**Data Structures**

There are many data structures in C++ that can be divided into two main categories: standard data structures provided by the C++ Standard Library, and custom data structures that programmers can create themselves for specific tasks. Here are some of the most commonly used data structures in C++.

***Standard data structures:***

***Sequential containers:***

std::vector: A dynamic array that allows you to quickly access elements and efficiently add/remove elements to the end of the container.

std::deque: A two-way queue that supports fast insertion and deletion of items both at the beginning and at the end.

std::array: A fixed-size static array that provides array functionality with the security and convenience of standard containers.

std::list: A doubly linked list that allows you to quickly add and remove items at any position in the list.

***Associative containers:***

std::set: A set that stores unique elements in sorted order.

std::map: An associative array storing key-value pairs in sorted order; each key is unique.

std::multiset: A set that allows storing non-unique elements in a sorted order.

std::multimap: An associative array for storing key-value pairs, where keys may not be unique.

***Unordered associative containers:***

std::unordered\_set: A set using a hash table to store unique elements without preserving order.

std::unordered\_map: A hash table for storing key-value pairs in no particular order; each key is unique.

std::unordered\_multiset and std::unordered\_multimap: std::unordered\_set and std::unordered\_map are similar, but they allow storing non-unique keys.

***Container adapters:***

std::stack: A container adapter that implements a stack-type data structure.

std::queue: Container adapter for queue implementation.

std::priority\_queue: A priority queue where items are retrieved according to their priority.

**The interfaces and complexity of the algorithms**

(most popular in competitive programming)

***Array:***

get\_element\_at(i): complexity O(1)

set\_element\_at(i, value): complexity O(1)

***List:***

prepend: O(1)

insert\_after: O(1)

remove\_first/remove\_after: O(1)

get\_nth/set\_nth: O(N)

***Vector:***

prepend: O(N)

append: Amort.O(1)

insert\_after: O(N)

remove\_first/remove\_after: O(1)

get\_nth/set\_nth: O(1)

**Trees**

A tree is a hierarchical data structure consisting of nodes connected by edges. The main types of trees include:

- Binary search trees (BST), where each node has up to two children, and the condition is met for all nodes: all elements in the left subtree are smaller than the current node, and in the right subtree are larger.

- AVL trees and red-black trees are balanced versions of binary search trees that provide higher performance of search, insert and delete operations.

- Heaps (in particular binary heaps) that can be used to implement priority queues.

- Ternary trees, B-trees, and other variants optimized for specific applications, such as databases and file systems.

**Graphs**

A graph consists of a set of nodes (or vertices) and edges connecting these nodes. Graphs can be directed and undirected, and can also contain edges with weight (weighted graphs). They are used to model various physical, social and information networks. Graphs are usually implemented using adjacency lists or adjacency matrices.

**Heaps**

A heap is a specialized tree-based data structure that satisfies the heap property: if B is a descendant of node A, then key(A) precedes key(B). This provides quick access to the node with the highest (or lowest) priority. Heaps are usually used to implement priority queues.

Although the C++ standard library does not provide direct support for trees, graphs, and heaps (with the exception of std::priority\_queue), these data structures can be implemented using standard containers (for example, std::vector for storing nodes and edges) or by defining custom classes and structures. There are many manuals and libraries on the Internet for working with these data structures in C++.