(*) bool next_permutation (Itr start, Itr stop) -

bidirectional

Given a range of elements [start, stop), modifies the range to contain the next lexicographically higher permutation of these elements.

(*) bool prev_permutation (start, stop) - Next lexicographically lower permutation of those elements.

(*) Itr search (Itr start 1, Itr stop 1, Itr start 2, Itr stop 2)

Returns whether seq. [start2, stop2) is a subsequence of the range [start 1, stop 1).

→ remove is an iterator function so it actually does not removes.

→ erase is a container class member function so it completely erases the element.

(*) size_t count_if (InputItr start, InputItr end, Predicate function fn)-

Returns - No of element in the range [start, stop) for which fn return true.
Useful for determining how many elements have certain property.

(*) void fill (ForwardItr start, ForwardItr stop, const Type& value)-

Set every element in the range [start, stop) to value.

(*) void fill_n (Itr start, size_t num, const Type& value)-
Sets the first num elements, starting at start to value.

(*) InputItr find (Itr start, Itr stop, const Type& value)-
Returns - an iterator to the first element in [start, stop) that is equal to value.

(*) function for_each (Itr start, Itr stop, Function fn)-
calls the function on each element in the range

(*) Type inner_product (Itr start1, Itr stop1, Itr start2, Type initial value)-

computes inner product $\Sigma a_i b_i$ + initial value
a_i and b_i denotes the ith element of the first and second range

(*) back_insert_iterator <vector<int>> itr (myVector);

back_insert_iterator <deque<char>> itr
= back_inserter <myDeque>;

(*) insert_iterator <set <int >> itr (myset, myset.begin());

insert_iterator <set<int >> itr = inserter (myset, myset.begin());

Removal Algorithms -

(*) accumulate (InputItr start, InputItr stop, Type value) -

Returns the sum of elements in the range [start, stop) plus the value of value (in the arg.)

(*) OutItr copy (InputItr start, InputItr stop, OutItr output start) -

copies the element in the range [start, stop) into the output range starting at output start.
Return → an iterator to one past the range written to.

(*) size_t count (InputItr start, InputItr stop, const Type& value)

Returns → no of elements in the range [start, stop) equal to value

STL Algorithms

* accumulate() -

* accumulate (values.begin(), values.end(), 0.0)
We can also give the values i.e

accumulate (values.lower_bound(42), values.upper_bound(137), 0)

→ In its implementation it also uses a loop.

Reordering algo:

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

→ sort requires random access iterators.
→ We can not use sort in set or map.

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

(*) rotate (v.begin(), v.begin()+2, v.end());

(*) find (v.begin(), v.end(), 137) != v.end())

NOTE: It is an STL algo. (find()) and we can equally well use it for set and map but we should use the member function instead of algorithms, as they are more optimal.

(*) binary_search (v.begin(), v.end(), 87)
On a sorted vector.

(*) .insert() - map <string, int> m;
m.insert (make_pair ("zero", 0));

→ map's insert() function returns a value of type pair <iterator, bool>.

(*) .empty() - Returns a bool value showing map is empty or not.

MULTICONTAINERS (Multimap, multiset)

(*) .equal_range() - returns a pair <iterator, iterator> that returns the span of entries equal to specified value.

→ All functions works same as map, set.
→ Apart from some like -

(•) .count() (•) .erase (•) .find()

Container	Insertion	Access	Erase	Find	Persistent Iterator
vector	Back: O(1) or O(n) Other: O(n)	O(1)	Back: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	No
deque	Back/Front: O(1) Other: O(n)	O(1)	Back/Front: O(1) Other: O(n)	Sort: O(log n) Other: O(n)	Pointers
set/map	O(log n)		O(log n)	O(log n)	Yes
unordered set/map	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	Pointers

(*) .size() - returns the no of elements.

(*) set<int> s(v.begin(), v.end());

it copies all the element in the vector to the set.

(*) .empty() - returns ~~nothing~~ whether set is empty

Insertion -

(*) set.insert(4) - inserts 4 into the set.
return type → pair (containing an iterator to the element and bool indicating whether element is inserted successfully).

(*) s.insert(v.begin(), v.end())

(*) .find() - Returns an iterator to the specified elements if exists, and end otherwise.

For eg - if (s.find(0) != s.end())

PAIR

pair < TypeOne, TypeTwo >

Declaration -

```
pair<int, string> p;
p.first = 137;
p.second = "C++";
```

(*) .make_pair() -

```
pair<int, string> p = make_pair(137, "C++");
```

(*) .erase() - It removes the element from the set, if present.

```
s.erase(137);
```

It removes 137 from the set.

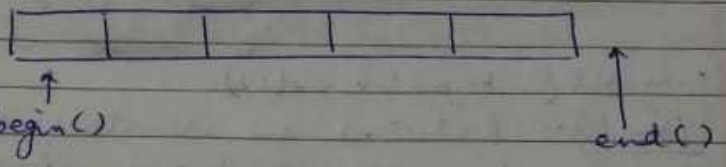
ITERATOR

(*) An iterator in a set or in a vector.

Syntax-

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

If we have got a vector-



Three major operations on ~~~~ iterators

→ Dereferencing ⇒ *itr
→ Advancing from one position to next
→ Comparing two iterators.

(*) .lower_bound() - it returns a pointer to the first element in the set greater than or equal to that value.

(*) .upper_bound() - accepts a value and returns an iterator to the first element in the set that is strictly greater than the specified element

C++ Iso Standard

vector is a type of sequence that should be used by default,
deque should be used when we have to make more insertions and deletions at the beginning or at the end

Set - It represents unordered collection of element and has a good support for the following operations.

