Standard Template Library (STL) - C++

Basics of Ds & Algo

This study can be broadly classified in two groups - **Data** and **functions** which modify the data

1. To store data efficiently we need **Data Structures**.
2. To perform action on stored data we need **Algorithms.**

Basics of DS & Algo

For example; Consider you are given a notebook which has page numbers written at the bottom of every page. Now your task is to find a particular page. Imagine two scenarios:

1. Page numbers are not sorted [Inefficient data structure. Efficient Algorithm] No matter how good your algorithm is, it cannot predict where the required page lies. It is going to take long time in worst/average case]
2. Page numbers are sorted [Efficient data structure, Inefficient algorithm]. Let's say we give the book to someone who starts iterating pages one by one from the start. So the algorithm is going to take a lot of time, correct? Ideally what that should have done is use binary search.

We will judge a DS on the basis of - Insert, Delete, and Find. Based on these parameters, every data structure has its own pros and cons.

1. Containers
2. Iterators
3. Algorithms

1. Containers
2. Iterators
3. Algorithms

Types of containers

- Sequence Containers
  - Array (Array)
  - Vector (ArrayList)
  - Deque (Deque)

- Associative Containers (Ordered/Unordered)
  - Set (Treeset)
  - Map (TreeMap)
  - Unordered Set (Hashset)
  - Unordered Map (Hashmap)

- Container Adapters
  - Stack (Stack)
  - Queue (Queue)
  - Priority Queue (Priority Queue)

Sequence containers

Array - Primitive data type. Fixed size containers.

Vector - Dynamic/Variable size containers. Arrays that can change in size

Deque - Doubly ended queue are containers with dynamic/variable sizes that can be expanded or contracted on both ends.

Container Adaptors

Stack - Supports LIFO. i.e. Elements are inserted and popped only from one end of the container. For ex: Washing piles of plates.

Queue - Supports FIFO. i.e Elements are inserted into one end of the container and popped from the other. For ex.: Movie ticket lane.

Priority Queue - Implemented as a heap. Depending upon implementation it provides the largest/smallest element when popped.

Associative containers

Set - Stores unique elements (no duplicates) in sorted order. Sorting can be either increasing, decreasing or even user defined way

Map - Stores elements formed by a combination of a key value pair. Elements are sorted according to keys and can be either increasing, decreasing or even user defined. For example: Your roll number in your universities.

Unordered Set - Similar to set but elements are not sorted.

Unordered Map- Similar to map but elements are not sorted.

1. Containers
2. Iterators
3. Algorithms

Iterators

Consider this scenario: You have an audio file on your laptop which you want to play. How will you open it?

1. Your file is on the disk in a specific format [It can have header, metadata info, audio data etc, etc] You can learn how to read binary data and learn formatting of that file and then read it.
2. Or you can simply use a VLC media player to play.

Iterators

- Similarly, a container (data structure) can be stored on a disk in a different manner for the purpose of efficiency. You don’t need to reinvent the wheel to read what value the given container (DS) holds. You can use something called **iterators**.
- In simple words, Iterators are used to traverse from one element to another element in a container. It is a concept common to both C++ and Java. In case of C++ iterators are pointers however in case of Java iterators are objects.

1. Containers
2. Iterators
3. Algorithms

Algorithms

- Searching
- Sorting
- Backtracking
- Divide and Conquer
- Greedy
- Dynamic Programming
- Pattern Searching
- Graph Algorithms

Deep Dive into Containers

Vectors

- **Insert**
  - push_back()
  - insert()
- **Iterators**
  - begin() and end()
  - rbegin() and rend()
- **Delete**
  - pop_back()
  - erase()
  - clear()
- **Access**
  - [] operator
  - at()
- **Capacity**
  - size()
  - capacity()

Stack

- **Insert**
  - push()
- **Iterators**
  - top()
- **Delete**
  - pop()
  - clear()
- **Capacity**
  - size()
  - empty()

Queue

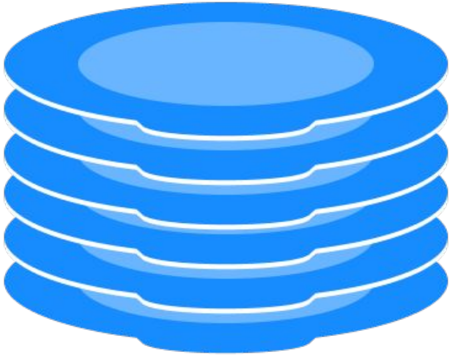
- **Insert**
  - push()
- **Iterators**
  - front()
- **Delete**
  - pop()
  - clear()
- **Capacity**
  - size()
  - empty()

