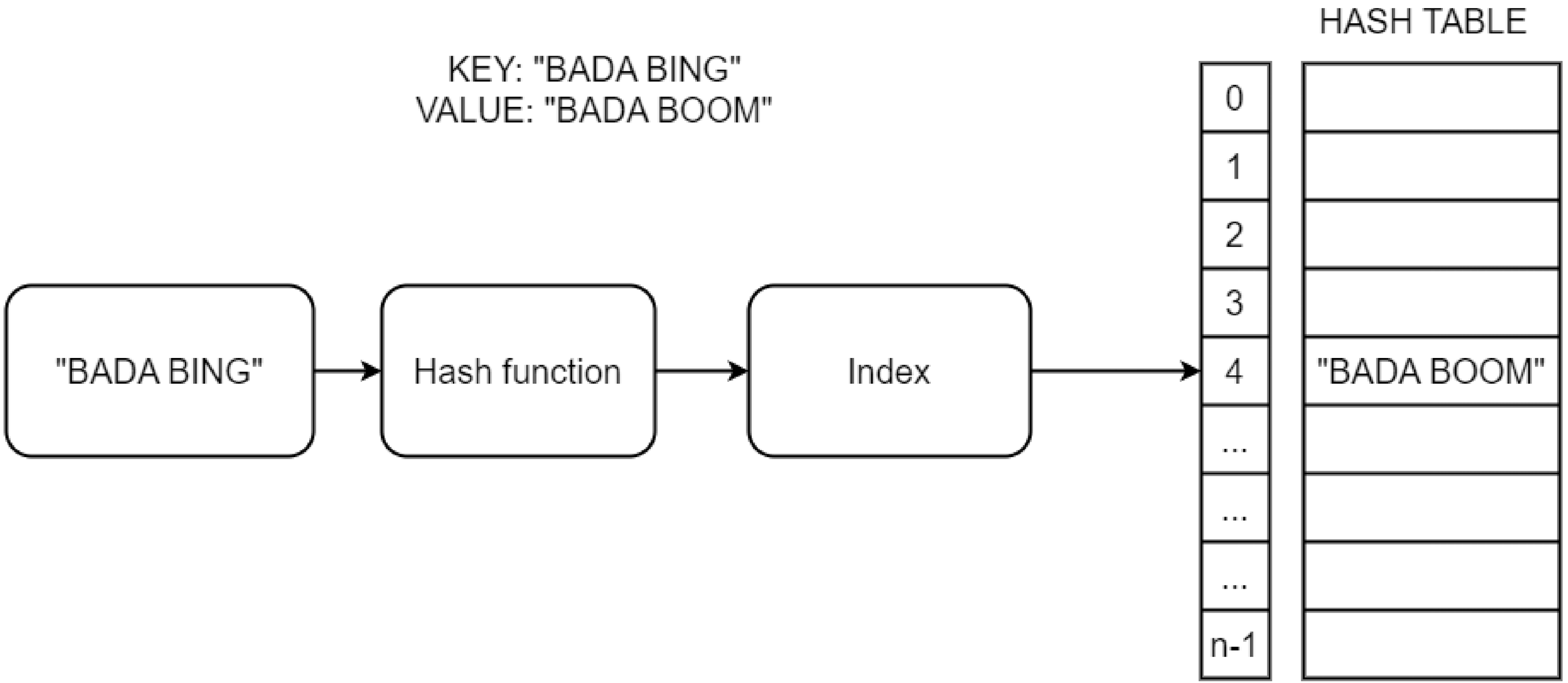
Hashtables


HASH TABLE
KEY
Hash function
Some index/value
SAY 4
0
1
2
3
4
...
...
...
n-1
VALUE

Say you want to add the key value pair
(“Bada Bing”, “Bada Boom”)

