STL LIBRARY BASICS

Containers, Algorithms and Iterators

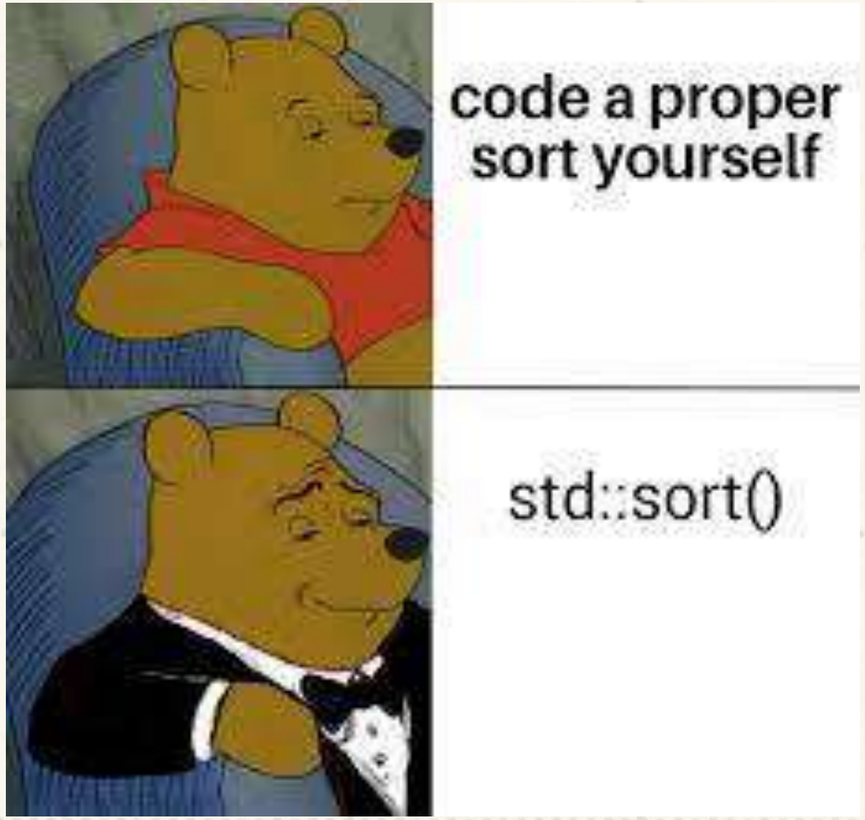
-Session By
ABHI JAIN
Expert on CodeForces
IIT'B CSE 27'

algorithms

WHAT IS STL LIBRARY?

containers → vectors, set

FRIEND LOOKING AT MY CODE: WHY DIDN'T YOU USE THE METHOD IN THE STANDARD LIBRARY
ME AFTER SPENDING 3 HOURS CODING A CUSTOM IMPLEMENTATION
The
WHAT


code a proper sort yourself
std::sort()

Standard Template Library (STL) is a collection of algorithms, data structures, and other components that can be used to simplify the development of C++ programs.

The STL provides a range of containers, such as vectors, lists, and maps, as well as algorithms for searching, sorting and manipulating data.

The STL also provides a way to write efficient code. Many of the algorithms and data structures in the STL are implemented using optimized algorithms, which can result in faster execution times compared to custom code.

IMPORTANT NOTE

Since, majority of people do CP in C++, that's why we are discussing the STL LIBRARY

ITS JUST A TOOL !!!

If you are already doing CP in Python, Java or any other language. You can find similar tools on the web. We will try to add the resources in the description later on.

From the next session things will be more focused on Concepts and Algorithms.

IMPORTANT NOTE

This lecture would have a lot of things to memorize so keep ur pens and notebooks together and don't forget to revise and practice after the session!!

PRE-REQUISITES FOR THIS SESSION

Lot of Passion to Learn!

Familiarity with basic C\C++ concepts like data-types, if-else statements, for-while loops, functions, arrays and basics of pointers.

Familiarity with the idea of Time-Complexity.

Pre-Recorded Lecture of Time Complexity will be uploaded soon watch it, if you have doubts!!

TIME-COMPLEXITY BASIC IDEA

The time complexity of an algorithm estimates how much time the algorithm will use for some input.

The constraints for time and the variables are provided in the CP problem.

According to them we decide our data structure and algorithms.

```
for ( int i=0 ; i<n, i++) {
    cout << i;
}
```

1 → n → n → n

$3n+1$ steps

Big-Oh

$O(n)$

Time Complexity :-

proportional to n in this case.