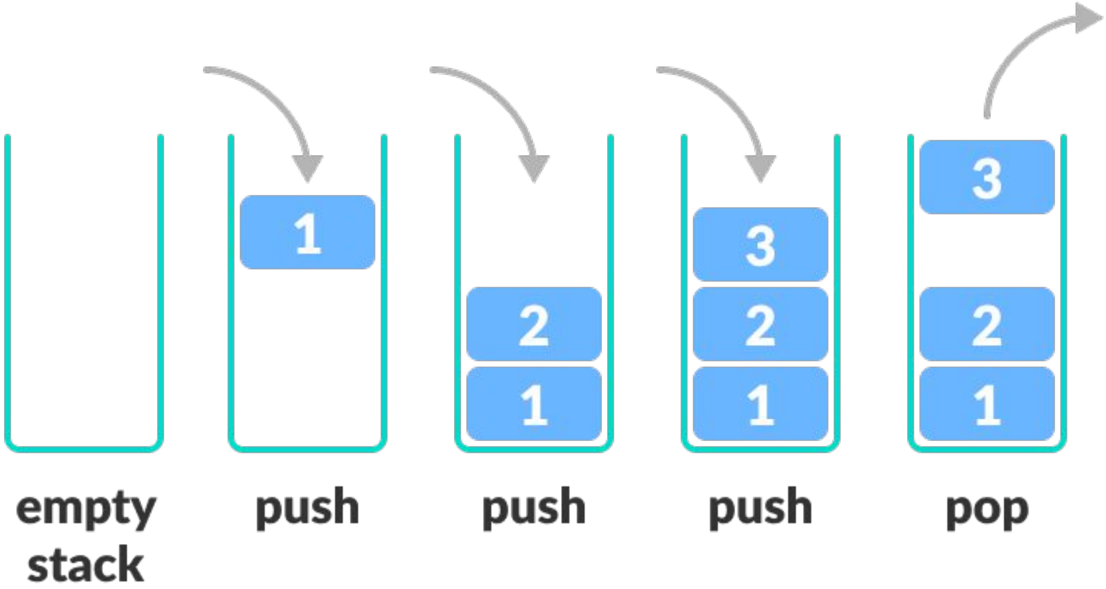
Priority Queue

- **Insert**
  - push()
- **Iterators**
  - top()
- **Delete**
  - pop()
  - clear()
- **Capacity**
  - size()
  - empty()

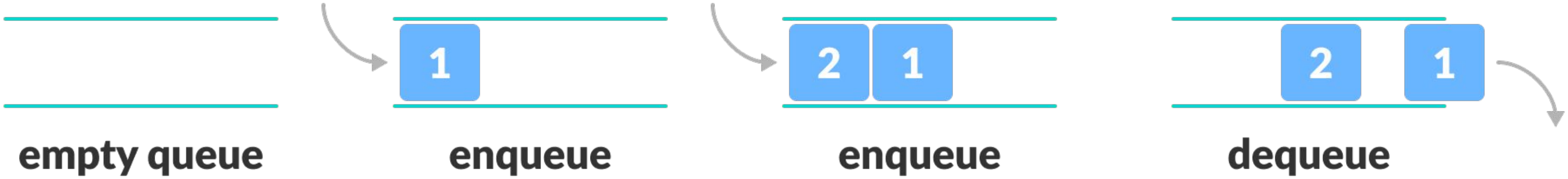
Note: Stack, Queue and Priority Queue do not support iterators as such.
However there are methods to fetch the top element.

Stack





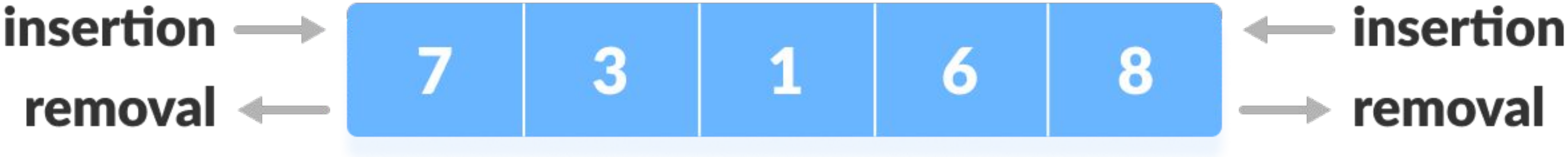
QUEUE



Deque - Doubly Ended queue

- **Insert**
  - push_back()
  - push_front()
  - insert()
- **Iterators**
  - begin() and end()
  - rbegin() and rend()
- **Delete**
  - pop_back()
  - pop_front()
  - erase()
  - clear()
- **Access**
  - [] operator
  - at()
  - front()
  - back()
- **Capacity**
  - size()
  - empty()
  - capacity()

DEQUE



Set/Unordered Set - Collection of Unique elements

- **Insert**
  - insert()
- **Iterators**
  - begin() and end()
  - rbegin() and rend()
- **Delete**
  - erase()
  - clear()

- **Capacity**
  - size()
  - empty()
- **Operations**
  - find()
  - lower_bound()
  - upper_bound()

Map/unordered map- Collection of Key Value Pair

- **Insert**
  - insert()
- **Iterators**
  - begin() and end()
  - rbegin() and rend()
- **Delete**
  - erase()
  - clear()

- **Access**
  - [] operator
  - at()
- **Capacity**
  - size()
  - empty()
- **Operations**
  - find()
  - lower_bound()
  - upper_bound()

unordered Set(Contd...)

- **Buckets**
  - bucket_count()
  - bucket_size()
- **Hash Policy**
  - load_factor()
  - max_load_factor()
  - rehash()
- **Observer**
  - hash_function()
  - key_eq()

unordered map (Contd...)

- **Buckets**
  - bucket_count()
  - bucket_size()
- **Hash Policy**
  - load_factor()
  - max_load_factor()
  - rehash()
- **Observer**
  - hash_function()
  - key_eq()

We’ll dive into these functions more once we learn about hashing.