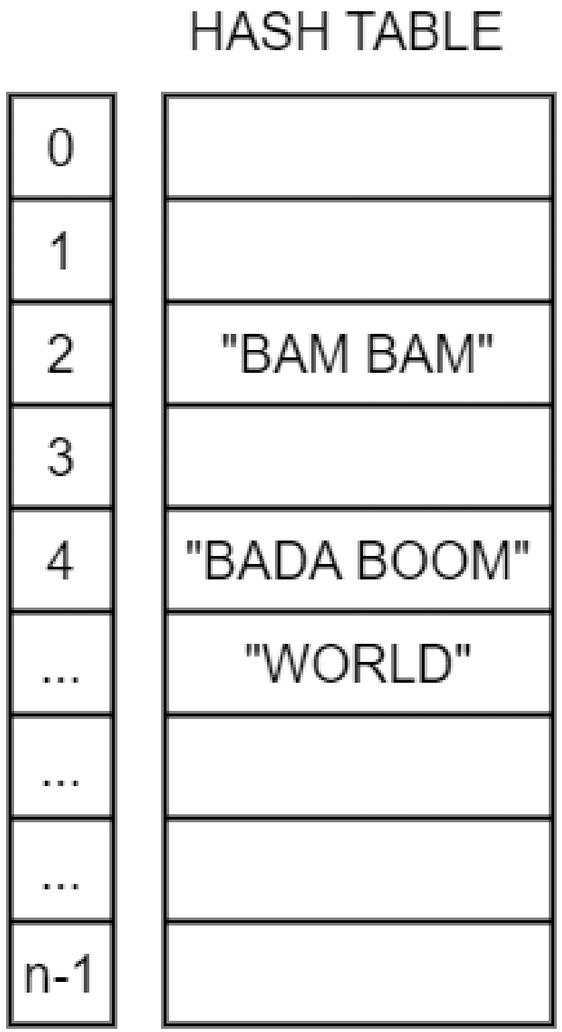
Hashtables

IT Club



After adding some more keys-values,

- **(bim bim, bam bam)**
- **(hello, world)**

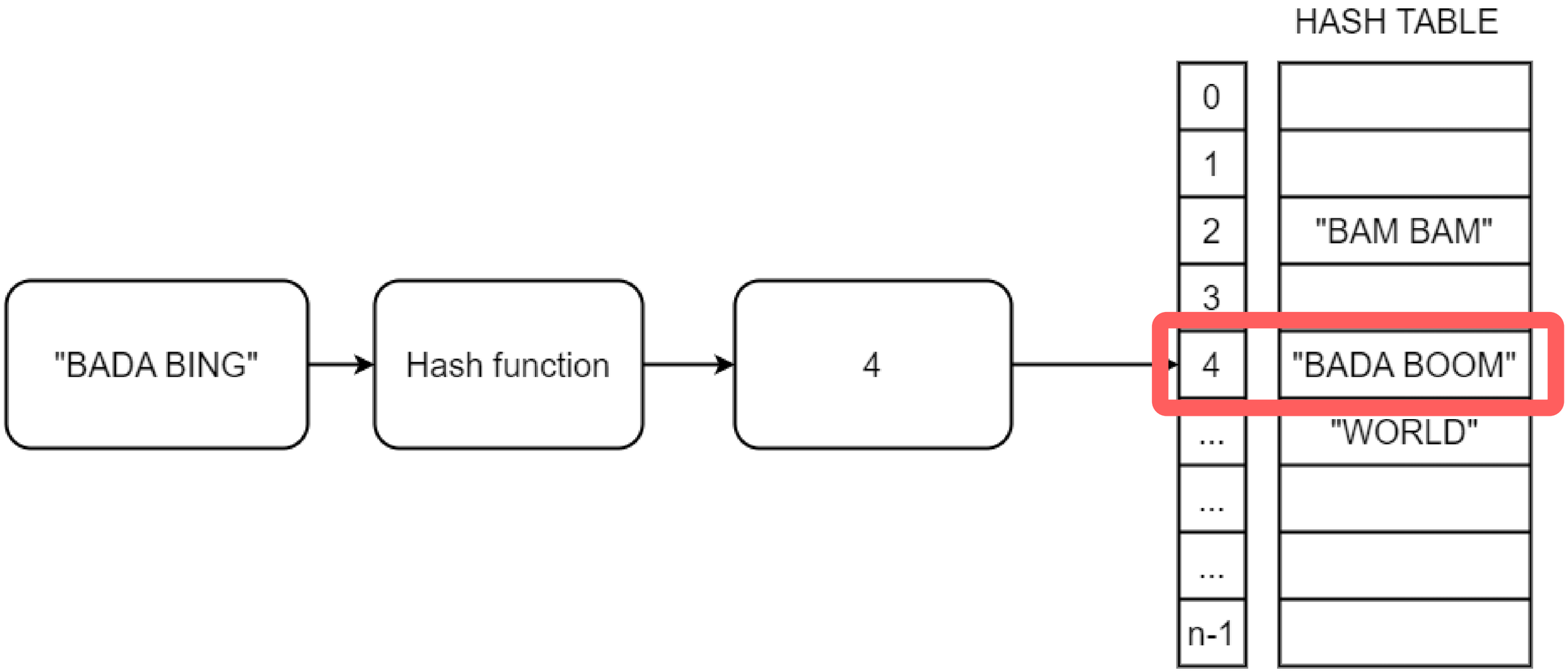


Now, if you want to search for the value corresponding to the key **“Bada Bing”**
it is very simple!

