C++ Data Structures & Algorithms (DSA) Cheat Sheet

This cheat sheet is a quick reference for the most commonly used data structures and algorithms in C++ for competitive programming and software development. It primarily focuses on the Standard Template Library (STL).

Table of Contents

1. Headers & Basic Setup

A common setup for competitive programming includes the <bits/stdc++.h> header (in GNU GCC) which imports all standard libraries. For production code, it's better to include specific headers.

```
// For competitive programming
#include <bits/stdc++.h>

// For production/specific includes
#include <iostream>
```

```
#include <vector>
#include <string>
#include <algorithm>
#include <set>
#include <map>
// ... etc.

using namespace std;

int main() {
    // Fast I/O
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    // Your code here

    return 0;
}
```


2. Sequence Containers

These containers store elements in a linear sequence.

std::vector

Dynamic array that can grow and shrink in size. Elements are stored contiguously.
Header: <vector>

Function	Description	Complexity
push_back(val)	Adds an element to the end.	Amortized O(1)
pop_back()	Removes the last element.	O(1)
size()	Returns the number of elements.	O(1)
begin(), end()	Returns iterators to the	O(1)

	beginning/end.	
rbegin(), rend()	Returns reverse iterators.	O(1)
[i] or at(i)	Access element at index i.	O(1)
insert(pos, val)	Inserts element at iterator pos.	O(N)
erase(pos)	Erases element at iterator pos.	O(N)
clear()	Removes all elements.	O(N)
sort(v.begin(), v.end())	Sorts the vector. (Requires <algorithm>)	O(N log N)

Example:

```
vector<int> v = {1, 2, 3};
v.push_back(4); // v is now {1, 2, 3, 4}
cout << v[1]; // prints 2
v.pop_back(); // v is now {1, 2, 3}
```


std::deque

Double-ended queue. Similar to a vector, but allows fast insertions and deletions at both the beginning and the end.
Header: <deque>

Function	**Description**	**Complexity**
push_back(val)	Adds an element to the end.	Amortized O(1)
pop_back()	Removes the last element.	O(1)
push_front(val)	Adds an element to the front.	Amortized O(1)
pop_front()	Removes the first element.	O(1)
front()	Access the first element.	O(1)
back()	Access the last element.	O(1)

[i] or at(i)	Access element at index i.	O(1)

Example:

```
deque<int> dq;
dq.push_front(10); // dq is {10}
dq.push_back(20);  // dq is {10, 20}
dq.push_front(5);  // dq is {5, 10, 20}
cout << dq.front(); // prints 5
dq.pop_front(); // dq is {10, 20}
```


std::list

Doubly-linked list. Fast insertions and deletions anywhere, but slow random access.
Header: <list>

Function	Description	Complexity
push_back(val), pop_back()	Add/remove from the end.	O(1)
push_front(val), pop_front()	Add/remove from the front.	O(1)
insert(pos, val)	Inserts element at iterator pos.	O(1)
erase(pos)	Erases element at iterator pos.	O(1)
sort()	Sorts the list (member function).	O(N log N)
reverse()	Reverses the list.	O(N)

Example:

```
list<int> l = {10, 30};
l.push_front(5); // l is {5, 10, 30}
auto it = l.begin();
it++; // it points to 10
l.insert(it, 7); // l is {5, 7, 10, 30}
```


std::array

Fixed-size array. Provides the efficiency of a C-style array with the benefits of an STL container (like iterators).
Header: <array>

```
array<int, 4> arr = {1, 2, 3, 4};
cout << arr[1];      // prints 2
cout << arr.size();  // prints 4
sort(arr.begin(), arr.end()); // arr becomes {1, 2, 3, 4}
```


3. Associative Containers

These containers store elements in a sorted order, which allows for fast searching.

std::set & std::multiset

Stores a collection of unique (set) or non-unique (multiset) sorted elements. Implemented as a balanced binary search tree (usually a Red-Black Tree).
Header: <set>

Function	Description	Complexity
insert(val)	Inserts an element.	O(log N)
erase(val)	Erases all occurrences of val.	O(log N + count)
erase(pos)	Erases element at iterator pos.	Amortized O(1)
find(val)	Returns iterator to val, or end() if not found.	O(log N)
count(val)	Returns number of occurrences of val.	O(log N + count)
lower_bound(val)	Iterator to first element not less than val.	O(log N)

upper_bound(val)	Iterator to first element greater than val.	O(log N)