O, Θ

$n^2+n+1 \longrightarrow O(n^2)$

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

1	2	4	8	2^{k-1}
1	2	3	4	k^{th} step

$2^{k-1} \geq n$

$k \geq 1 + \log_2 n$

no. of steps $\propto k$

$\log_2 n$

$\boxed{O(\log n)}$

```
for( i=0, i<n ; i+=2) {
    cout << i << " ";
}
```

$1 \to 0$

$2 \to 2$

$3 \to 4$

$4 \to 6$

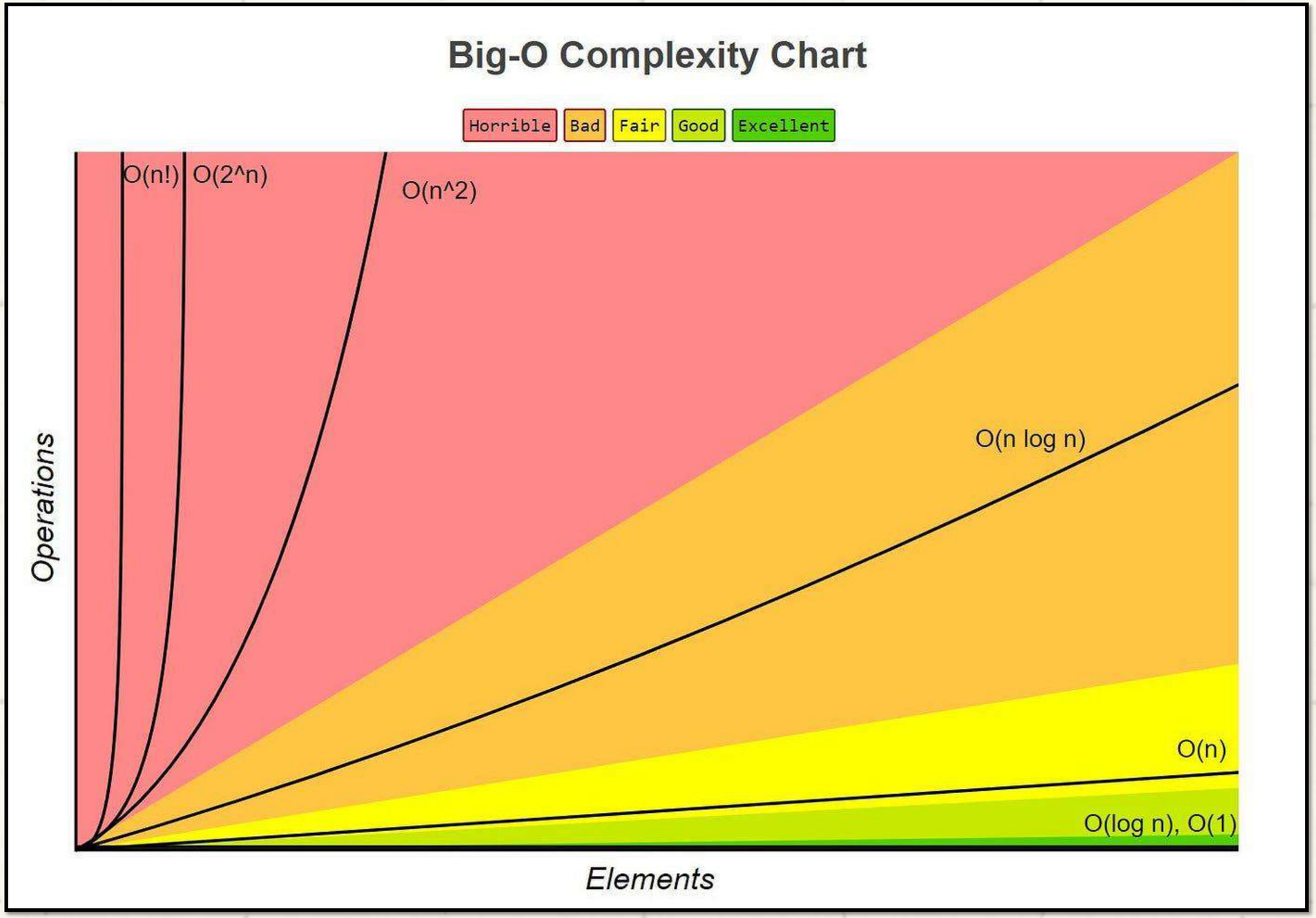
$5 \to 8$

$k \to 2(k-1)$

$2(k-1) \geq n$

$\Rightarrow k \geq \frac{n}{2} + 1$

$O\left(\cancel{\frac{n}{2}}\right) \to \underline{\underline{O(n)}}$


Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

BIG – O NOTATION

O(1): Constant Time

O(log n): Logarithmic Time

O(n): Linear Time

O(n log n): Linearithmic Time

O(n^2): Quadratic Time

O(2^n): Exponential Time

O(n!): Factorial Time

For Details watch the pre-recorded session on Time Complexity

input size	required time complexity for 1s processing time
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ or $O(n)$
n is large	$O(1)$ or $O(\log n)$

HOW TO INCLUDE THE STL LIBRARY?

include < bits/std c++. h >

```
#include <bits/stdc++.h>
//this header file contains all the necessary header files of STL Library

using namespace std;

int main() {
    // solution comes here
}
```

To know more about **<bits/stdc++.h> in C++** visit https://www.geeksforgeeks.org/bitsstdc-h-c/

SORTING

$O(N^2)$ ALGORITHMS

Bubble Sort, Insertion Sort, Selection Sort, etc..

We won't be discussing these algorithms here otherwise it would take a lot of time.

To know more about these algorithms visit:
https://www.geeksforgeeks.org/sorting-algorithms/

To visualise these algorithms visit:

https://algorithm-visualizer.vercel.app/sorting#

MERGE SORT
O(N LOG N)

Merge Sort is based on recursion. dividing → recursively sorting → merging

Merge sort sorts a **subarray array** $[\boldsymbol{a} \ldots \boldsymbol{b}]$ as follows:

(Note: a, b are indexes)

1. If $\boldsymbol{a} = \boldsymbol{b}$, do not do anything, because the subarray is already sorted.

It means if array has more than 2 elements than divide the array in two

2. Calculate the position of the middle element: $\boldsymbol{k} = \frac{\boldsymbol{a}+\boldsymbol{b}}{\boldsymbol{2}}$

divide

3. Recursively sort the subarray array $[\boldsymbol{a} \ldots \boldsymbol{k}]$.
4. Recursively sort the subarray array $[\boldsymbol{k} + \boldsymbol{1} \ldots \boldsymbol{b}]$.

recursively sort

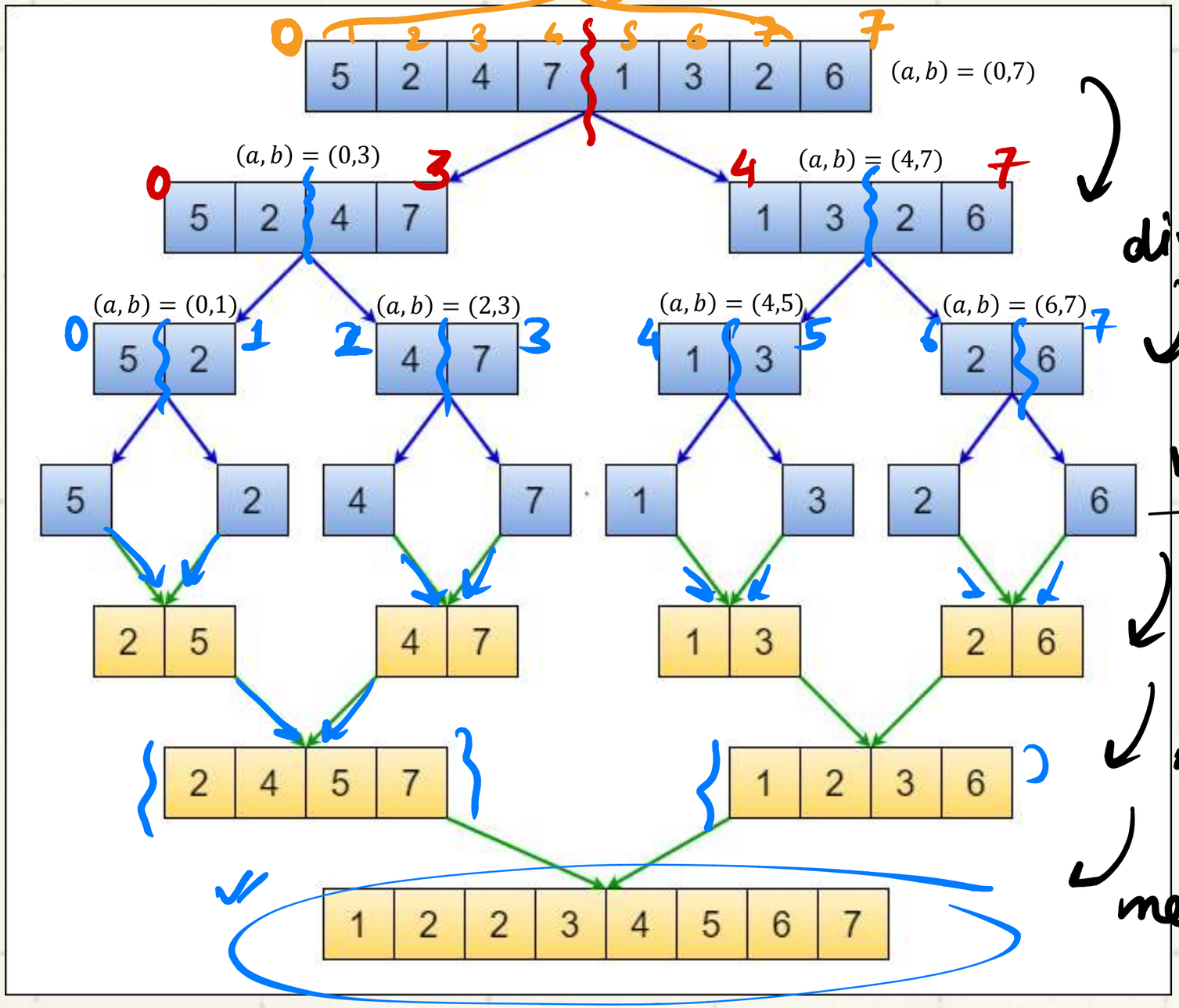
5. **Merge the sorted subarrays** array $[a \ldots k]$ and array $[k + 1 \ldots b]$ into a sorted subarray array[a...b].

merge

The code for Merge Sort in C++ can be found here:
https://www.geeksforgeeks.org/merge-sort/?ref=lbp

```
// begin is for left index and end is right index
// of the sub-array of arr to be sorted
void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end) //sorting ends when begin is greater than or equal to end
        return;
    //find the middle point of the array
    int mid = begin + (end - begin) / 2;
    //recursively sort the left and right subarrays
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    //merge the sorted subarrays
    merge(array, begin, mid, end); //use merge function to merge the subarrays
}
```