Just throw the key through the **hash function again**!

It gives the index which was calculated when inserting the value!

Hashtables


HASH TABLE
"BADA BING"
Hash function
4
0
1
2
3
4
...
...
...
n-1
"BAM BAM"
"BADA BOOM"
"WORLD"

You get the exact index of the required value when using the hash function!

This allows for constant time O(1) searches!

But wait, what if there are more than one value for a given key?

That, my frendo, is called a **collision.**

For example, consider the example from earlier

- **(PKR, Hari)**
- **(PKR, Old)**

Both pairs have the **same key**, but **different values,** meaning they get mapped to the **same index** by the hash function.

Name	Ram	Shyam	Hari	Sita	Keanu	Old
Age	30	25	45	23	23	123
Address	KTM	KTM	PKR	PTN	BKT	PKR

So, how do we store these values?

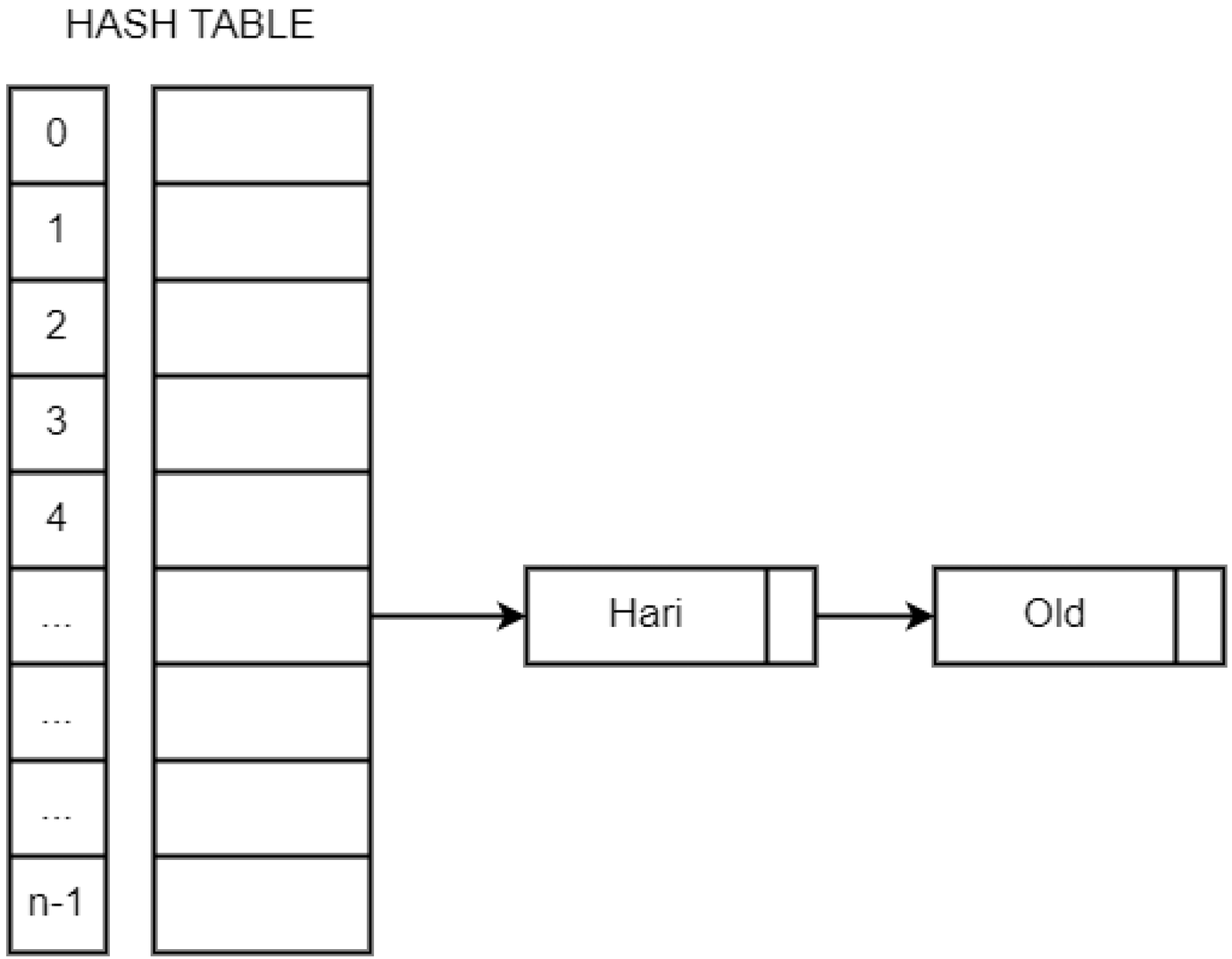
One answer is **chaining**

There are two types of chaining:
Open and **closed**

Open chaining

- the hashtable stores a **linked list of values** in each array index **instead of a single value**
- each collision adds a node to the list
- allows for quick search since linear search is done for only the nodes in a list

Then for the example, the values would be stored as



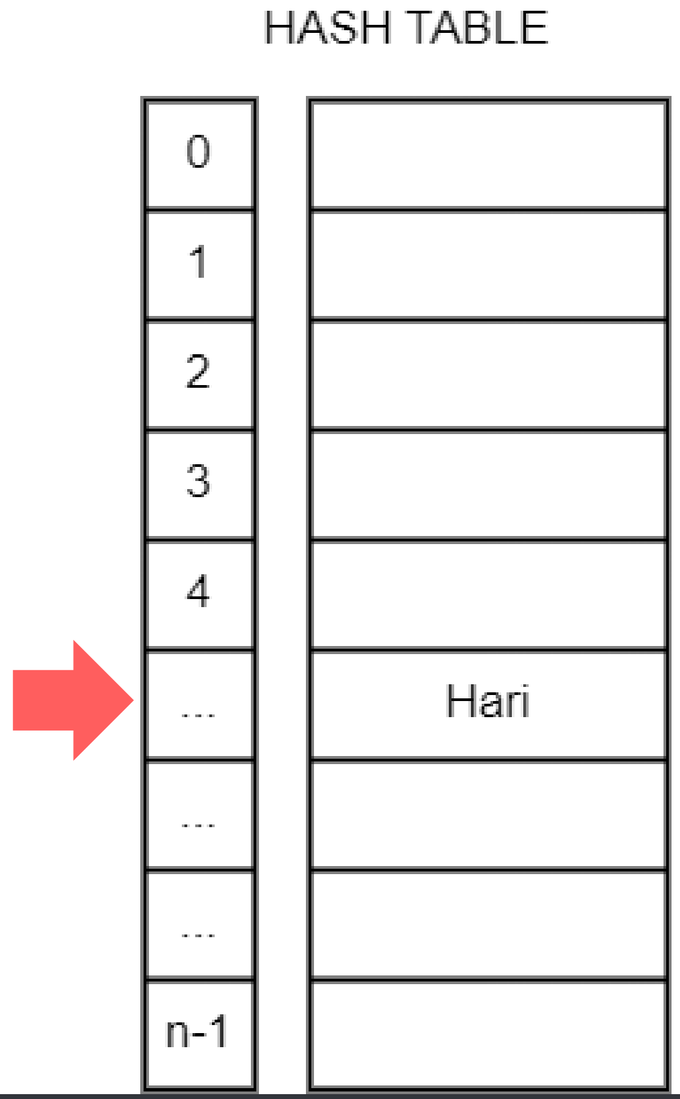
Closed chaining

- the new value is **stored at a different index** in the same table, by manipulating the hashed index
- Examples include linear probing, quadratic probing, rehashing, etc.

