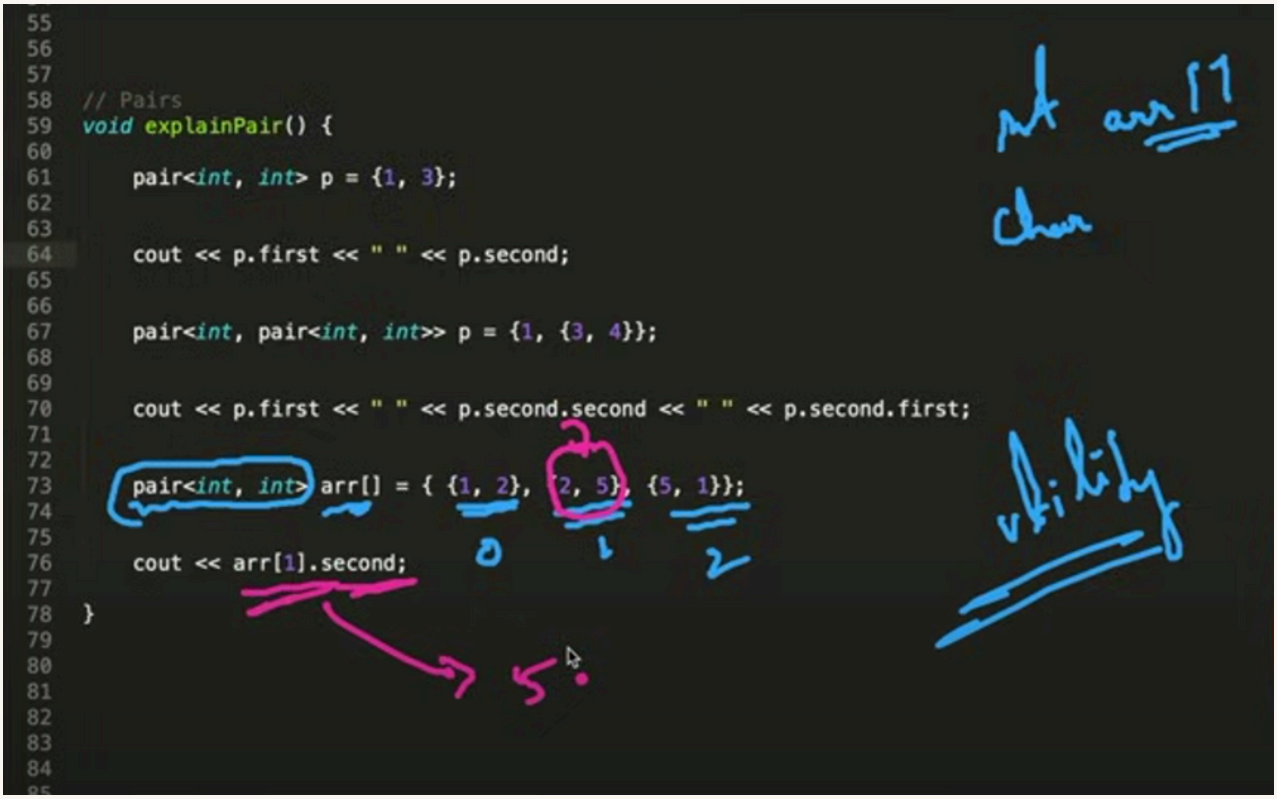
NOTES

(STL)