- Adding elements
- Removing elements
- Determining whether a particular element is in the collection.

(*) insert() - It does not require an index, as set is an unordered collection of elements.

```
set <int> s;
s.insert(value);
```

(*) count() - It finds out whether an element is present in a set.

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

It returns true in case the set contains element otherwise returns false

(*) It does not permit duplicate elements.
(*) Set are implemented using Balanced Binary Tree

(*) iterator end()-

```
while (itr != v.end());
```

Returns an iterator to the element after last. The iterator returned by end does not point to an element in the vector.

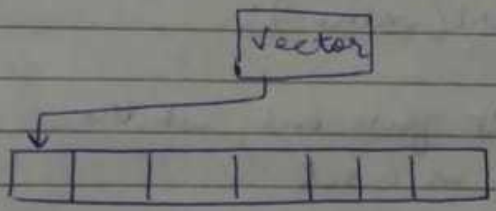
Double ended queue (deque)-

→ deque supports all the functions as vector.
→ But it comes with some more fire powers.

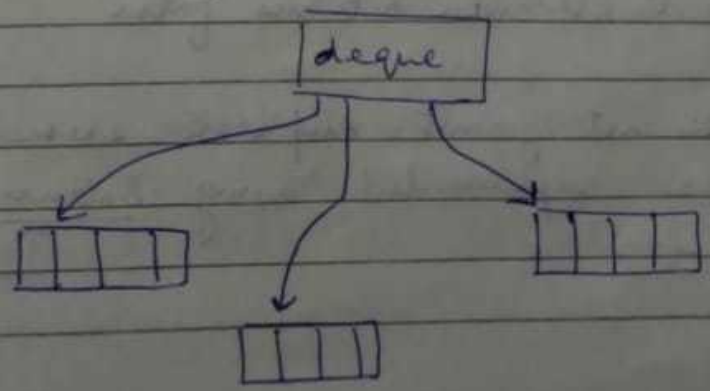
Two more functions are-
- push_front()
- pop_front().

Element storage in vectors and deque-

(*) Vector stores element in contiguous memory address



(*) deques on the other hand maintains a list of different "pages"



(*) Initialization while declaration -

vector < int > v (10);
size 10, all initialized to 0.

Other way out -

vector <string> v(5, "none");

(*) .resize() - It takes two arguments.

→ It initializes the vector with given values only if the element is newly added to the vector.

v. resize (10); → it will have 10 elements all initialized to 0.

~~(*) .erase~~

v.resize(5) - it will have 5 elements all 0.

v.resize (7,1) - only two newly added elements will have 1 in them.

(*) .erase() - v.erase(v.begin() + n);
It removes the desired index element

(*) .clear() - It clears the whole vector.

(*) .empty() - Returns whether vector is empty or not.

(*) [] - Referencing a particular index.

(*) .at() - Refering a particular index.

MAP -

map < string, int > m;

- It is implemented using Balanced Binary Tree

Function supported by map -

- Inserting a new key/value pair.
- Checking whether a particular key exists.
- Querying which value is associated with a given key.
- Removing a pair.

(*) m["zero"] = 0;
it will store the pair.
Now if we do not have a key in our key value pair

m["xyyz"]

Then it will implicitly map to 0.

(*) · find() - takes in a key, returns an iterator ~~that returns~~ to the key value pair that has specified key.

Actually map stores a constant key value and a mapped value i.e

pair < const KeyType, ValueType >;

↓

a key can not be modified.

(*) · erase() - It will take key as argument and will remove both key/value pair.

STL containers

STL is logically divided into six pieces:

- Containers: Storing the data in an efficient manner.

  for eg-
  → map: Associative collection of key, value
  → vector: growing list of elements.

- Iterators: They have a common interface, they work as pointers

- Algorithms: Functions that operate at data specified by iterators.

- Adapter · Functors · Allocators

VECTOR

(*) We will be sorting the vector in-place, we will not be creating a new vector.

(*) v.size() – gives the size

(*) Declaration – vector < primitive or non-primitive data type > V1;

(*) Insertion – v.insert(v.begin()+n, 137)

↓

iterator (gives a pointer, pointing to starting of vector).

(*) push_back(): V.push_back(15);
Inserts 15 at the end of list

Problem with cin?

→ It does not read after space and pushes the input after a space into next cin

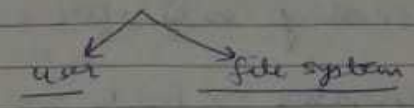
(*) Solution- We can use getline (cin, —).
It ends up reading only after a newline.

(*) Extraction operator → Reads one token at a time

(*) getline → Reads complete one line from the file.

STREAMS (streams library)

Stream lib: collection of functions that allows you to read and write formatted data from a variety of sources.



Unit of communication — stream

CONSOLE i/p - o/p -

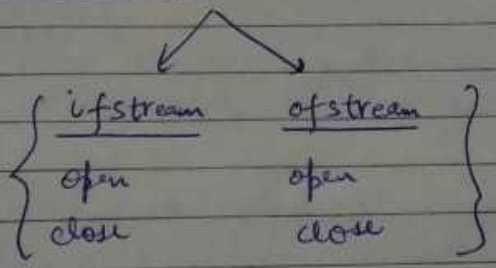
include <iostream>

<< - stream insertion operator

cout ← stream object

>> - stream extraction operator

FILE i/p - o/p -



Stream manipulators - endl → is a stream manipulator that can be inserted into a stream to change some stream property.