UNDERSTANDING MERGE SORT THROUGH AN EXAMPLE



fibonnaci

$a_n = a_{n-1} + a_{n-2}$

divide → n steps

divide → n steps

divide → n steps

From this step $a = b$. Therefore, starting to merge the subarrays

merge → n steps

merge → n steps

merge → n steps

$\log_2 8$

3

$\log_2 8$

3

$O(n \log n)$

$2 \log_2 n \cdot n$

$2 \log_2 8 \cdot 8$

$n/2 \{2, 4, 5, 7\}$ $\quad$ $n/2 \{1, 2, 3, 6\}$

$n \{1, 2, 2, 3, 4, 5, 6, 7\}$

1. comparing
2. making the element of bigger array
3. moving the pointer

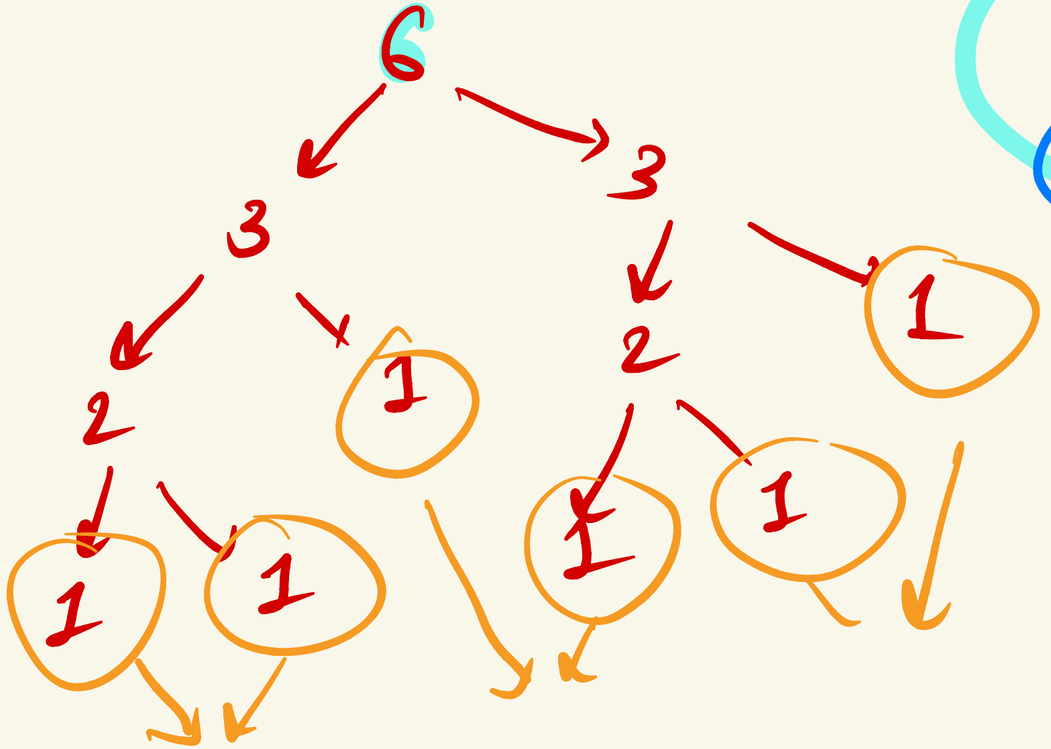
$\Big\}\ n$

$\downarrow$ n times

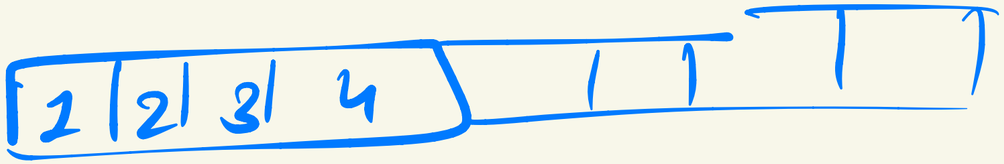
$n \cdot \log_2 n$

merging

$O(n)$ $\leftarrow$ 3n steps



dividing



$O(n)$

for (i=0 , i<n ; i++) { $\rightarrow n/2$

for ($\quad$) { $\rightarrow n/2$

O(N LOG N)

Merge sort is an efficient algorithm, because it halves the size of the subarray at each step.

The recursion consists of O(log n) levels, and processing each level takes O(n) time.

Merging the subarrays array [a...k] and array [k +1...b] is possible in linear time, because they are already sorted.

H.W. - Try to write code for Merge Sort and prove that its not possible to sort an array based on comparing elements faster than O(n log n)!!

NOW HERE COMES A REALLY INTERESTING PART. WHAT IF SORTING IS JUST A SMALL PART OF A QUESTION? ARE WE GONNA SPENT OUR TIME TO JUST SORT THE ARRAY?

SORT FUNCTION

The C++ standard library contains the function sort that can be easily used **for sorting arrays and other data structures.**

There are many benefits in using a library function. First, it saves time because there is no need to implement the function. Second, the library implementation is certainly correct and efficient

```
int main()
{
    int n = 7; // array size
    int a[] = {4, 2, 5, 3, 5, 8, 3};
    sort(a, a + n);
}
```

code a proper sort yourself
std::sort()

mathew ✅
@mathew@mastodon.social
I came up with a single pass O(n) sort algorithm I call StalinSort. You iterate down the list of elements checking if they're in order. Any element which is out of order is eliminated. At the end you have a sorted list.
2018/10/26 04:20:16

Interviewer : Write a program to sort the array
Me :
int v[]={1,4,5,9,7,8} ;
sort(v,v+6);
*Me
*Interviewer

By default sort function sorts in ascending order. But it is also possible to give an external comparison function to the sort function as a callback function.

sort(first iterator, last iterator, comparison function)

For example, the following comparison function comp sorts strings primarily by length and secondarily by alphabetical order:

```
#include <bits/stdc++.h>
using namespace std;
bool comp(string a, string b)
{
    if (a.size() != b.size())
        return a.size() < b.size();
    return a < b;
}
int main()
{
    string arr[] = {"Abhi", "Ayush", "Aditya", "Harsh", "Punya", "Abhishek"};
    sort(arr, arr + 6, comp);
    for (int i = 0; i < 7; i++)
        cout << arr[i] << " ";
}
// Output: Abhi Ayush Harsh Punya Aditya Abhishek
```

1. string size >
2. normal lexicographic --

4 5 5 5 6 8

cout << y;

cout << *y;

dereencing

#0x.....

y

x

BINARY SEARCH

pointer

memory → &x

0x.....

x

pointer

int x;

int* y = &x

Binary Search Algorithm is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

Note: These functions assume that the Array is sorted.

- lower_bound returns a pointer to the first array element whose value is atleast x. >= x
- upper_bound returns a pointer to the first array element whose value is larger than x. > x
- equal_range returns both above pointers.

We already have a session planned for Binary Search, where we will be discussing the implementation and problem solving through binary search. Here, we are going to discuss the inbuilt functions for binary search in the STL Library.

If there is no such element, the pointer points to the element after the last array element.

For example, the following code finds out whether an array contains an element with value x:

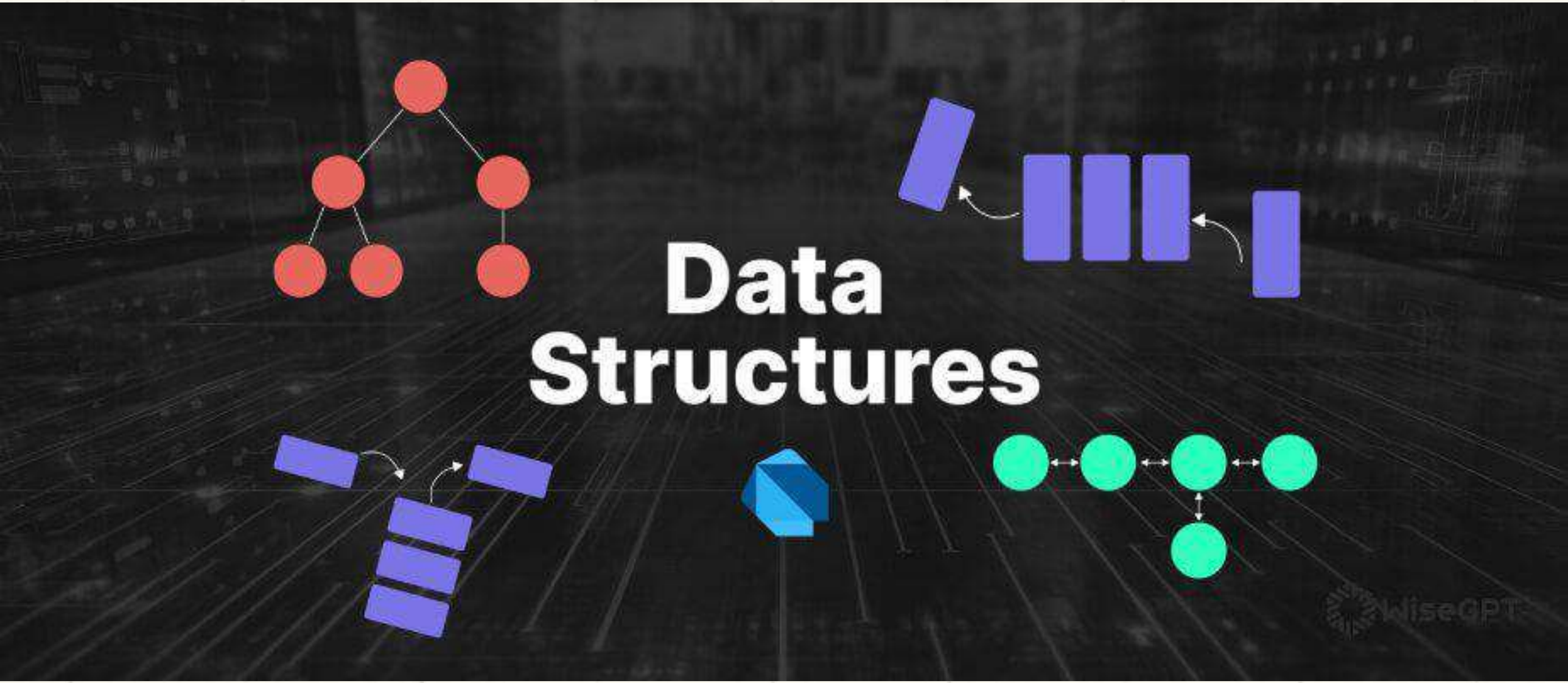
```
// auto keyword automatically determines the data type of the variable
auto k = lower_bound(array, array + n, x) - array;
if (k < n && array[k] == x)
{
    cout << k << "\n";
}
```

arr = {1, 2, 3, 3, 3, 4, 5, 7, 11, 13}
0 1 2 3 4 5 6 7 8 9

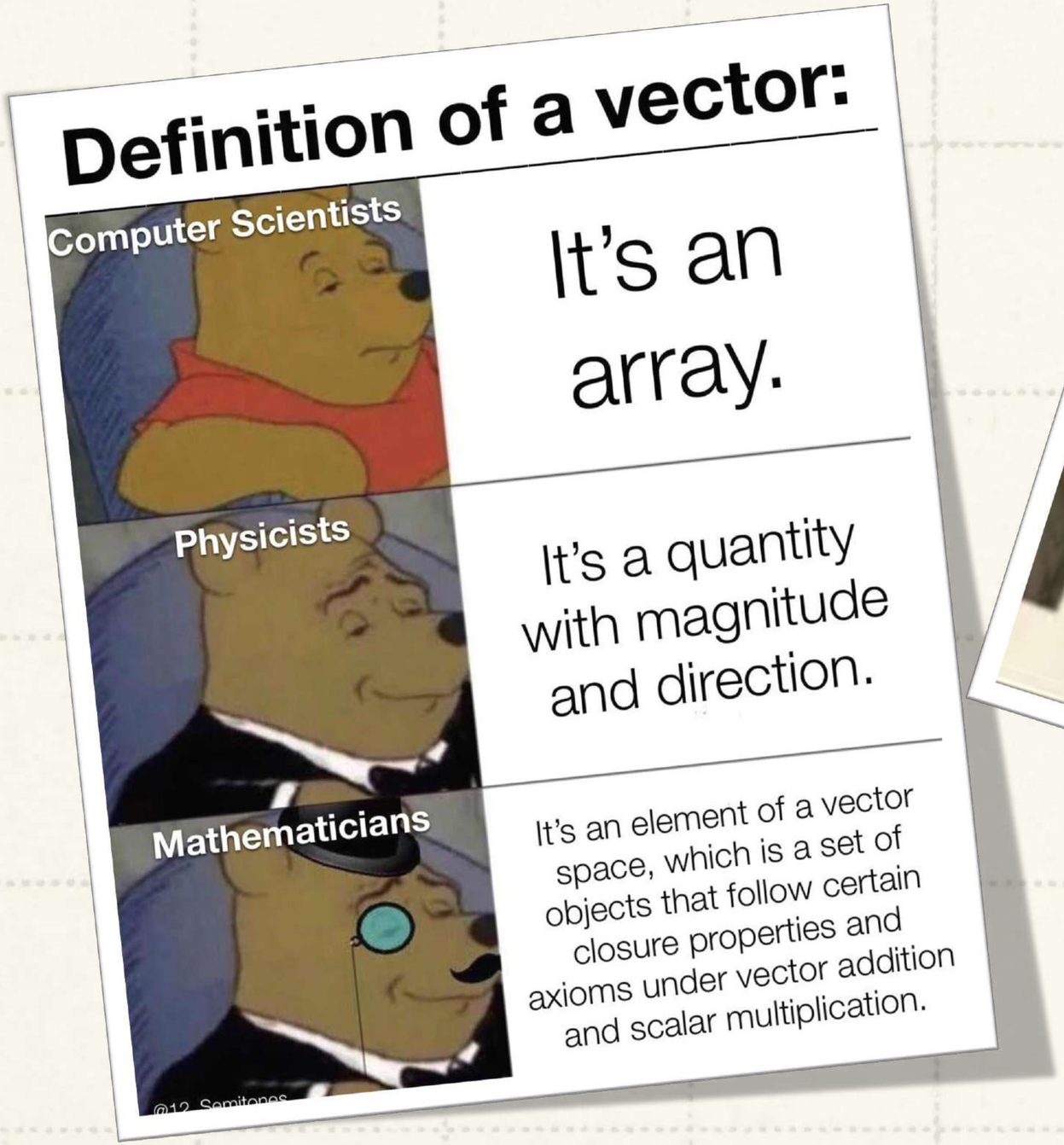
	$x = 3$,	$x = 6$
cout << *lower_bound(arr, arr + 10, x)	3	7
cout << *upper_bound(arr, arr + 10, x)	4	7

