**Competitive Programming C++ STL by YT-Luv.**

Parts of STL :

1.Container

2.Iterator

3.Algorithm

4.Functors

**1.Containers:**

A. Sequential

- Vector

- Stack

- Queue

- Pair (not a container)

B. Ordered:

- Maps

- multimap

- Set

- Multiset

C. Unordered

- Unordered map

- Unordered set

D. Nested containers

Vector<vector<int>>

Map<int,vector<int>>

Set<pair<int,string>>

Vector<map<int,set<int>>>

**2.Iterators:**

- Point to memory address of containers

- begin(),end()

- vector<int>::iterator it; -> How to write in short?

- Continuity for containers

**3. Algorithms:**

-upper bound

- Lower bound

- sort (comparator)

- max-element

- min-element

-Accumulate

- Reverse

- count

- find

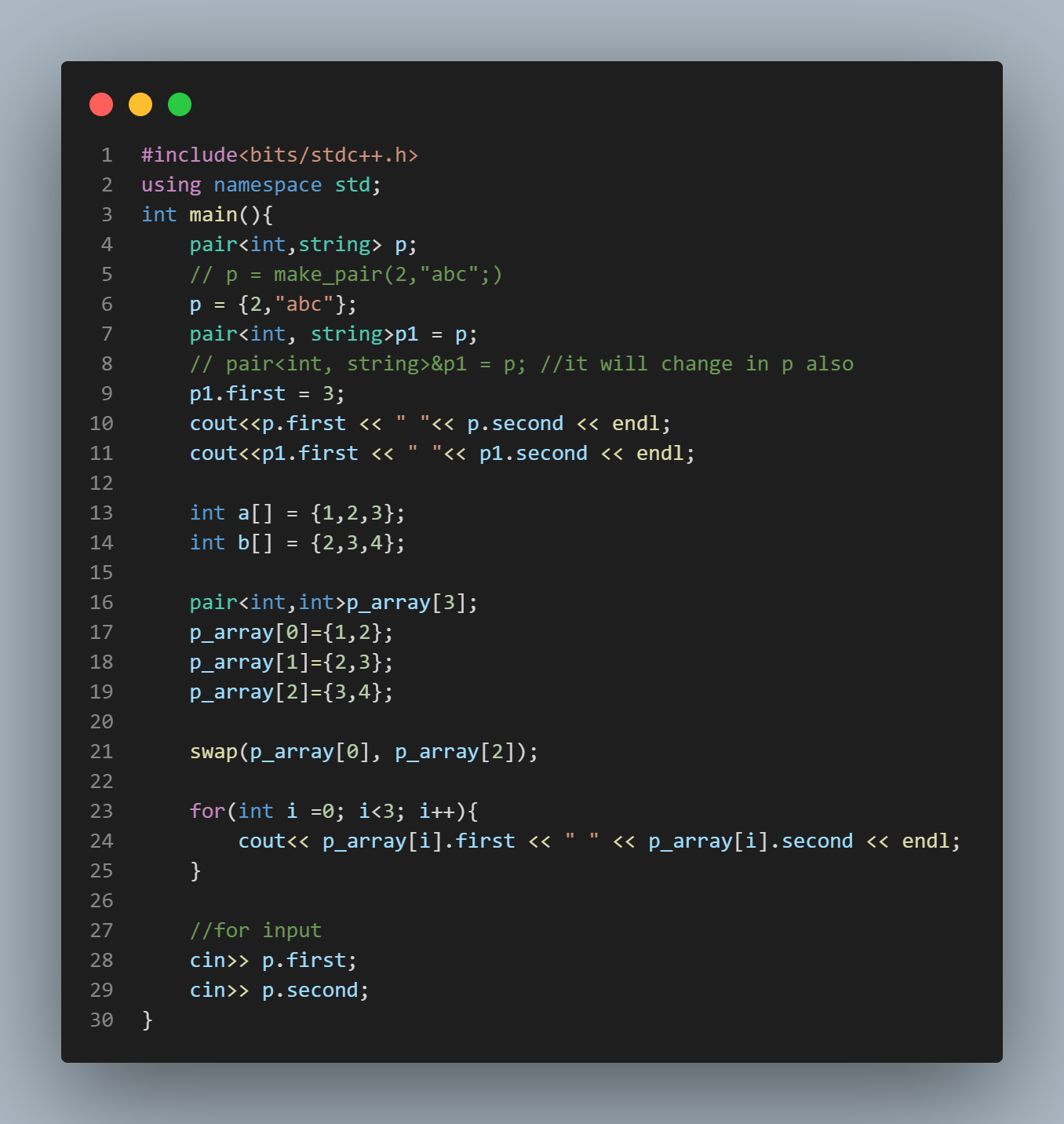
- Next permutation

- Prev-permutation

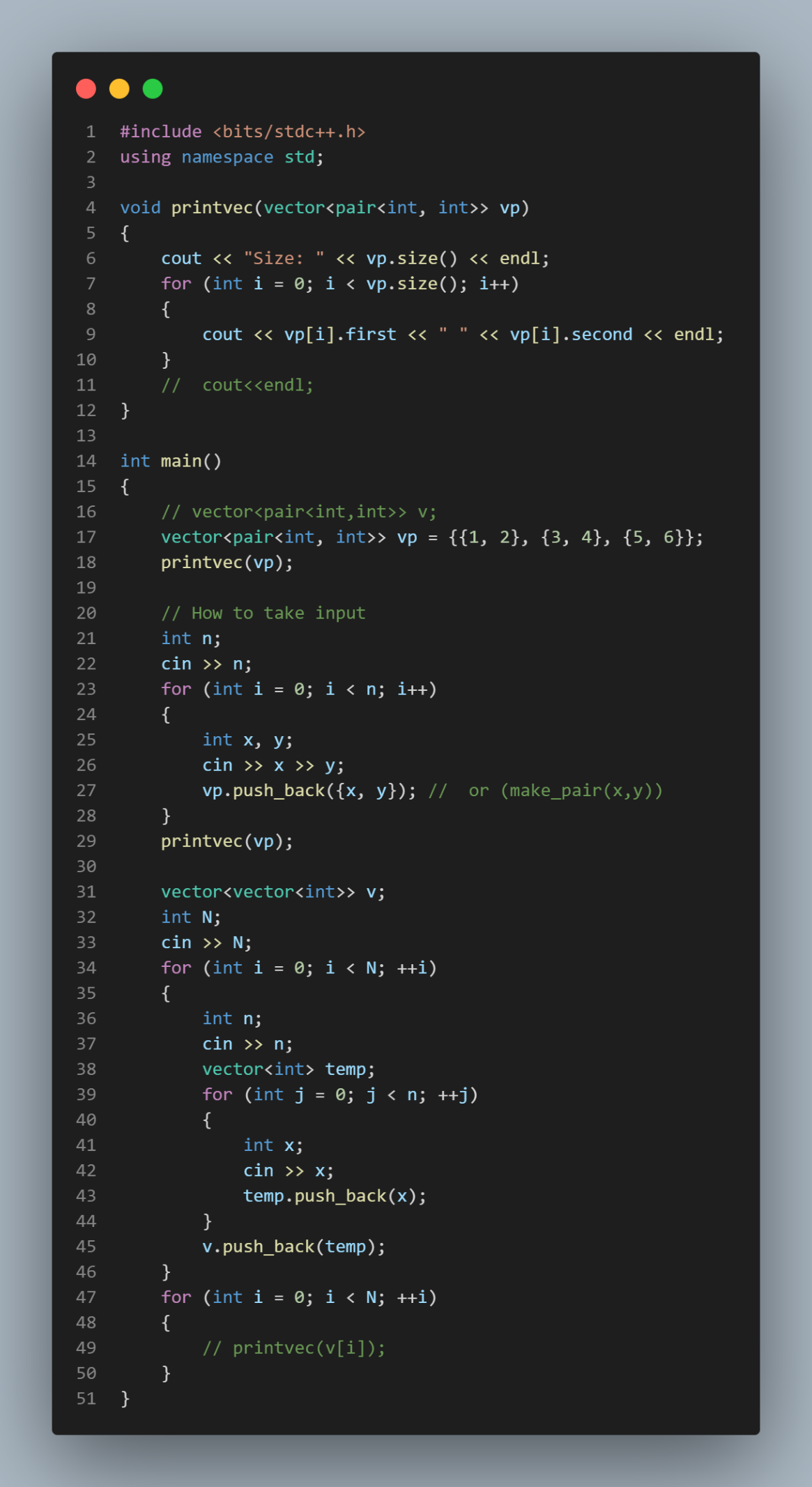
**4. Functors** (no need in competitive programming)

- Classes which can act as functions.

**Pair and Vectors.**



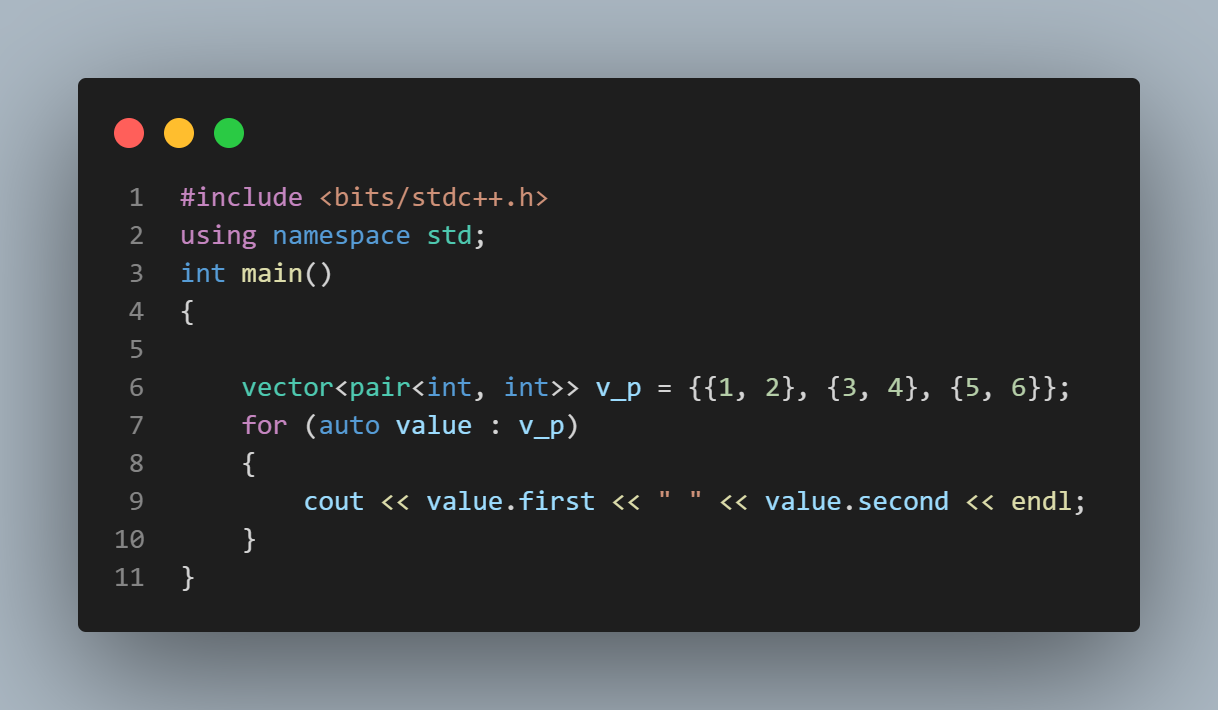
**Nesting in vector:**



**Iterators:**



**Iterator Code in Short:**



**MAPS:**

Store in this form: Key -> value

Eg: int -> string internal implement using Red-Black tree

1. ajay
2. priyanshu
3. rajat

1. takes O(logn) for simple writing m[1]; only.