First, (PKR, Hari) is added

Then, there is a collision on adding **(PKR, Old)** due to same key

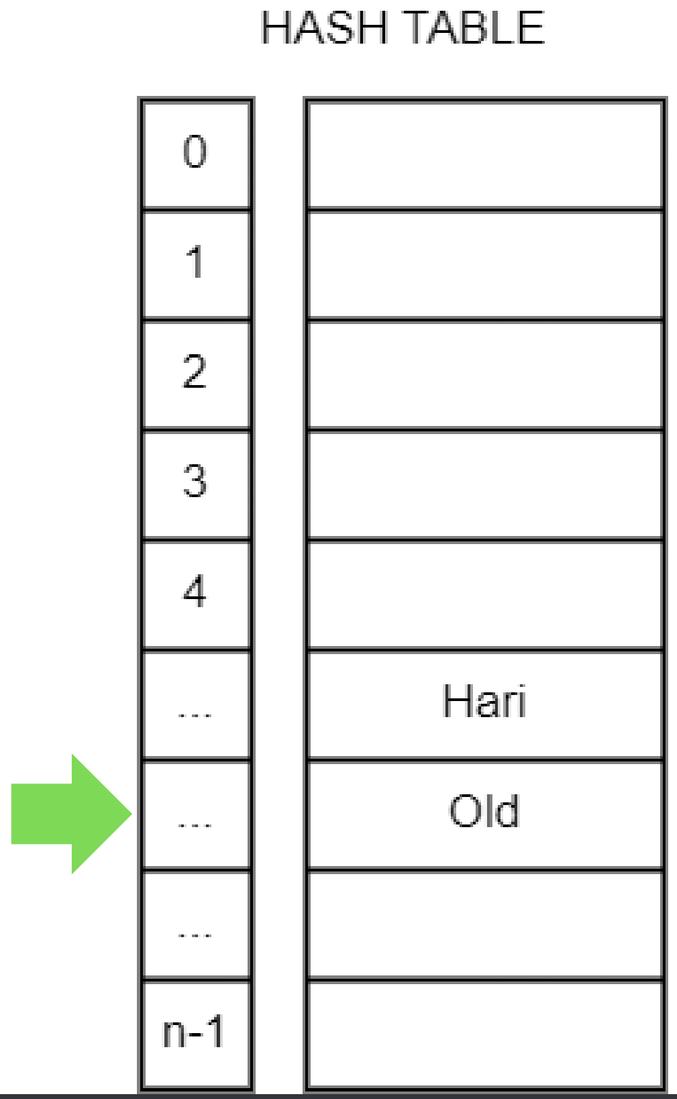
So, the index is **manipulated until a new available index is found**



In **linear probing**, the index right after the hashed value is checked.

new = old + i

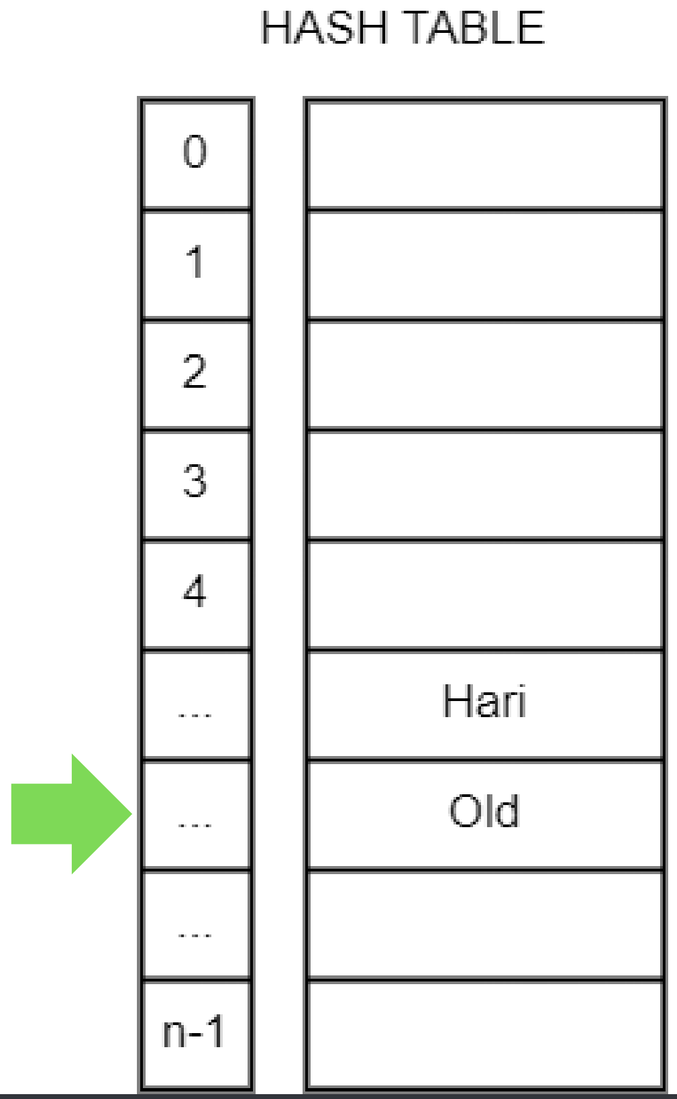
i = number of manipulations



In **quadratic probing**, the next index is given by

$$\textbf{new} = \textbf{old} + \textbf{i}^2$$

i = number of manipulations



For interactive visualization, there are some websites listed in the resources.

Any questions?

Thank you!

Kalpesh Manandhar

KalpeshManandhar

Surav Shrestha

suravshrestha