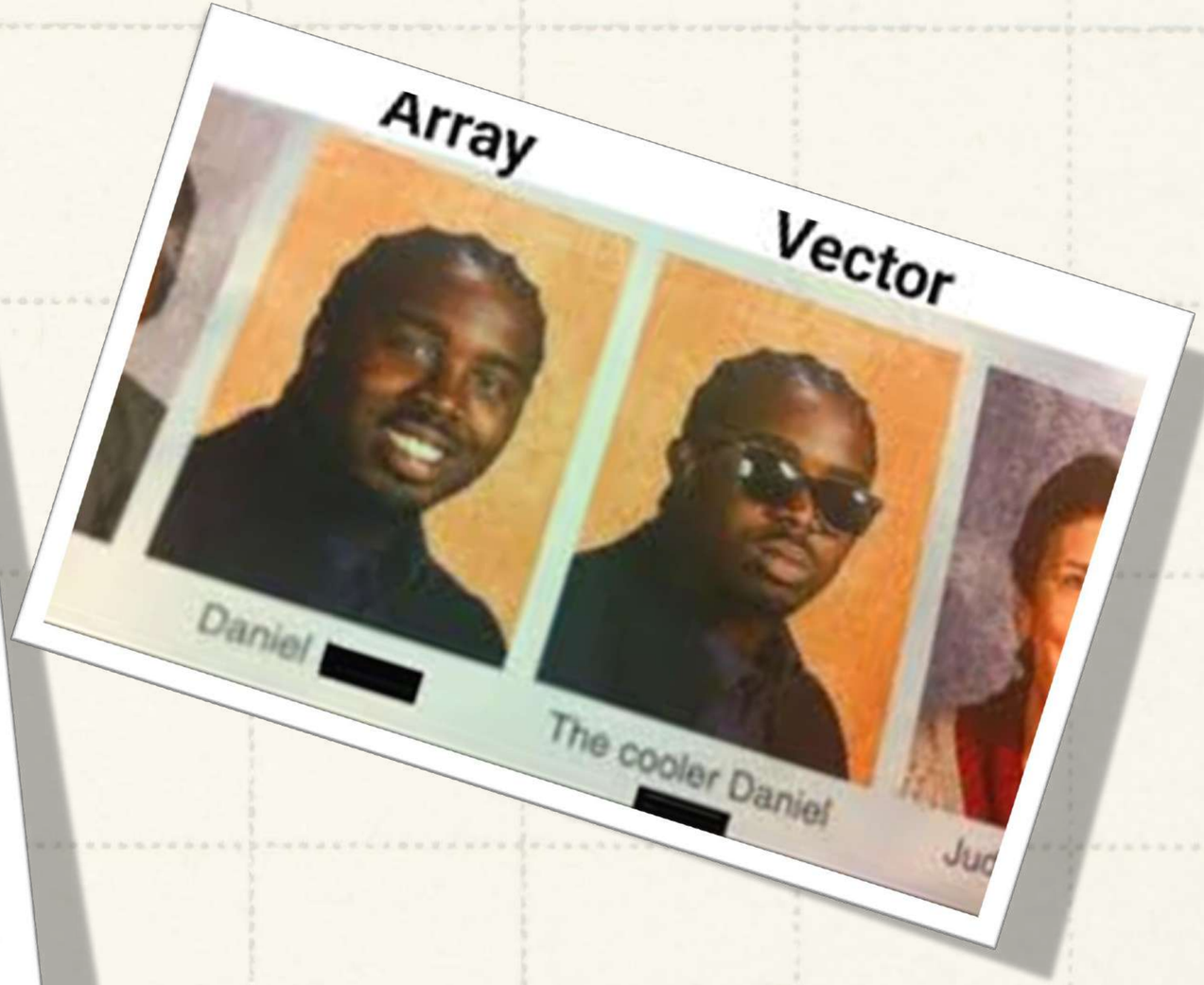
To know more about binary search visit

https://www.geeksforgeeks.org/binary-search/


Data
Structures
WiseGPT

1 VECTORS


Definition of a vector:
Computer Scientists
It's an array.
Physicists
It's a quantity with magnitude and direction.
Mathematicians
It's an element of a vector space, which is a set of objects that follow certain closure properties and axioms under vector addition and scalar multiplication.
©12 Semitones


Array
Vector
Daniel [redacted]
The cooler Daniel [redacted]

{0, 1, 2, 3}

VECTORS

A vector in C++ is a dynamic array provided by the Standard Template Library (STL). It can change its size automatically when elements are added or removed.

Advantages of using vector:

- Allows random access to elements.
- Automatically manages the size of the array.
- Provides many useful member functions for various operations.

CREATING A VECTOR

vector <int> v;

- It declares a vector v of data type int, just like an empty integer array.
- One can define vector for any data-type.

vector<int> v = {2, 4, 5, 2, 1};

- It declares a vector v of 5 elements.

int arr[10]

vector<int> v (10);

- It creates an empty vector of size 10 with all values initialised to 0.

vector<int> v (10 , 5);

- It creates a empty vector of size 10 with all values initialised to 5.

COMMON FUNCTIONS OF VECTOR

v.size() → size

- It returns the size of a vector.

access element index with 5

v.at(5)

v[5]

- You can access, use and edit any element like an ordinary array. For example you can print the vector like this:

```
for (int i = 0; i < v.size(); i++)
{
    cout << v[i] << " ";
}
```

v.push_back(5)

v = {0, 1, 2, 3, 4}

v.push_back(5)

- It pushes elements to the end of a vector. Its average time complexity is O(1).

v.pop_back( )

{0, 1, 2, 3, 4, 5}

v.pop_back( )

- It deletes the last element of a vector. Its average time complexity is O(1).

v. back( )

{0, 1, 2, 3, 4}

- Using it you can access the last element of a vector.

v. clear( )

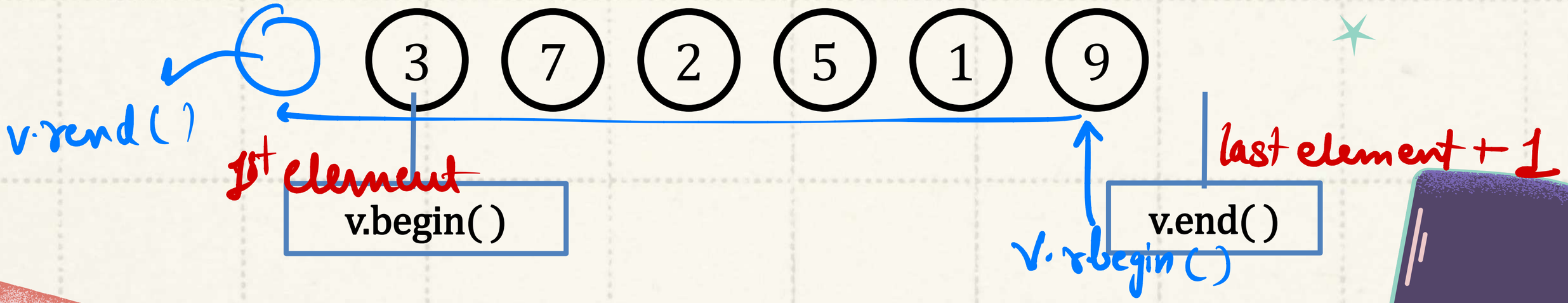
- This deletes all elements of the array making it of size 0.

ITERATORS IN VECTORS

An iterator is a variable that points to an element in a data structure as many functions operate with the iterators.

The often used iterators begin and end define a range that contains all elements in a data structure. The iterator begin points to the first element in the data structure, and the iterator end points to the position after the last element.

The range defined is a half-open range $[v.begin(), v.end())$



There also exist reverse iterators rbegin and rend, rbegin points to the last element of the vector and rend points to the element before the first element of the vector

FUNCTIONS USING ITERATORS

sort(v.begin( ) , v.end( ))

- Sorts the array in ascending order in O(n log n) time complexity

sort (v.rbegin(), v.rend()) → descending order

reverse(v.begin( ) , v.end( ))

- Reverses the array in O(n) time complexity.

random_shuffle(v.begin( ) , v.end( ))

- Shuffles the array in O(n) time complexity.

Can be also used for ordinary arrays as:

- sort(a , a+n)
- reverse(a , a+n)
- random_shuffle(a , a+n)

sort (v.begin(), v.end()
- - - -)

max_element(v.begin( ) , v.end( ))

- Returns the pointer to maximum element in the vector in O(n) time complexity.

min_element(v.begin( ) , v.end( ))

- Returns the pointer to minimum element in the vector in O(n) time complexity.