2. Map<int,string> //keys stores in sorted order.

3. Maps keys are unique no duplicate keys are inserted instead the value changes if u do it.

4. inserting and accessing T.C = O(nlogn).

Eg:

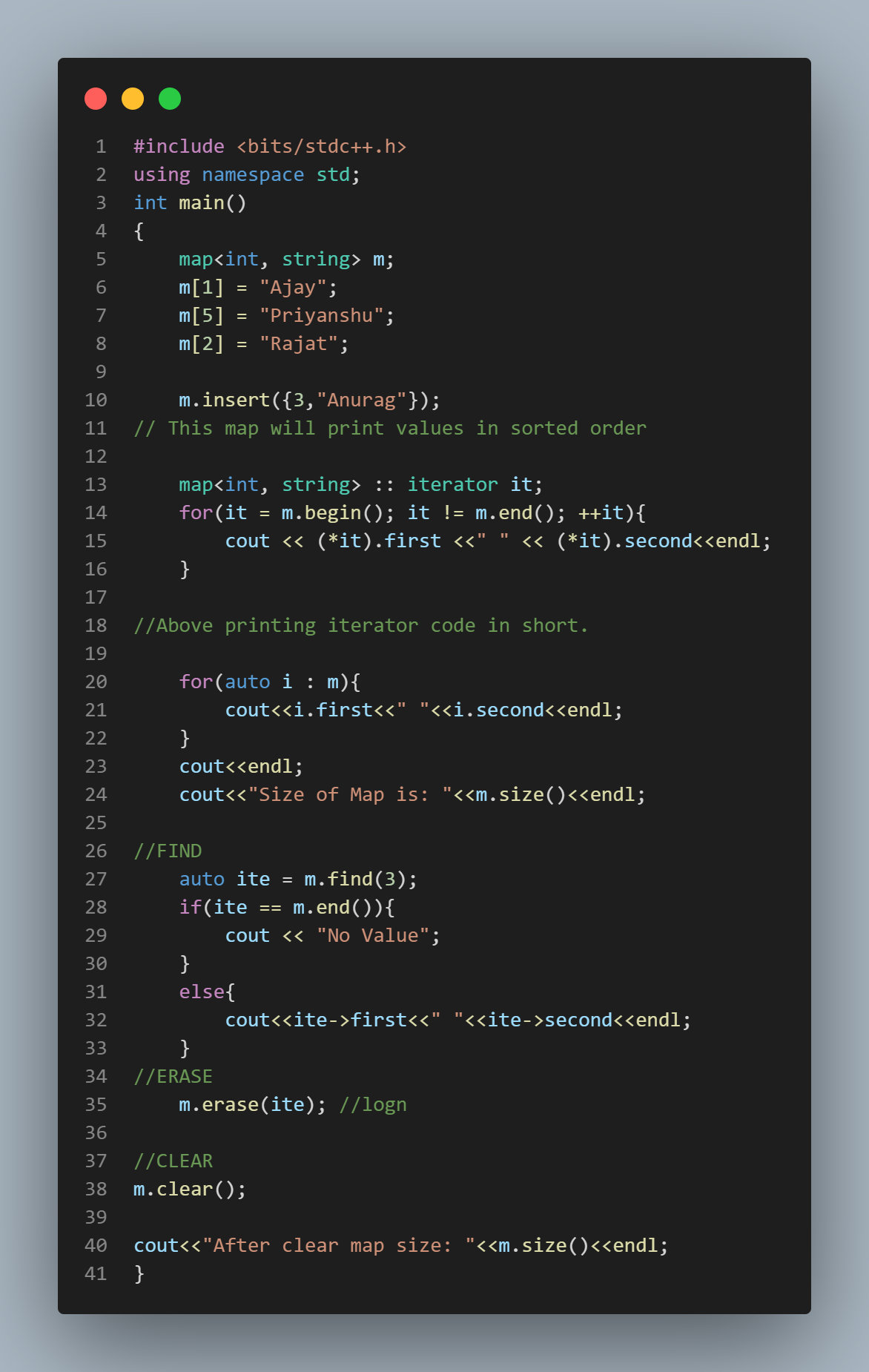
<int, int> implementation => m[1] = 5;