Example (set):

```
set<int> s;
s.insert(10);
s.insert(30);
s.insert(10); // Ignored, as 10 is already present
// s contains {10, 30}
if (s.find(30) != s.end()) {
   cout << "Found 30";
}
```


std::map & std::multimap

Stores key-value pairs sorted by key. map has unique keys, multimap allows duplicate keys.
Header: <map>

Function	Description	Complexity
insert({key, val})	Inserts a key-value pair.	O(log N)
erase(key)	Erases element(s) with the given key.	O(log N)
find(key)	Returns iterator to element with key.	O(log N)
[key]	Accesses value. If key doesn't exist, it's created.	O(log N)
at(key)	Accesses value. Throws exception if key doesn't exist.	O(log N)
lower_bound(key)	Iterator to first element with key not less than key.	O(log N)
upper_bound(key)	Iterator to first element with key greater than key.	O(log N)

Example (map):

```
map<string, int> m;
m["apple"] = 100;
m["banana"] = 50;
m.insert({"cherry", 120});
// m contains {("apple", 100), ("banana", 50), ("cherry", 120)}
cout << m["apple"]; // prints 100
m.erase("banana");
```


4. Unordered Associative Containers

These containers store elements using a hash table, providing average-case constant time complexity for most operations. Order is not guaranteed.

std::unordered_set & std::unordered_multiset

Unordered version of set and multiset.
Header: <unordered_set>

Function	Description	Complexity (Avg / Worst)
insert(val)	Inserts an element.	O(1) / O(N)
erase(val)	Erases all occurrences of val.	O(1) / O(N)
find(val)	Returns iterator to val, or end().	O(1) / O(N)
count(val)	Returns number of occurrences of val.	O(1) / O(N)

Example:

```
unordered_set<int> us;
us.insert(20);
us.insert(10);
us.insert(20); // Ignored
```

// us contains {10, 20} in some arbitrary order

std::unordered_map & std::unordered_multimap

Unordered version of map and multimap.
Header: <unordered_map>

Function	Description	Complexity (Avg / Worst)
insert({key, val})	Inserts a key-value pair.	O(1) / O(N)
erase(key)	Erases element(s) with the given key.	O(1) / O(N)
find(key)	Returns iterator to element with key.	O(1) / O(N)
[key]	Accesses value. If key doesn't exist, it's created.	O(1) / O(N)
at(key)	Accesses value. Throws exception if key doesn't exist.	O(1) / O(N)

Example:

```
unordered_map<string, int> um;
um["task1"] = 1;
um["task2"] = 2;
cout << um["task1"]; // prints 1
```


5. Container Adapters

These provide a specific interface on top of an underlying container.

std::stack

LIFO (Last-In, First-Out) structure. Default underlying container is std::deque.
Header: <stack>

Function	Description	Complexity
push(val)	Adds element to the top.	O(1)
pop()	Removes element from the top.	O(1)
top()	Accesses the top element.	O(1)
empty()	Checks if the stack is empty.	O(1)
size()	Returns the number of elements.	O(1)

Example:

```
stack<int> s;
s.push(1); // s: {1}
s.push(2); // s: {1, 2}
cout << s.top(); // prints 2
s.pop(); // s: {1}
```


std::queue

FIFO (First-In, First-Out) structure. Default underlying container is std::deque.
Header: <queue>

Function	Description	Complexity
push(val)	Adds element to the back.	O(1)
pop()	Removes element from the front.	O(1)
front()	Accesses the front element.	O(1)
back()	Accesses the back element.	O(1)
empty()	Checks if the queue is empty.	O(1)
size()	Returns the number of elements.	O(1)

Example:

```
queue<int> q;
q.push(1); // q: {1}
q.push(2); // q: {1, 2}
cout << q.front(); // prints 1
q.pop(); // q: {2}
```


std::priority_queue

A queue where the element with the highest "priority" is always at the front. Implemented as a max-heap by default.
Header: <queue>

Function	Description	Complexity
push(val)	Inserts an element.	O(log N)
pop()	Removes the top priority element.	O(log N)
top()	Accesses the top priority element.	O(1)
empty()	Checks if empty.	O(1)
size()	Returns the number of elements.	O(1)

Example (Max-Heap):

```
priority_queue<int> pq; // Max-heap
pq.push(10);
pq.push(30);
pq.push(20);
// pq contains {30, 10, 20} internally, but top is always max
cout << pq.top(); // prints 30
pq.pop(); // removes 30
cout << pq.top(); // prints 20
```

Example (Min-Heap):

```
priority_queue<int, vector<int>, greater<int>> min_pq;
min_pq.push(10);
min_pq.push(30);
min_pq.push(5);
cout << min_pq.top(); // prints 5
```


6. Common Algorithms (<algorithm>)

These functions operate on ranges of elements, typically specified by iterators.