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<int> v = {14, 16, 19, 34, 56, 78, 89, 90, 100};
    cout << *max_element(v.begin(), v.end()) << endl;
    cout << *min_element(v.begin(), v.end()) << endl;

    int arr[] = {14, 16, 19, 34, 56, 78, 89, 90, 100};
    cout << *max_element(arr, arr + 9) << endl;
    cout << *min_element(arr, arr + 9) << endl;
}
```

v. erase(position)

- For deletion at specific position in O(n) time complexity

v. erase(starting_position, end_position)

- For deletion in range in O(n) time complexity

```
vector<int> v;
v = {9, 3, 6, 1, 2, 5, 8, 7, 4};
v.erase(v.begin() + 2);
//v = {9, 3, 1, 2, 5, 8, 7, 4}
v.erase(v.begin() + 2, v.begin() + 5);
// first is inclusive, second is exclusive
//v = {9, 3, 8, 7, 4}
```

v. insert(position , value)

- For insertion at specific position in O(n) time complexity

```
// Inserting the value 100 at position 3(0-based
// indexing) in the vector
vector_name.insert(vector_name.begin() + 3, 100);
```

There are many more functions in vector class: resize, swap, unique function. All can't be covered and remembered in a single session and not all are useful but the code for all of them will be given in the description, try to read them so you at least know that such a function exists. You could still hard code most of them.

BONUS H.W.

If you want to clear all your concepts of dynamic memory allocation, oops programming and c++, try to create your own vector library and try to imitate maximum features as you can.
You could also try this homework for the string class that we are going to learn.
This won't be helpful for C.P. but overall a great experience.

Useful Resources:

- https://www.geeksforgeeks.org/how-to-implement-our-own-vector-class-in-c/
- https://www2.cs.sfu.ca/CourseCentral/125/tjd/vector_example.html
- https://medium.com/@oyemayanq/how-to-create-a-custom-string-class-in-c-2062292c5edd

PAIR

pair<int, int> p;

pair<type1, type2> pair_name;

cout << p.first << p.second;

- For example: pair<int, int> p1; pair <int, string> p2;

pair<int, string> p(1, “Hello”);

- It initialises the pair with given values.

make_pair(1, “Hello”)

p = makepair(1, 2)

- It makes a pair.

p.first, p.second

- It can be used to access elements of a pair.

TUPLE

tuple<type1, type2, ..., typeN> tuple_name;

- For example: tuple<int, int, string> p1;

tuple<int, string, float> t(1, “Hello”, 3.14);

- It initialises the pair with given values.

make_tuple(1, “Hi”,2)

- It makes a pair.

g ~~G~~et<i> (t)

- It can be used to access i^{th} element of a tuple.

```
// defining a pair
cout << "Defining a pair: \n";
pair<int, int> p1 = make_pair(1, 12);
cout << p1.first << " " << p1.second << endl;
pair<int, string> p2(2, "Hello");
cout << p2.first << " " << p2.second << endl;

// defining a tuple
cout << "\nDefining a tuple: \n";
tuple<int, int, int> t1 = make_tuple(1, 2, 3);
cout << get<0>(t1) << " " << get<1>(t1) << " " << get<2>(t1) << endl;
tuple<int, string, string> t2(4, "Hello", "World");
cout << get<0>(t2) << " " << get<1>(t2) << " " << get<2>(t2) << endl;
```

VECTOR OF PAIRS AND TUPLES

vector<pair<int,int>> vp;

- It creates a vector of pairs.

vector<tuple<int,int,int>> vt;

- Creates a vector of tuples.

vector<vector<int>> vv;

- nested vectors, each element of vector is a vector itself, a kind of matrix.

All the above 3 can be sorted using the sort function, the sort proceeds with the first element then second and so on.

2 STRINGS

STRINGS

The string structure is also a dynamic array that can be used almost like a vector.

- In addition, there is special syntax for strings that is not available in other data structures. Strings can be combined using the + symbol.
- The function s.substr(k, x) returns the substring that begins at position k and has length x, and
- the function s.find(t) finds the position of the first occurrence of a substring t.

There are many more functions in code link in description, not necessary to learn them all.

```
string s1 = "Hello";
string s2 = "World";
string s3 = s1 + s2; //"HelloWorld"
cout << s3 << endl;
cout << s3.size() << endl; // 10
cout << s3[0] << endl;     // H
cout << s3[1] << endl;     // e
cout << s3[2] << endl;     // l

// substring of a string
cout << "\nSubstring of a string: \n";
cout << s3.substr(0, 5) << endl; // Hello
// 0 is the starting index and 5 is the length of the substring
// time complexity of substr is O(n)

// finding a substring in a string
cout << "\nFinding a substring in a string: \n";
cout << s3.find("lo") << endl; // 3
```

0
Hello
5

index of first occurance of substring

3 SET STRUCTURES

SETS

A set is a data structure that maintains a collection of elements. The basic operations of sets are element insertion, search and removal.

The C++ standard library contains two set implementations:

sorted

- The structure set is based on a balanced binary tree and its operations work in O(logn) time.

~~sorted~~

- The structure unordered_set uses hashing, and its operations work in O(1) time on average.

The choice of which set implementation to use is often a matter of taste. The benefit of the set structure is that it maintains the order of the elements and provides functions that are not available in unordered_set. On the other hand, unordered_set can be more efficient.

SET

→ stored in sorted-order

O(log n)

- The set operations insert, erase, count takes O(log n) time.
- The set is sorted and implemented using balanced binary trees.

```
set<int> s;
s.insert(1);
s.insert(2);
s.insert(3);
s.insert(3); // this will not be inserted as set stores unique elements
// insertion takes O(logn) time
// s = {1, 2, 3}

cout << "\nChecking if 2,4 are present in the set: \n";
cout << "4: " << s.count(4) << endl; // returns 0 as 4 is not present in the set
cout << "2: " << s.count(2) << endl; // returns 1 as 2 is present in the set
// count takes O(logn) time

cout << "\nErasing 2 from the set: \n";
s.erase(2);
// s = {1, 3}
// erasing an element also takes O(logn) time
```

UNORDERED_SET

→ On avg. take O(1)

- The unordered_set operations insert, erase, count takes O(n) time.
- It is implemented using hashing.

```
unordered_set<int> us;
us.insert(4);
us.insert(2);
us.insert(3);
us.insert(3); // this will not be inserted as unordered set stores unique elements
us.insert(5);
us.insert(1);
// insertion takes O(1) time
// us = {4, 2, 3, 5, 1};

// checking if an element is present in the unordered set
cout << "\nChecking if 2,4 are present in the unordered set: \n";
cout << "4: " << us.count(4) << endl; // returns 1 as 4 is present in the unordered set
cout << "2: " << us.count(2) << endl; // returns 1 as 2 is present in the unordered set
// count takes O(1) time

// erasing an element from the unordered set
cout << "\nErasing 2 from the unordered set: \n";
us.erase(2);
// erasing an element takes O(1) time
// us = {4, 3, 5, 1}
```

→ O (log N)

MULTISET

Multiset is similar to set but it can store duplicate elements.

It also has similar operations. Checkout the erase function in multiset.

Similarly, unordered multiset also exists.

↓ O(1)

```
// multiset
// multiset is similar to set but it can store duplicate elements
multiset<int> ms;
ms.insert(4);
ms.insert(2);
ms.insert(3);
ms.insert(3); // this will be inserted as multiset stores duplicate elements
ms.insert(5);
ms.insert(5);
// ms = {2, 3, 3, 4, 5, 5}
//  insertion takes O(logn) time

// checking if an element is present in the multiset
cout << "\nChecking if 2,3 are present in the multiset: \n";
cout << "3: " << ms.count(3) << endl; // returns 2 as 3 is present twice in the multiset
cout << "2: " << ms.count(2) << endl; // returns 1 as 2 is present once in the multiset
```

{2, 3, 3, 4, 5, 5}

```
// erasing an element from the multiset
cout << "\nErasing 3 from the multiset: \n";
ms.erase(3); // this will erase all the occurrences of 3
// erasing an element takes O(logn) time
// ms = {2, 4, 5, 5}
```

{2, 4, 5, 5}

```
// to erase only one occurrence of an element from the multiset
cout << "\nErasing only one occurrence of 5 from the multiset: \n";
ms.erase(ms.find(5)); // this will erase only one occurrence of 5
// ms = {2, 4, 5}
```

→ pointer to first element i.e., 5

(vector<int> iterator it = v.begin( ))

ITERATORS IN SETS

Iterators are often used to access elements of a set.

The following code creates an iterator it that points to the smallest element in a set: set<int>::iterator it = s.begin( )

~~s[0], s[1]~~