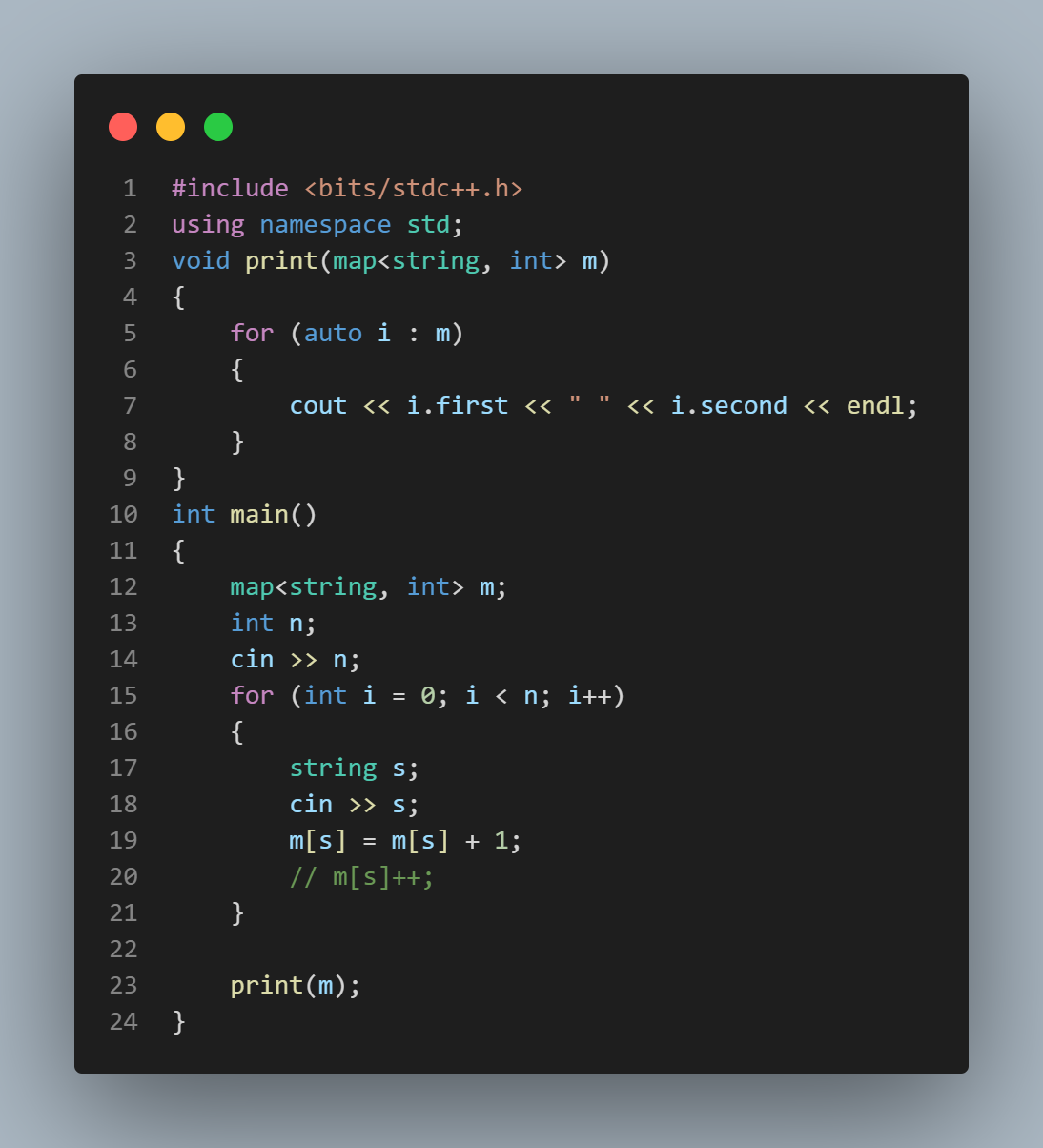
<string, string> implementation => m[ajay] = ”vijay”;

<string, int> implementation => m[abc] = 1;

Taking input:

1. string s;
2. cin>>s;
3. m[s] = 1 or if count calculate then m[s] = m[s] + 1; or m[s]++;

 **QUESTION: Given N strings, print unique strings in lexiographical order with their frequency ?**



**Input:**

5

abc

def

ghi

abc

def

**Output:**

abc 2

def 2

ghi 1

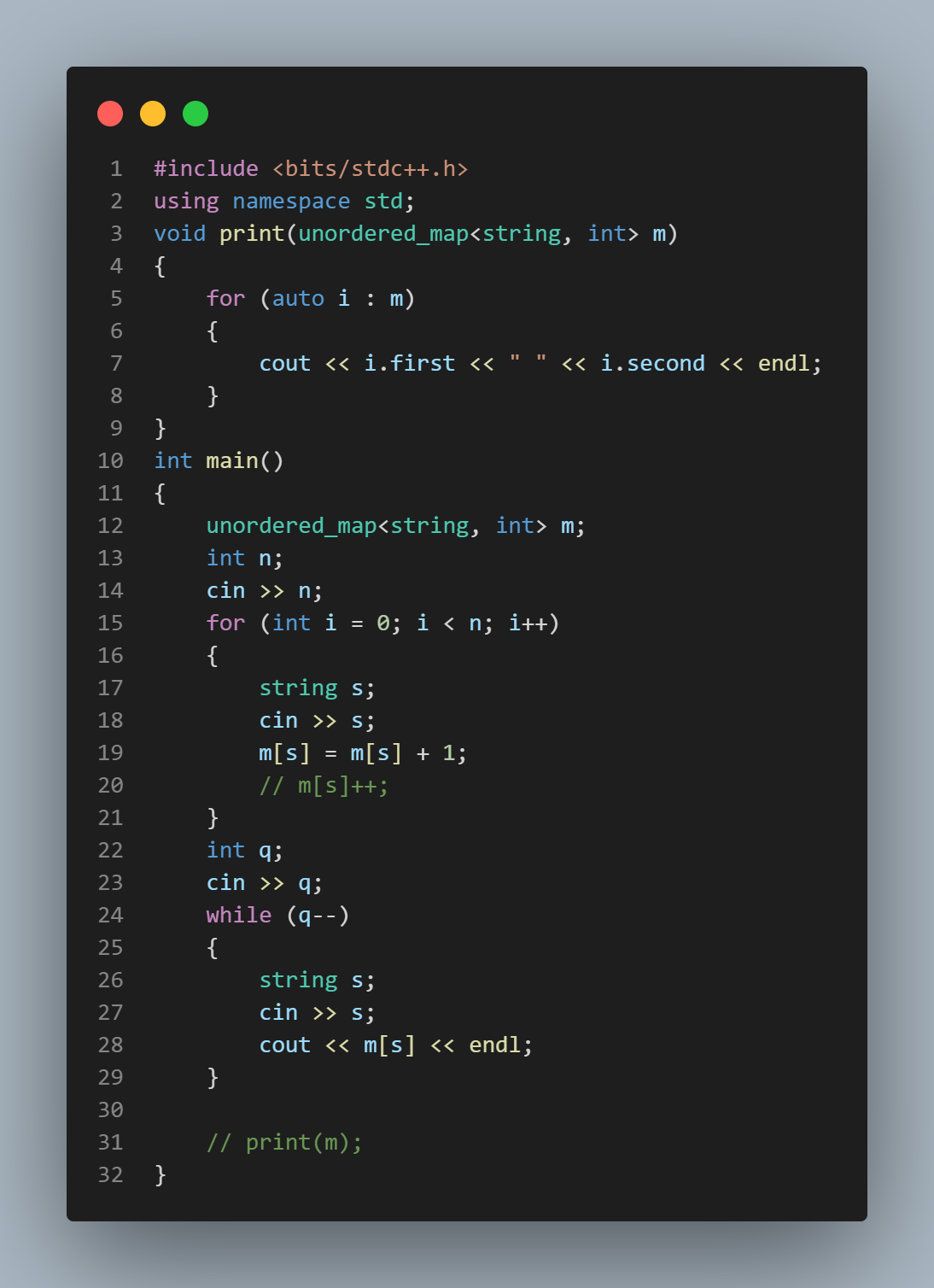
**Unordered Map:**

Differences-

1. Inbuilt implementation – Use hash table
2. Time complexity // O(1) in both insertion and retrival
3. Valid keys datatype // does not support complex data type like pair,set ,pair of pair,vector etc.
4. Keys->values are not sorted