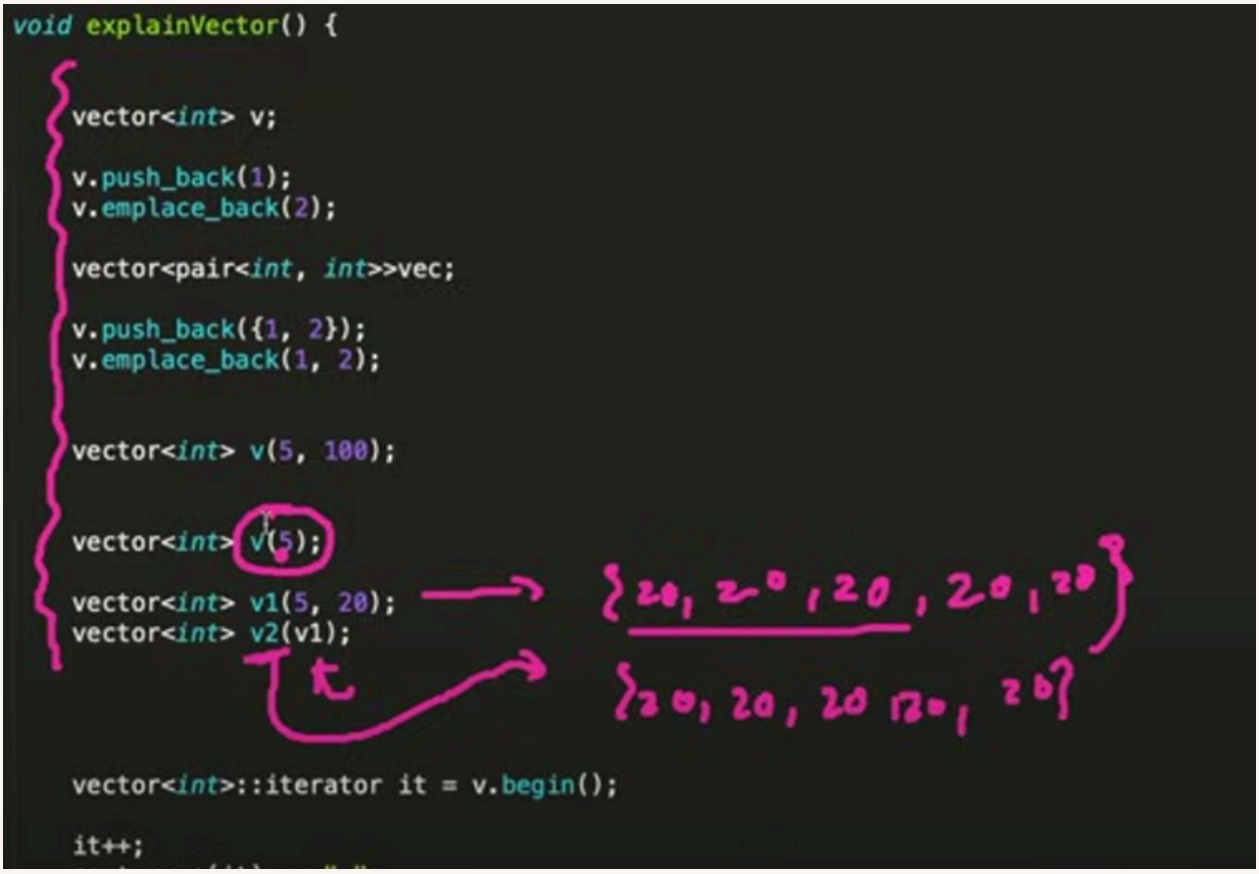
source : striver(youtube)

1. Pair



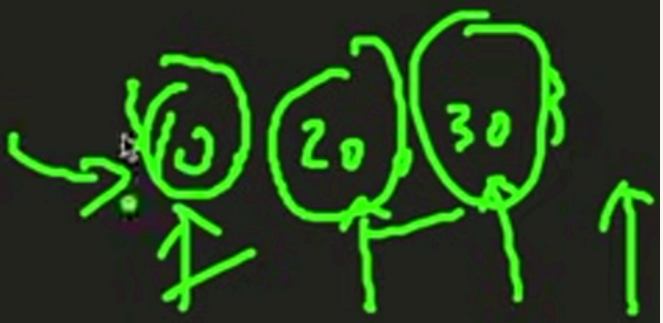
```
// Pairs
void explainPair() {

    pair<int, int> p = {1, 3};

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

    pair<int, pair<int, int>> p = {1, {3, 4}};

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

    pair<int, int> arr[] = { {1, 2}, {2, 5}, {5, 1}};

    cout << arr[1].second;
}
```

2. Vector

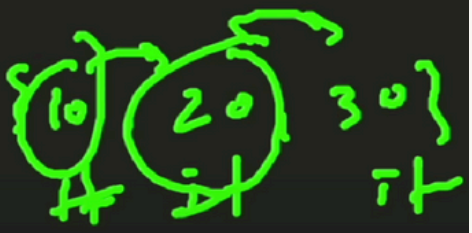


```
void explainVector() {

    vector<int> v;

    v.push_back(1);
    v.emplace_back(2);

    vector<pair<int, int>>vec;

    v.push_back({1, 2});
    v.emplace_back(1, 2);


    vector<int> v(5, 100);


    vector<int> v(5);

    vector<int> v1(5, 20);
    vector<int> v2(v1);


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

    it++;
```

```cpp
vector<int>::iterator it = v.begin();

it++;
cout << *(it) << " ";


it = it + 2;
cout << *(it) << " ";

vector<int>::iterator it = v.end();

vector<int>::iterator it = v.rend();

vector<int>::iterator it = v.rbegin();


cout << v[0] << " " << v.at(0);

cout << v.back() << " ";


for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
    cout << *(it) << " ";
}

for (auto it = v.begin(); it != v.end(); it++) {
    cout << *(it) << " ";
}
```