```
//  set iterator is used to iterate through the set because we can't access the elements of
the set using the index operator []
set<int> s1;
s1 = {21, 12, 33, 24, 15};
set<int>::iterator it = s.begin();
cout << "\n\nFirst Element : " << *it << endl; // 12
cout << "elements of the set using the iterator: \n";
// iterator can be incremented using ++ operator or it can be decremented using -- operator
// iterator can be dereferenced using * operator
// printing the elements of the set using the iterator
for (auto it = s1.begin(); it != s1.end(); it++)
{
    cout << *it << " ";
}
```

for loop

s.find(x)

if it doesn't find

s.end()

- It returns an iterator that points to an element whose value is x. However, if the set does not contain x, the iterator will be end.

s.lower_bound(x)

- It returns an iterator to the smallest element in the set whose value is at least x

s.upper_bound(x)

- It returns an iterator to the smallest element in the set whose value is larger than x.

In both functions, if such an element does not exist, the return value is end. These functions are not supported by the unordered_set structure which does not maintain the order of the elements.

4

MAP STRUCTURES

MAPS — Sim. to dict. in python

A map is a generalized array that consists of key-value-pairs. While the keys in an ordinary array are always the consecutive integers 0,1,...,n-1, where n is the size of the array, the keys in a map can be of any data type and they do not have to be consecutive values.

The C++ standard library contains two map implementations that correspond to the set implementations:

- The structure map is based on a balanced binary treeand accessing elements takes O(logn) time.

sorted

- The structure unordered_map uses hashing and accessing elements takes O(1) time on average.

not sorted

Map can be initialized as

key

map<type1, type2> m;

elements

Elements of map can be accessed using m[key]

To check if a key exists m.count(key)

If trying to access a key not in the map, it will be initialized to default value.

Unordered mam can be declared in a similar way using unordered_map instead of map.

It is similar to a dictionary in python

```
map<string, int> m;
m["Ayush"] = 21;
m["Punya"] = 23;
m["Abhi"] = 31;
m["Aditya"] = 45;
m["Harsh"] = 15;
// the map will be sorted according to the keys
// all operations are O(logn) in map
cout << m["Ayush"] << endl; // 21

// to check if a key is present in the map
cout << "Count of Ankit : " << m.count("Ankit") << endl; // 0

// if a key is not present in the map, it will be inserted with a default value 0
cout << "Default value : " << m["Ankit"] << endl; // 0

// printing the map
cout << "\nPrinting the map: \n";
for (auto x : m)
{
    cout << x.first << " " << x.second << endl;
}

// erasing a key from the map
cout << "\nErasing Ayush key from the map: \n";
m.erase("Ayush");
```

key

value

m["Ankit"] = 0

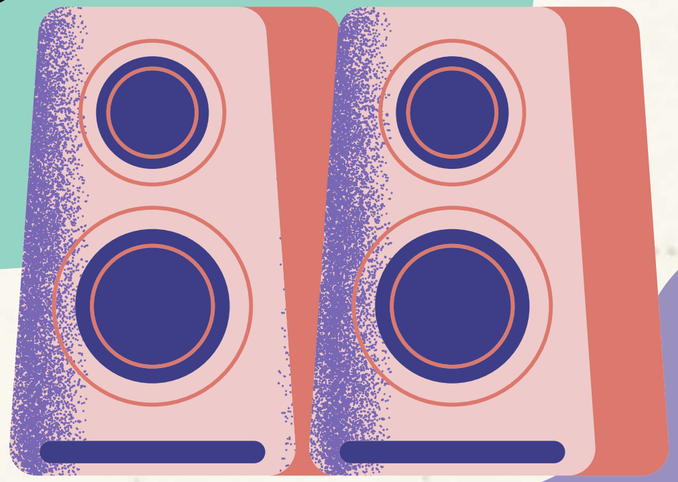
if key exist or not

m.count(key)

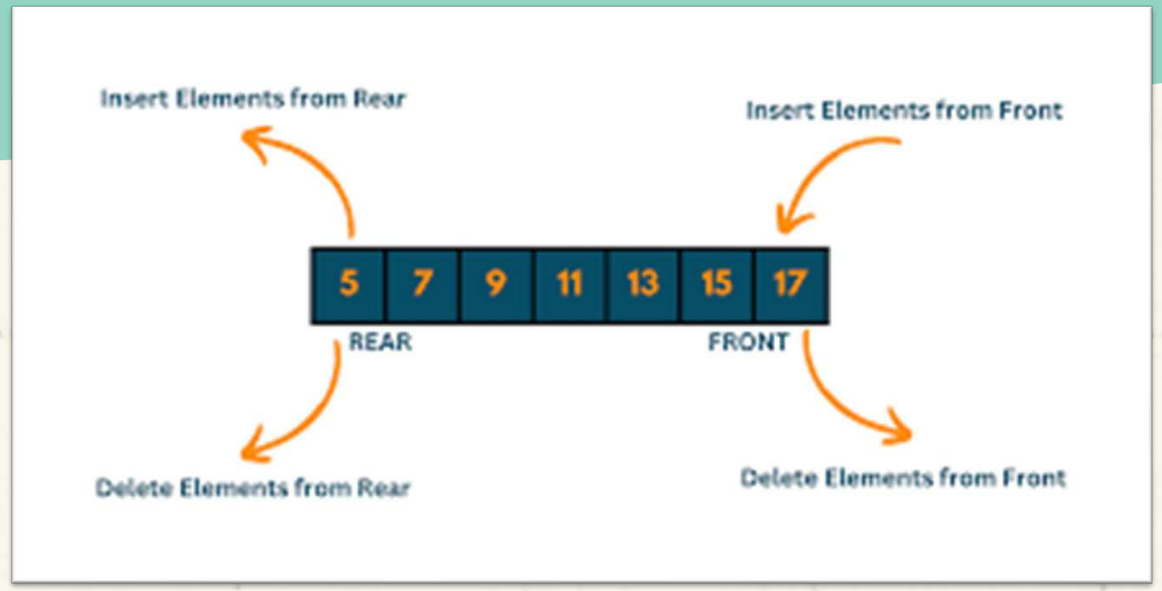
0 key ✗

1 key ✓

MORE DATA STRUCTURES



1 DEQUE


Insert Elements from Rear
Insert Elements from Front
5
7
9
11
13
15
17
REAR
FRONT
Delete Elements from Rear
Delete Elements from Front

DEQUE

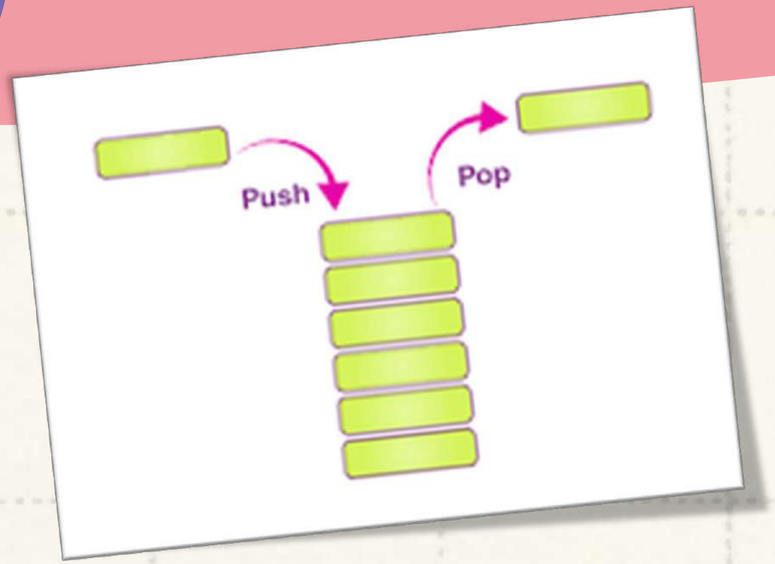
A deque is a dynamic array whose size can be efficiently changed at both ends of the array.

Like a vector, a deque provides the functions push_back and pop_back.

But it also includes the functions push_front and pop_front which are not available in a vector.