Rest all are same-

QUE:



**SET:**

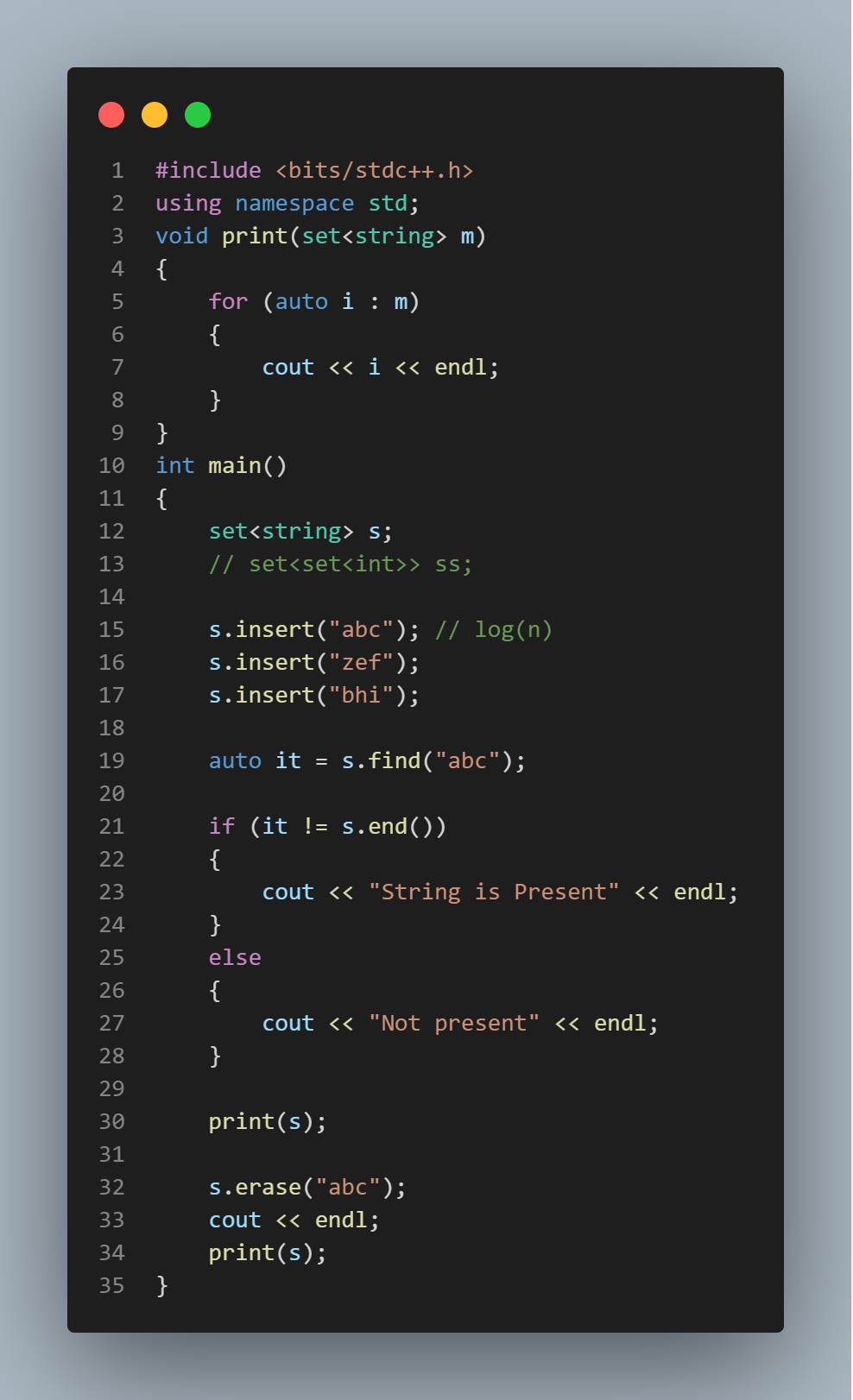
The difference between set and map is in set only ket or one vale is stored