Function	Description
sort(begin, end)	Sorts the range [begin, end).
sort(begin, end, comp)	Sorts using a custom comparator.
stable_sort(begin, end)	Sorts, preserving relative order of equal elements.
reverse(begin, end)	Reverses the order of elements in the range.
max_element(begin, end)	Returns an iterator to the largest element.
min_element(begin, end)	Returns an iterator to the smallest element.
accumulate(begin, end, init)	Sums up elements in a range. (Requires <numeric>)
count(begin, end, val)	Counts occurrences of val.
find(begin, end, val)	Finds the first occurrence of val.
binary_search(begin, end, val)	Checks if val exists in a **sorted** range.
lower_bound(begin, end, val)	Finds first element not less than val in a **sorted** range.
upper_bound(begin, end, val)	Finds first element greater than val in a **sorted**

	range.
next_permutation(begin, end)	Generates the next lexicographically greater permutation.
prev_permutation(begin, end)	Generates the next lexicographically smaller permutation.

Example:

vector<int> v = {4, 2, 5, 1, 3};
sort(v.begin(), v.end()); // v is {1, 2, 3, 4, 5}
reverse(v.begin(), v.end()); // v is {5, 4, 3, 2, 1}
int sum = accumulate(v.begin(), v.end(), 0); // sum is 15
bool has_3 = binary_search(v.begin(), v.end(), 3); // false, because v is not sorted

7. String Manipulation (<string>)

std::string is a sequence of characters.

Function	Description
s.length(), s.size()	Returns the length of the string.
s.push_back(char)	Appends a character.
s.pop_back()	Removes the last character.
s.substr(pos, len)	Returns a substring of length len starting at pos.
s.find(str2)	Finds the first occurrence of str2 in s. Returns string::npos if not found.
s.replace(pos, len, str2)	Replaces a part of the string.
stoi(s), stoll(s)	Converts string to int or long long.
to_string(num)	Converts a number to a string.

Example:

```
string s = "hello";
s += " world"; // s is "hello world"
cout << s.substr(0, 5); // prints "hello"
size_t pos = s.find("world"); // pos will be 6
if (pos != string::npos) {
   s.replace(pos, 5, "C++"); // s is "hello C++"
}
```


8. Time & Space Complexity Summary

Data Structure Operations

Data Structure	Access	Search	Insertion	Deletion	Space
array	O(1)	O(N)	-	-	O(N)
vector	O(1)	O(N)	O(N)*	O(N)*	O(N)
deque	O(1)	O(N)	O(N)*	O(N)*	O(N)
list	O(N)	O(N)	O(1)	O(1)	O(N)
set / map	-	O(log N)	O(log N)	O(log N)	O(N)
unordered_set / map	-	O(1) avg	O(1) avg	O(1) avg	O(N)
stack / queue	O(1)	-	O(1)	O(1)	O(N)
priority_queue	O(1) (top)	-	O(log N)	O(log N)	O(N)

**Insertion/Deletion at the end of a vector or either end of a deque is amortized O(1).*

Common Sorting Algorithms

Algorithm	**Time Complexity (Best / Avg / Worst)**	**Space Complexity**
std::sort (Introsort)	O(N log N) / O(N log N) / O(N log N)	O(log N)
std::stable_sort (Merge Sort)	O(N log N) / O(N log N) / O(N log N)	O(N) or O(log N)
Bubble Sort	O(N) / $O(N^2)$ / $O(N^2)$	O(1)
Insertion Sort	O(N) / $O(N^2)$ / $O(N^2)$	O(1)
Selection Sort	$O(N^2)$ / $O(N^2)$ / $O(N^2)$	O(1)
Heap Sort	O(N log N) / O(N log N) / O(N log N)	O(1)
Quick Sort	O(N log N) / O(N log N) / $O(N^2)$	O(log N)