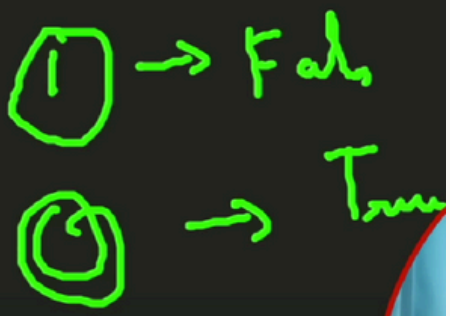



```cpp
cout << v[0] << " " << v.at(0);

cout << v.back() << " ";



for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
    cout << *(it) << " ";
}


for (auto it = v.begin(); it != v.end(); it++) {
    cout << *(it) << " ";
}

for (auto it : v) {
    cout << it << " ";
}

// {10, 20, 12, 23}
v.erase(v.begin()+1);

// {10, 20, 12, 23, 35}
v.erase(v.begin() + 2, v.begin() + 4); // // {10, 20, 35} [start, end)
```



```
// {10, 20, 12, 23}
v.erase(v.begin()+1);

// {10, 20, 12, 23, 35}
v.erase(v.begin() + 2, v.begin() + 4); // // {10, 20, 35} [start, end)
```

```
//Insert function

vector<int>v(2, 100); // {100, 100}
v.insert(v.begin(), 300); // {300, 100, 100};
v.insert(v.begin() + 1, 2, 10); // {300, 10, 10, 100, 100}

vector<int> copy(2, 50); // {50, 50}
v.insert(v.begin(), copy.begin(), copy.end()); // {50, 50, 300, 10, 10, 100, 100}

// {10, 20}
cout << v.size(); // 2

//{10, 20}
v.pop_back(); // {10}

// v1 -> {10, 20}
// v2 -> {30, 40}
v1.swap(v2); // v1 -> {30, 40} , v2 -> {10, 20}

v.clear(); // erases the entire vector

cout << v.empty();
```