In map => key->value

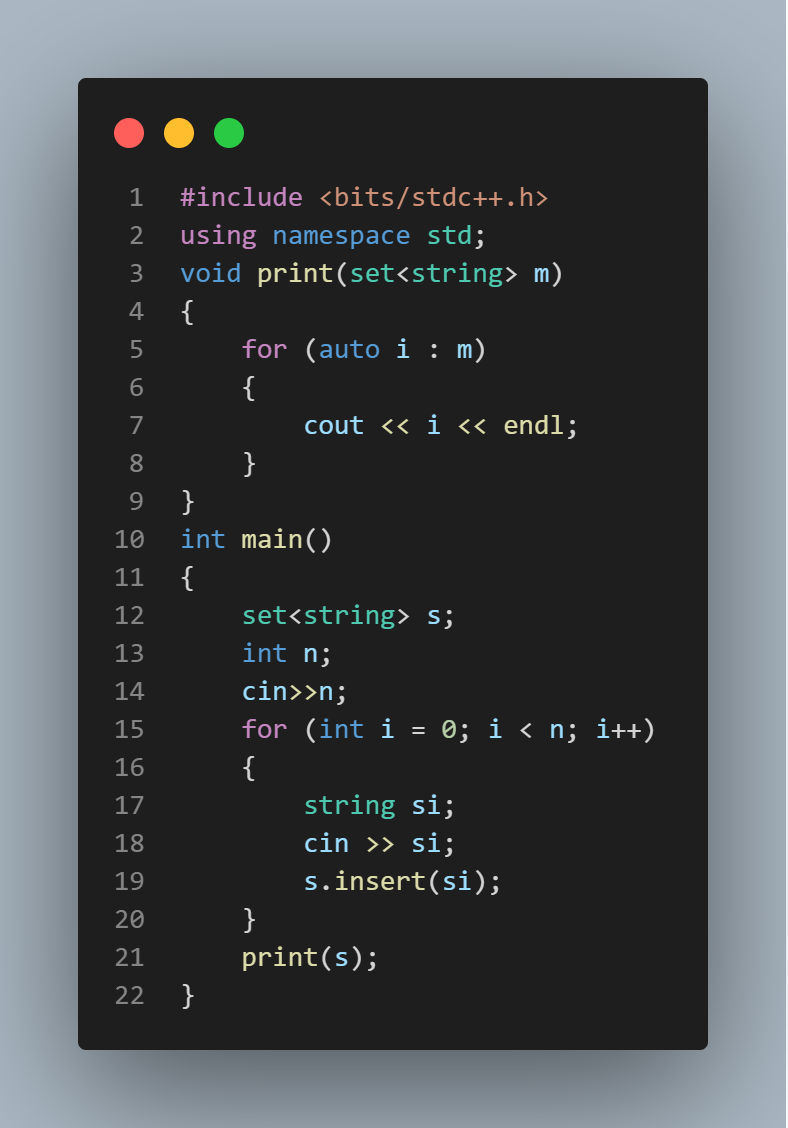
In set => key

Eg: set<string> s;

Output values in sorted order.



Que:



**Unordered Set:**

Unordered set ka use hum tab kr sakte hai jab hume pata karna ho ki koi value set mai present hai ya nahi bcoz set ka T.C O(1) hoti hai.

1. Ismai bhi same unordered map ki tarah Valid keys datatype does not support complex data type like pair,set ,pair of pair,vector etc.

Multiset:

Multiple values ko store kr sakte hai

Eg:

s.insert(“abc”);

s.insert(“abc”);

duplicates are allowed.

Multiset ko hum priority queue ki jagah use kr sakte hai or kafi easy ho jate hai.

Or agar duplicates hai toh pehle wale ka iterator return karta hai.

int main(){

multiset<string> s;

s.insert("abc"); // o(log(n))

s.insert("zsdf");

s.insert("bcd");

s.insert("abc");

auto it=s.find("abc"); // 0(log(n))

**if(it !=s.end ()){**

**s.erase(it); }** // yeh kewal eek abc ko delete karega jo ki pehle iterator se pointed thi

s.erase(“abc”); //delete all abc.

**Very good concept to find element:**

Using iterator it store the address of element if element is not present then it store s.end() so use it

            auto it = stud.find(candi2);

            if(it != stud.end()){

                cout<<"YES"<<endl;