```
// deque
deque<int> dq;
dq.push_back(1);  // [1]
dq.push_back(2);  // [1,2]
dq.push_front(3); // [3,1,2]
dq.push_front(4); // [4,3,1,2]
cout << "\nDeque: \n";
for (auto x : dq)
{
    cout << x << " ";
}
cout << "\n";
dq.pop_back();  // [4,3,1]
dq.pop_front(); // [3,1]
```

2 STACK



STACK

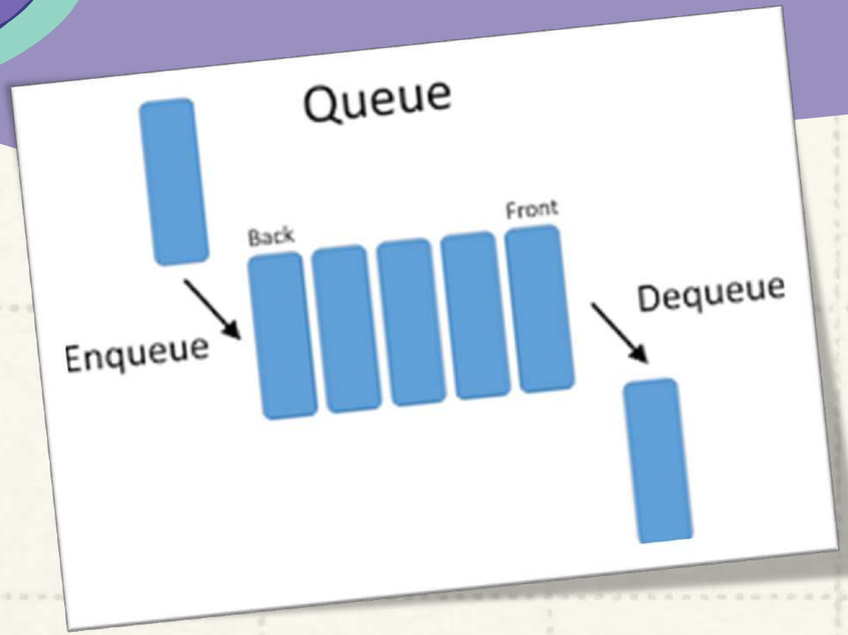
A stack is a data structure that provides two O(1) time operations:

- Adding an element to the top.
- Removing an element from the top.

It is only possible to access the top element of a stack.

```
// stack
stack<int> s;
s.push(1);
s.push(2);
s.push(3); // [1,2,3]
cout << "\nStack: \n";
while (!s.empty())
{
    cout << s.top() << " "; // s.top() returns the top element of the stack
    s.pop();                // s.pop() removes the top element of the stack
}
// Output: 3 2 1
cout << "\n";
```

3 QUEUE



QUEUE

A queue also provides two O(1) time operations :

- Adding an element to the end.
- Removing the first element from the queue.

It is only possible to access the first and last element of a queue.

```
// queue
queue<int> q;
q.push(1);
q.push(2);
q.push(3); // [1,2,3]
cout << "\nQueue: \n";
while (!q.empty())
{
    cout << q.front() << " "; // q.front() returns the front element of the queue
    q.pop();                  // q.pop() removes the front element of the queue
}
// Output: 1 2 3
```

4 PRIORITY QUEUE

PRIORITY QUEUE

A priority queue maintains a set of elements. The supported operations are insertion and, depending on the type of the queue, retrieval and removal of either the minimum or maximum element.

Insertion and removal take O(logn) time and retrieval takes O(1) time.

```
priority_queue<int> pq;
```

A priority queue is usually implemented using a heap structure that is much simpler than a balanced binary tree used in an ordered set.

By default, the elements in a C++ priority queue are sorted in decreasing order, and it is possible to find and remove the largest element in the queue.

If we want to create a priority queue that supports finding and removing the smallest element, we can do it as follows:

```
priority_queue<int,vector<int>,greater<int>> pq;
```

```
// priority queue
priority_queue<int> pq;
pq.push(3); // [3]
pq.push(1); // [3,1]
pq.push(5); // [5,3,1]
pq.push(4); // [5,4,3,1]
cout << "\nPriority Queue: \n";
while (!pq.empty())
{
    cout << pq.top() << " "; // pq.top() returns the top element of the priority queue
    pq.pop();                // pq.pop() removes the top element of the priority queue
}
// Output: 5 4 3 1
cout << "\n";

// priority queue with custom comparator
priority_queue<int, vector<int>, greater<int>> pq1; // minumum element at the top
```

5 BITSET

BITSET

A bitset is an array whose each value is either 0 or 1.

```
bitset<10> b;
```

- It creates a bitset of 10 elements. The size of bitset is a compile time constant.

```
b[0]
```

- Used to access bits, note that in bitset indexing is from right to left.

```
bitset<10> s(string("0010011010"));
```

- It creates bitset from the string.

```
s.count( );
```

- It returns the number of ones.

Bit operations

- Operations like b1 & b2, b1 | b2, b1 ^ b2, b1 >> 1, b2 << 1 can be done.

```cpp
// Bitset
bitset<5> b; // 5 bit bitset
b[1] = 1;    // 00010
b[2] = 1;    // 00110
// indexing is from right to left
cout << "\nBitset: \n";
cout << b; // 00110
cout << "\n";
// bitset size is fixed at compile time
// string to bitset
string s1 = "10101";
bitset<5> b1(s1);
cout << b1; // 10101
cout << "\n";
// bitset to string
cout << b1.to_string(); // 10101
cout << "\n";
// operations on bitset
bitset<5> b2("10101");
bitset<5> b3("01101");
cout << "\nBitset operations: \n";
cout << "b2 & b3 :" << (b2 & b3) << "\n"; // 00101
cout << "b2 | b3 :" << (b2 | b3) << "\n"; // 11101
cout << "b2 ^ b3 :" << (b2 ^ b3) << "\n"; // 11000
cout << "b2 << 1 :" << (b2 << 1) << "\n"; // 01010
cout << "b2 >> 2 :" << (b2 >> 2) << "\n"; // 00101
```

6 POLICY-BASED DATA STRUCTURES

PBDS

The g++ compiler also supports some data structures that are not part of the C++ standard library. Such structures are called policy-based data structures. To use these structures, the following lines must be added to the code:

```
#include<ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
```

We can define a data structure indexed_set that is like set but can be indexed like an array : `typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update> indexed_set;`

The function `s.find_by_order( index )` returns an iterator to the element at a given position and the function `s.order_of_key( element )` returns the position of a given element, if the element does not appear in the set, we get the position that the element would have in the set.

```cpp
#include<bits/stdc++.h>
using namespace std;

#include<ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update>
indexed_set;

int main()
{
    indexed_set s;
    //same as set<int> s; but with some extra features like finding by order and order of key
    s.insert(1);
    s.insert(2);
    s.insert(4);
    s.insert(8);
    s.insert(16);
    // s = {1, 2, 4, 8, 16}
    cout << *s.find_by_order(1) << endl; // 2
    cout << *s.find_by_order(2) << endl; // 4
    cout << *s.find_by_order(4) << endl; // 16

    cout << s.order_of_key(10) << endl; // 4
    cout << s.order_of_key(5) << endl; // 3
    cout << s.order_of_key(17) << endl; // 5
}
```

NOW, YOU CAN TRY ON A LOT OF INTRODUCTORY PROBLEMS BASED ON WHAT WE LEARNED IN THIS SESSION.

Check out the Introductory Problems, Sorting and Searching Problems from CSES Problem Set and give them a try.

THANKS FOR WATCHING!
HAPPY CODING!!