3. List

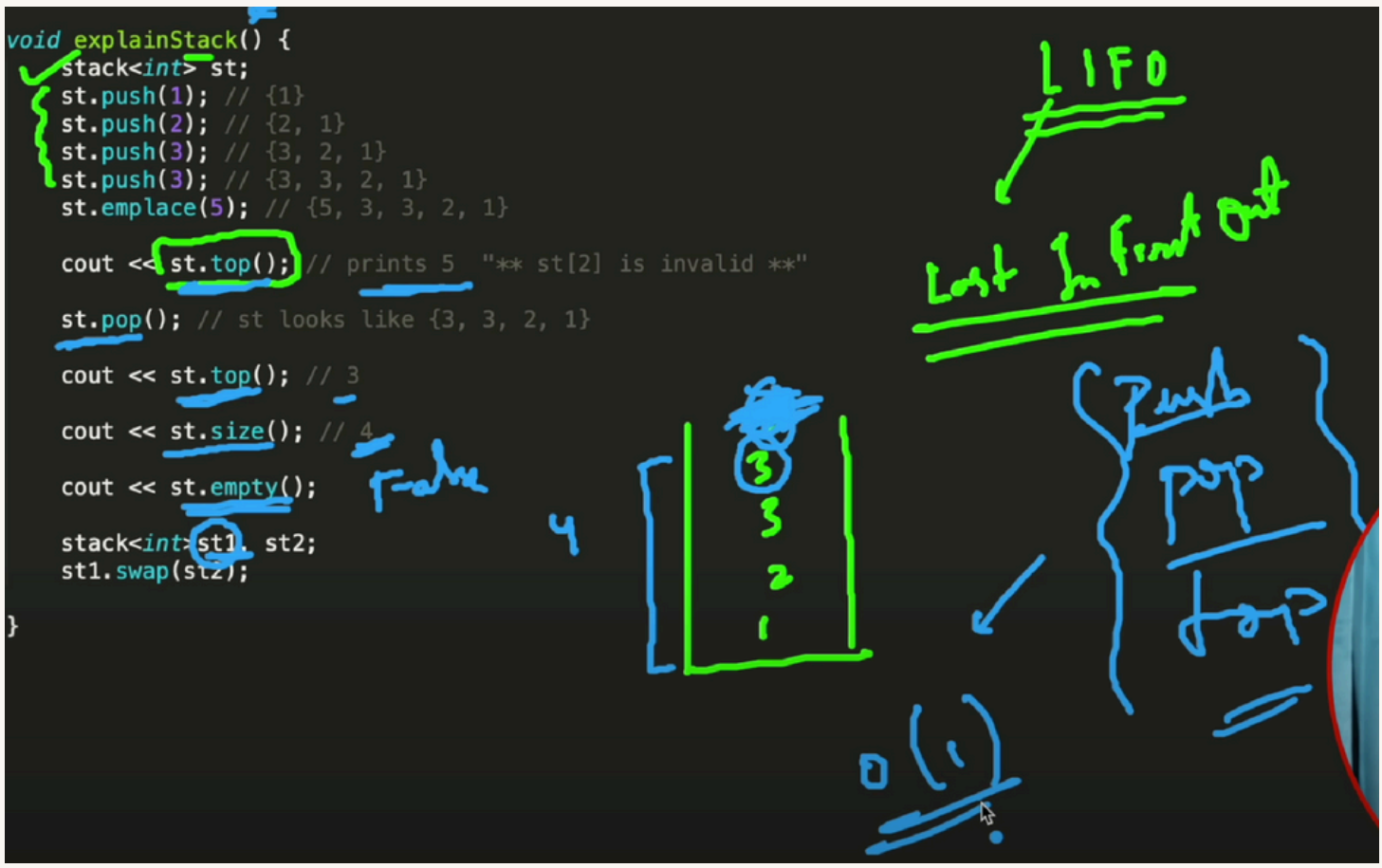
```
void explainList() {
    list<int> ls;

    ls.push_back(2); // {2}
    ls.emplace_back(4); // {2, 4}

    ls.push_front(5); // {5, 2, 4};

    ls.emplace_front(); {2, 4};

    // rest functions same as vector
    // begin, end, rbegin, rend, clear, insert, size, swap
}
```