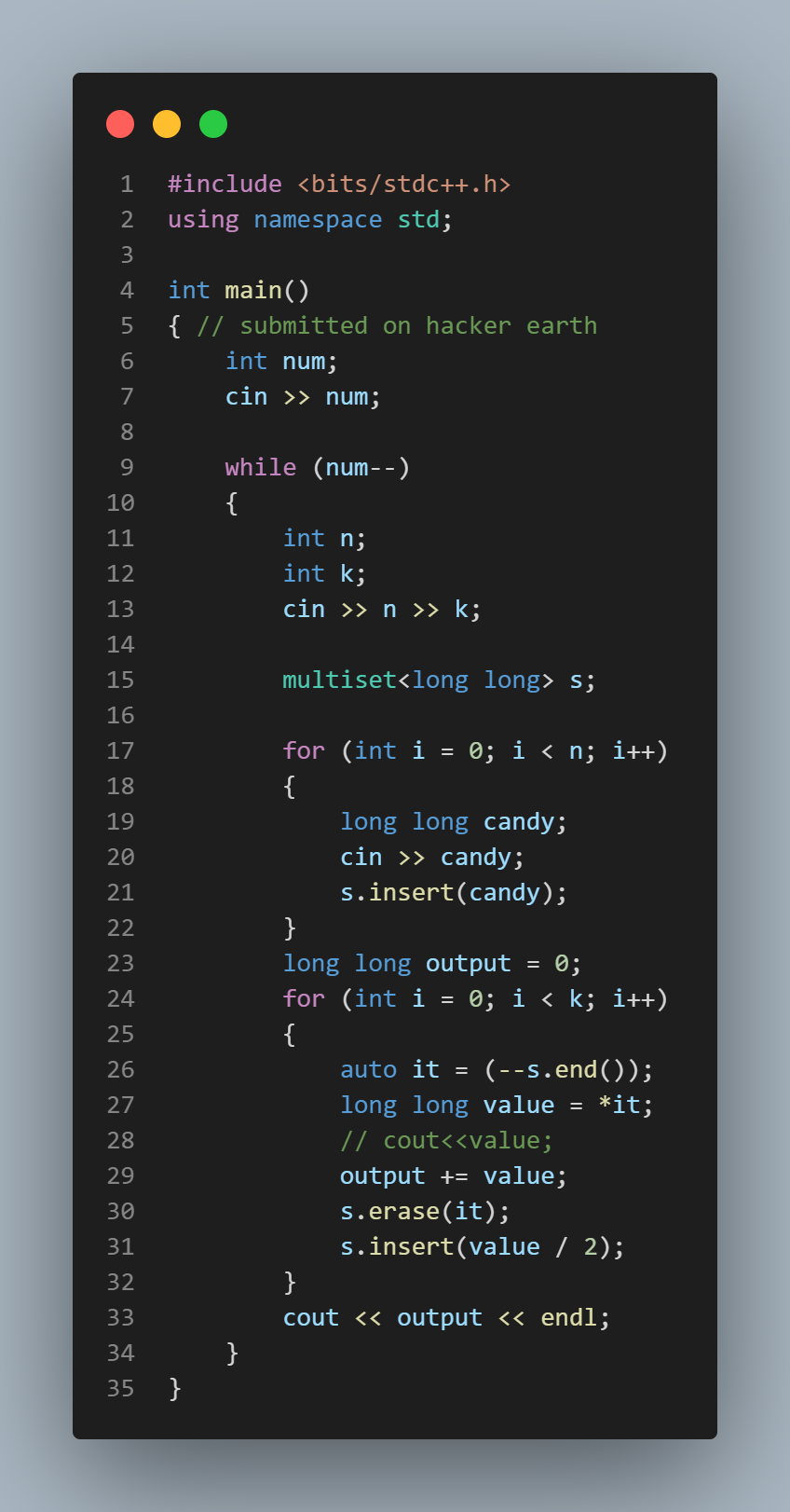
            }

            else{

                cout<<"NO"<<endl;

            }

**Multiset QUESTION: Very GOOD Que Must Revise**



**Nesting in STL:**

Normal data type: map<int , int> m;

Map<pair<int, int>, int> m;

Pair<int, int> p1, p2

P1 = {1,2};

P2 = {2,3};

Cout << p1<p2 ;

Less than and greater than operator works here.

P1<p2

1.Pair ka comparision pehli value se hoti hai if equal hai then second key se compare hota hai.