4. Deque

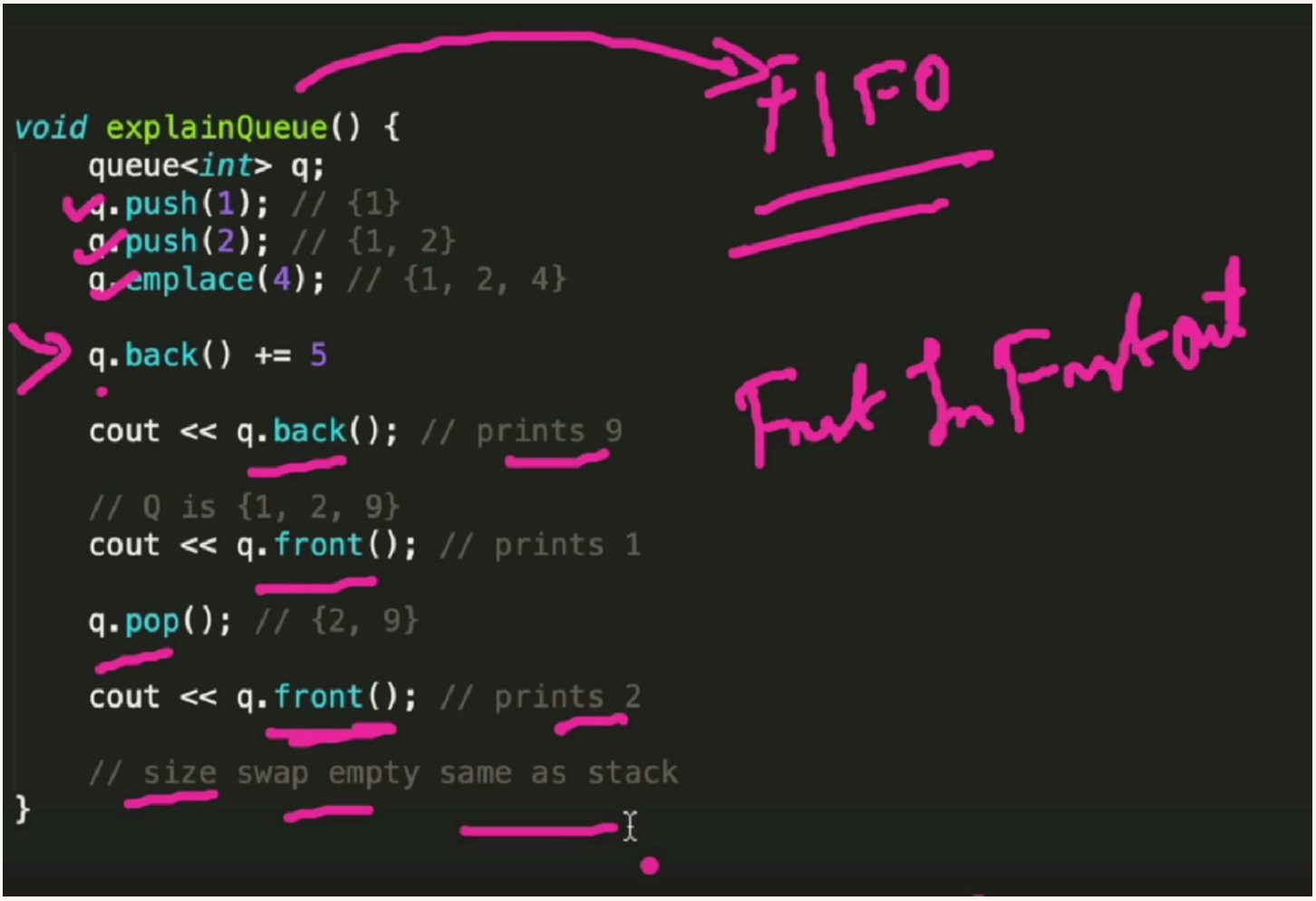
```
void explainDeque() {

    deque<int>dq;
    dq.push_back(1); // {1}
    dq.emplace_back(2); // {1, 2}
     dq.push_front(4); // {4, 1, 2}
     dq.emplace_front(3); // {3, 4, 1, 2}

    dq.pop_back(); // {3, 4, 1}
    dq.pop_front(); // {4, 1}

    dq.back();

     dq.front();

     // rest functions same as vector
     // begin, end, rbegin, rend, clear, insert, size, swap
}
```

5. Stack

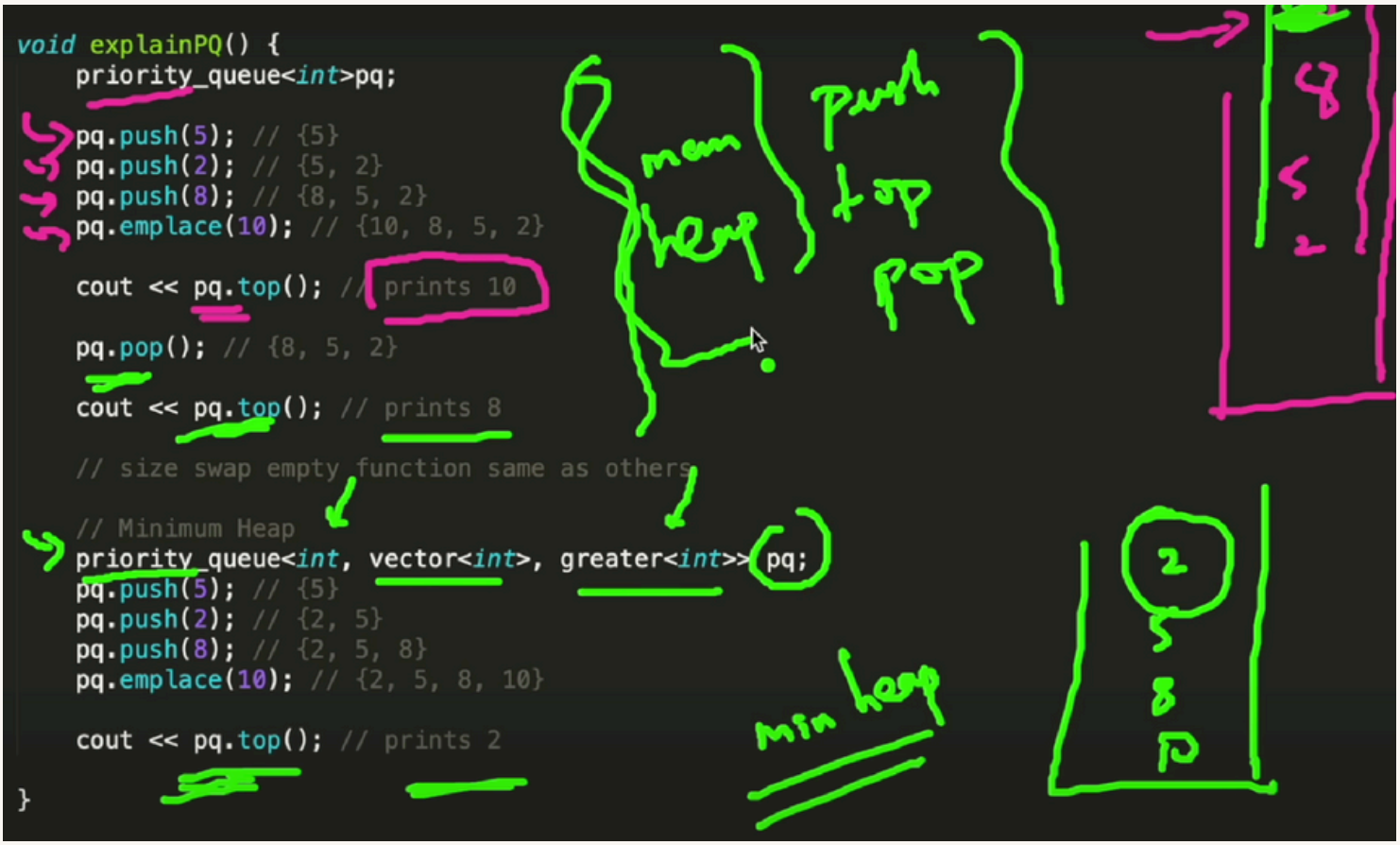
```
void explainStack() {
    stack<int> st;
    st.push(1); // {1}
    st.push(2); // {2, 1}
    st.push(3); // {3, 2, 1}
    st.push(3); // {3, 3, 2, 1}
    st.emplace(5); // {5, 3, 3, 2, 1}

    cout << st.top(); // prints 5  "** st[2] is invalid **"

    st.pop(); // st looks like {3, 3, 2, 1}

    cout << st.top(); // 3

    cout << st.size(); // 4

    cout << st.empty();

    stack<int> st1, st2;
    st1.swap(st2);

}
```



6. Queue



7. Priority Queue



8. Set

note: Always store in sorted order and unique elements

```
void explainSet() {
    set<int>st;
    st.insert(1); // {1}
    st.emplace(2); // {1, 2}
    st.insert(2); // {1, 2}
    st.insert(4); // {1, 2, 4}
    st.insert(3); // {1, 2, 3, 4}

    // Functionality of insert in vector
    // can be used also, that only increases
    // efficiency

    // begin(), end(), rbegin(), rend(), size(),
    // empty() and swap() are same as those of above

    // {1, 2, 3, 4, 5}
    auto it = st.find(3);

    // {1, 2, 3, 4, 5}
    auto it = st.find(6);

    // {1, 4, 5}
    st.erase(5); // erases 5 // takes logarithmic time
```

```
// {1, 2, 3, 4, 5}
auto it = st.find(3);

// {1, 2, 3, 4, 5}
auto it = st.find(6);

// {1, 4, 5}
st.erase(5); // erases 5 // takes logarithmic time


int cnt = st.count(1);


auto it = st.find(3);
st.erase(it); // it takes constant time
```

```
// {1, 2, 3, 4, 5}
auto it1 = st.find(2);
auto it2 = st.find(4);
st.erase(it1, it2); // after erase {1, 4, 5} [first, last)

// lower_bound() and upper_bound() function works in the same way
// as in vector it does.

// This is the syntax
auto it = st.lower_bound(2);

auto it = st.upper_bound(3);
```

9. Multi- Set

note: Always store in sorted order but not unique elements

```
void explainMultiSet() {
    // Everything is same as set
    // only stores duplicate elements also


    multiset<int>ms;
    ms.insert(1); // {1}
    ms.insert(1); // {1, 1}
    ms.insert(1); // {1, 1, 1}

    ms.erase(1); // all 1's erased

    int cnt = ms.count(1);

    // only a single one erased
    ms.erase(ms.find(1));


    ms.erase(ms.find(1), ms.find(1)+2);

    // rest all function same as set
}
```