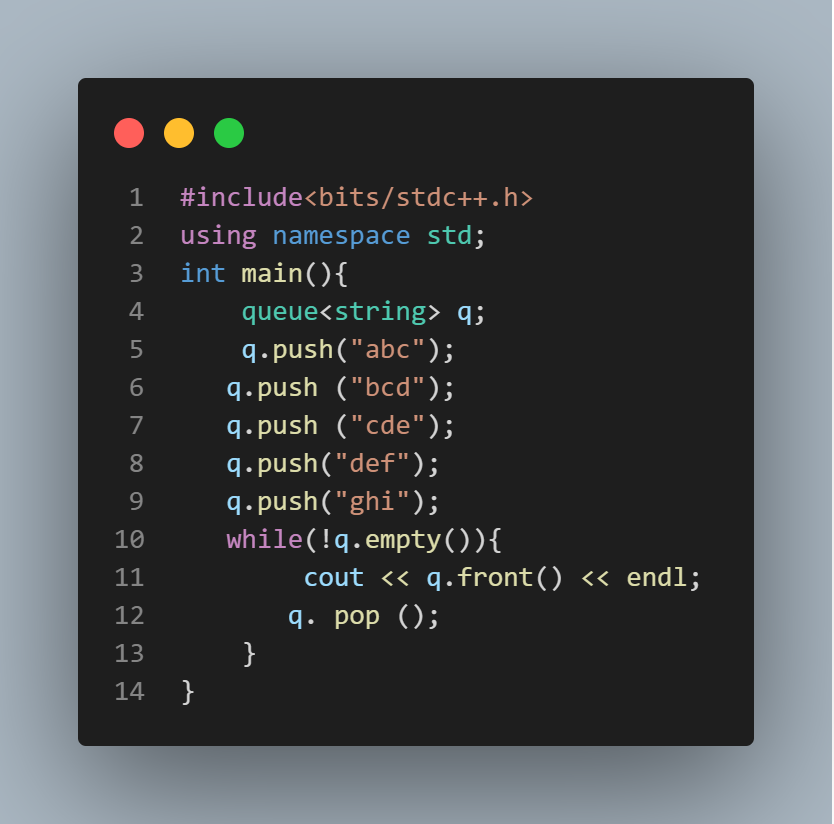
**Nesting in stl Que**



**Balanced paranthesis II nd approach**



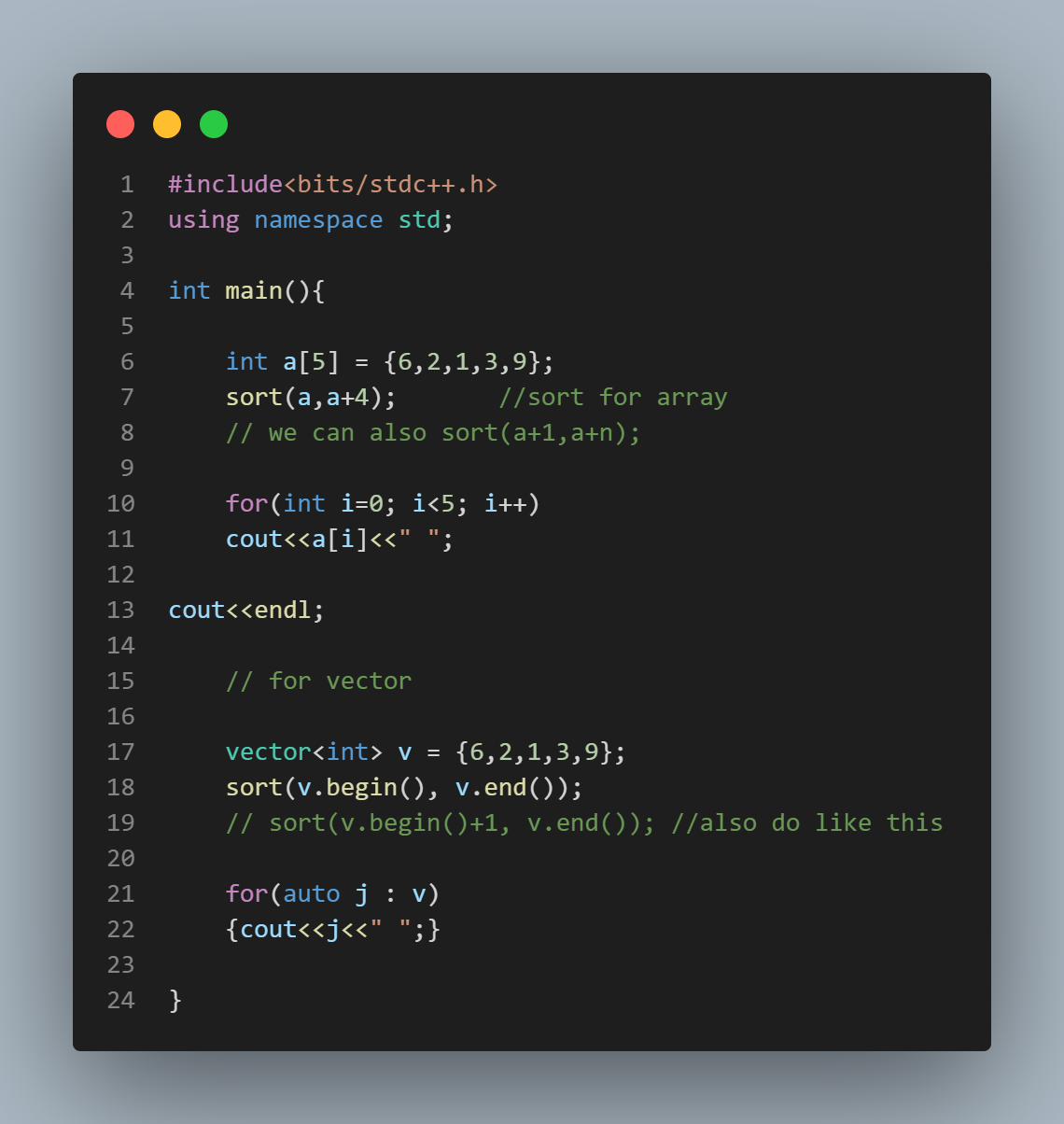
**Queue**



**Stack**



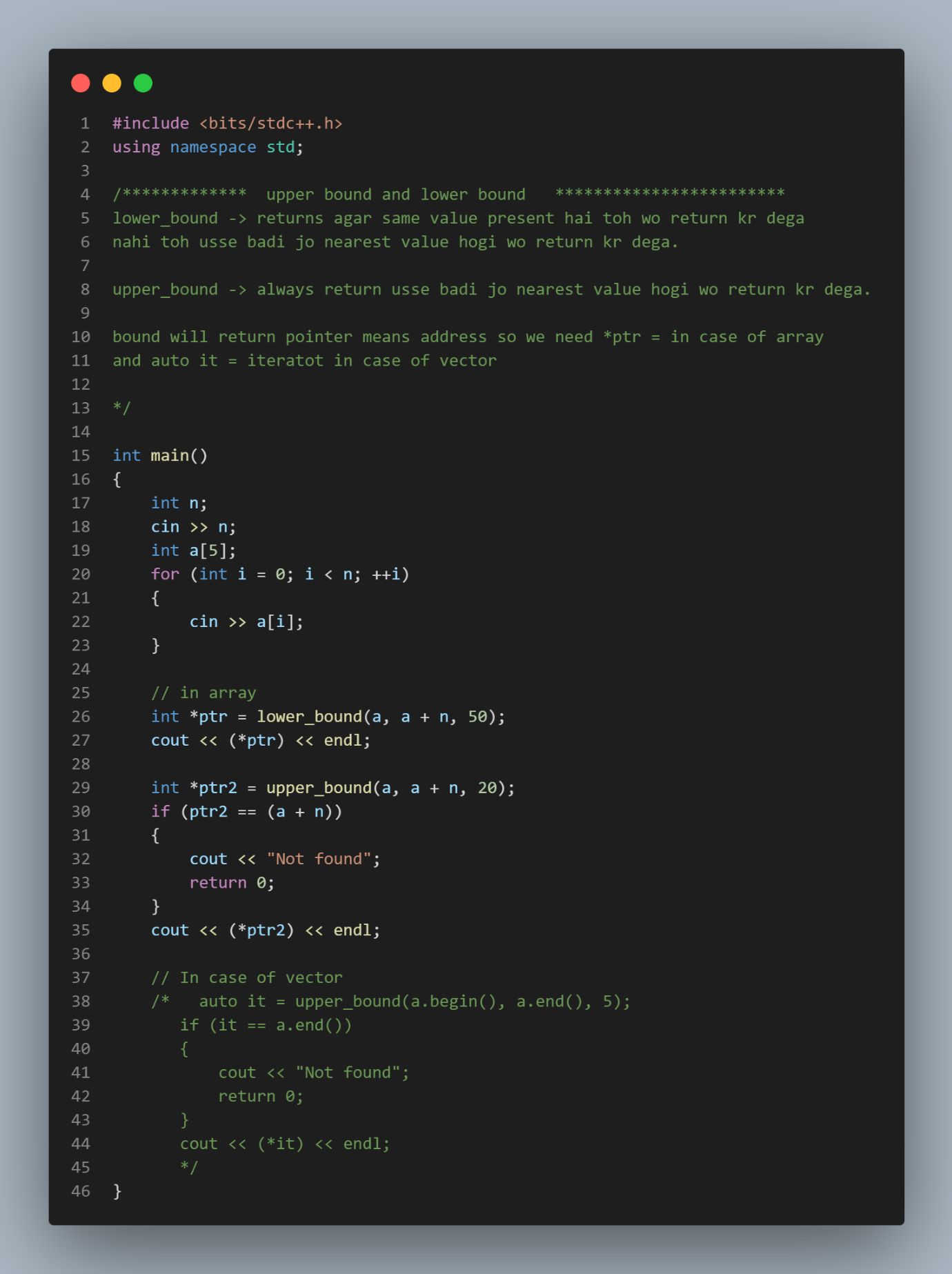
**Sort**



**Comparator**



**Upper bound lower bound**



**Algo 1**