10. Unordered - Set

note: store unique elements but Randomise order

```cpp
void explainUSet() {
    unordered_set<int> st;
    // lower_bound and upper_bound function
    // does not works, rest all functions are same
    // as above,  it does not stores in any
    // particular order it has a better complexity
    // than set in most cases, except some when collision happens
}
```

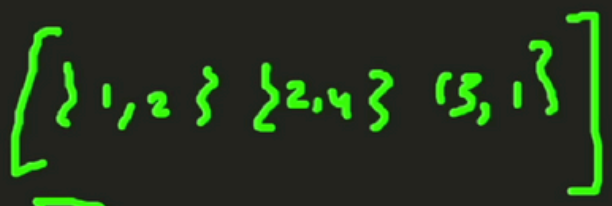
11. Map

note:

- **syntax: map<key,value>**
- **key: any data structure(unique)**
- **value: any data structure(can be duplicate)**

Note: * map store unique Keys in sorted data (just like sets) -> sorted means Keys not values

```cpp
void explainMap() {

    map<int, int> mpp;

    map<int, pair<int, int>> mpp;

    map< pair<int, int>, int> mpp;


    mpp[1] = 2;
    mpp.emplace({3, 1});

    mpp.insert({2, 4});

    mpp[{2,3}] = 10;


    for(auto it : mpp) {
        cout << it.first << " " << it.second << endl;
    }


    cout << mpp[1];
    cout << mpp[5];

    auto it = mpp.find(3);
    cout << *(it).second;

    auto it = mpp.find(5);
```



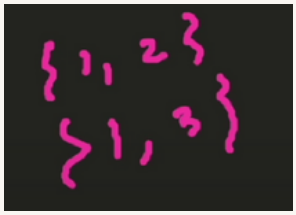
```cpp
    for(auto it : mpp) {
        cout << it.first << " " << it.second << endl;
    }


    cout << mpp[1];
    cout << mpp[5];

    auto it = mpp.find(3);
    cout << *(it).second;

    auto it = mpp.find(5);

    // This is the syntax
    auto it = mpp.lower_bound(2);

    auto it = mpp.upper_bound(3);

    // erase, swap, size, empty, are same as above

}
```

mpp.end()

12. Multi-Map

note: **store sorted order, duplicates can be stored(just like multi-set)**

LIKE THIS



read below

```cpp
void explainMultimap() {
    // everything same as map, only it can store multiple keys
    // only mpp[key] cannot be used here

}
```

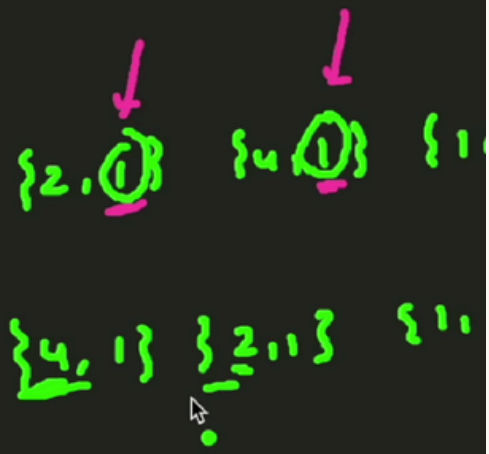
13. Unordered-Map

UNQIUE KEYS BUT NOT RANDOMIZE

```cpp
void explainUnorderedMap() {
    // same as set and unordered_Set difference.
}
```

Some Imp- Algotithms:

```
void explainExtra() {

    sort(a, a + n);
    sort(v.begin(), v.end());

    sort(a+2, a+4);

    sort(a, a+n, greater<int>);

    pair<int,int> a[] = {{1,2}, {2, 1}, {4, 1}};

    // sort it according to second element
    // if second element is same, then sort
    // it according to first element but in descending

    sort(a, a+n ,comp);

    // {4,1}, {2, 1}, {1, 2}};
```



- use of my comparator

```
bool comp(pair<int,int> p1, pair<int,int> p2) {
    if(p1.second < p2.second) return true;
    if(p1.second > p2.second) return false;
    // they are same

    if(p1.first > p2.first) return true;
    return false;
}
```

end