Standard Template Library

C++ STL

- Love Babbar

Standard Template Library

C++ STL

- Love Babbar

What is STL ?

- C++ STL stands for the "Standard Template Library." It is a powerful and extensive collection of template classes and functions in the C++ programming language that provides general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures.

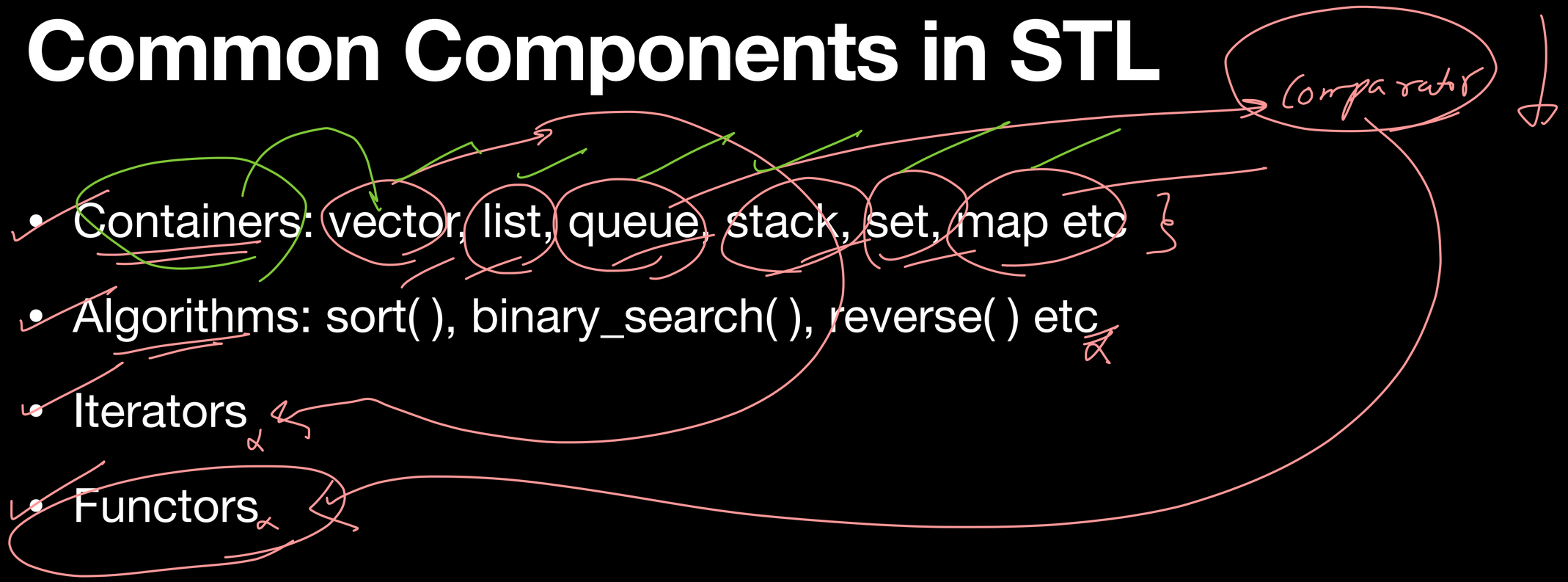
- The C++ STL is an essential part of the C++ Standard Library, and it plays a crucial role in simplifying and accelerating C++ software development. It offers a wide range of data structures (containers) and algorithms that can be easily used in C++ programs, allowing developers to focus on solving specific problems rather than reinventing the wheel for common tasks.

Why we need STL ?

The C++ Standard Template Library (STL) is a critical component of the C++ Standard Library, and it serves several important purposes in C++ programming. Here are some key reasons why we need STL:

- Efficiency: STL provides highly optimized and efficient implementations of commonly used data structures and algorithms. These implementations are typically developed and tested by experts, ensuring that they perform well in various scenarios. Using STL can save you the time and effort required to implement these data structures and algorithms from scratch.
- Productivity: STL promotes code reuse and reduces the need to reinvent the wheel. It offers a wide range of container classes (like vectors, lists, sets, and maps) and algorithms (such as sorting and searching) that can be readily used in your programs. This leads to faster development and more maintainable code.
- Consistency: STL provides a standardized and consistent interface for working with data structures and algorithms. This consistency makes it easier to learn and use the library effectively. Once you understand how to use one STL container or algorithm, you can apply similar knowledge to others.
- Portability: Code written using STL is generally more portable across different compilers and platforms. The library is part of the C++ standard, ensuring that it is available and behaves consistently across compliant compilers.
- Safety: STL helps reduce common programming errors, such as buffer overflows and memory leaks, by encapsulating low-level memory management and providing safer alternatives. For example, smart pointers in STL (unique_ptr, shared_ptr) help manage memory automatically, reducing the risk of memory-related issues.
- Maintainability: Code using STL tends to be more maintainable because it leverages well-tested and standardized components. This makes it easier for developers to understand and modify code written by others or by their past selves.
- Performance Optimization: STL algorithms are designed to work efficiently with different container types, and they often outperform hand-rolled implementations. This allows you to focus on your application's logic rather than low-level performance optimizations.
- Expressiveness: STL promotes expressive code by providing a high-level interface for working with data structures and algorithms. This can lead to more readable and concise code, making your programs easier to understand and maintain.
- Community and Resources: STL has a vast user community, which means there are numerous tutorials, books, and online resources available to help you learn and use it effectively. You can tap into this wealth of knowledge to improve your C++ programming skills.

Common Components in STL



- Containers: vector, list, queue, stack, set, map etc
- Algorithms: sort( ), binary_search( ), reverse( ) etc
- Iterators
- Functors

Containers:

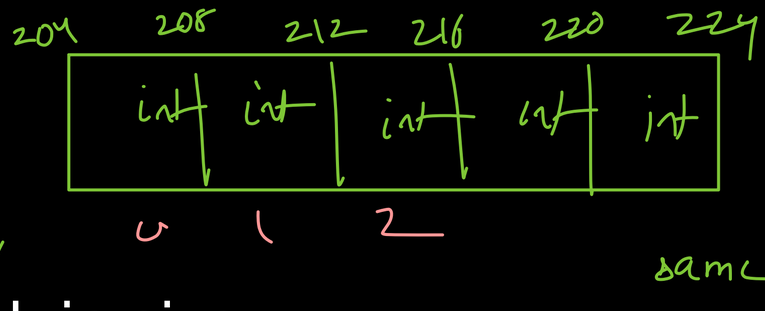
Containers in C++ STL are classes or data structures that are designed to store and manage collections of objects. They provide a standardized way to store, retrieve, and manipulate data in various ways. C++ STL provides a variety of container classes, each with its own characteristics and use cases.

→ Data Structure

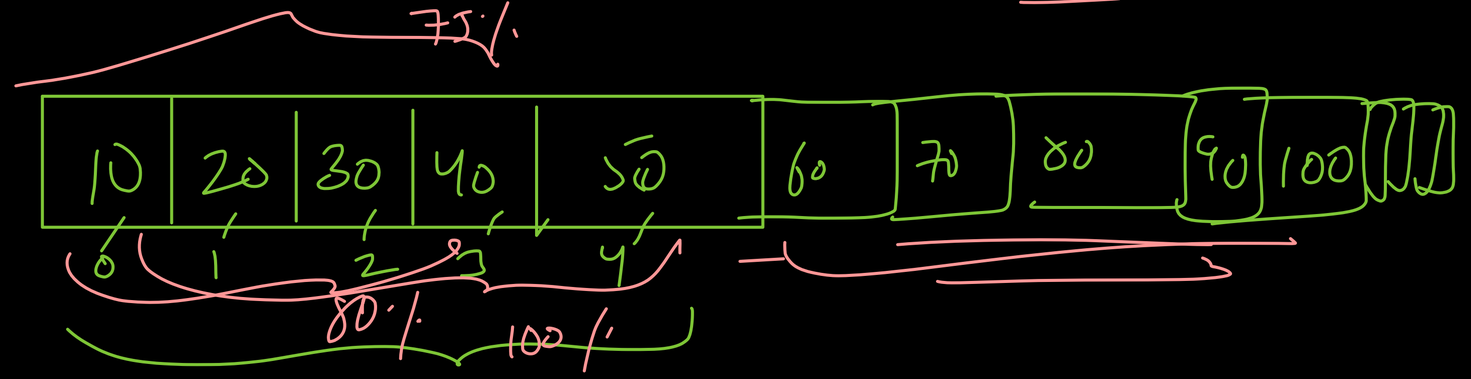
int arr [5]

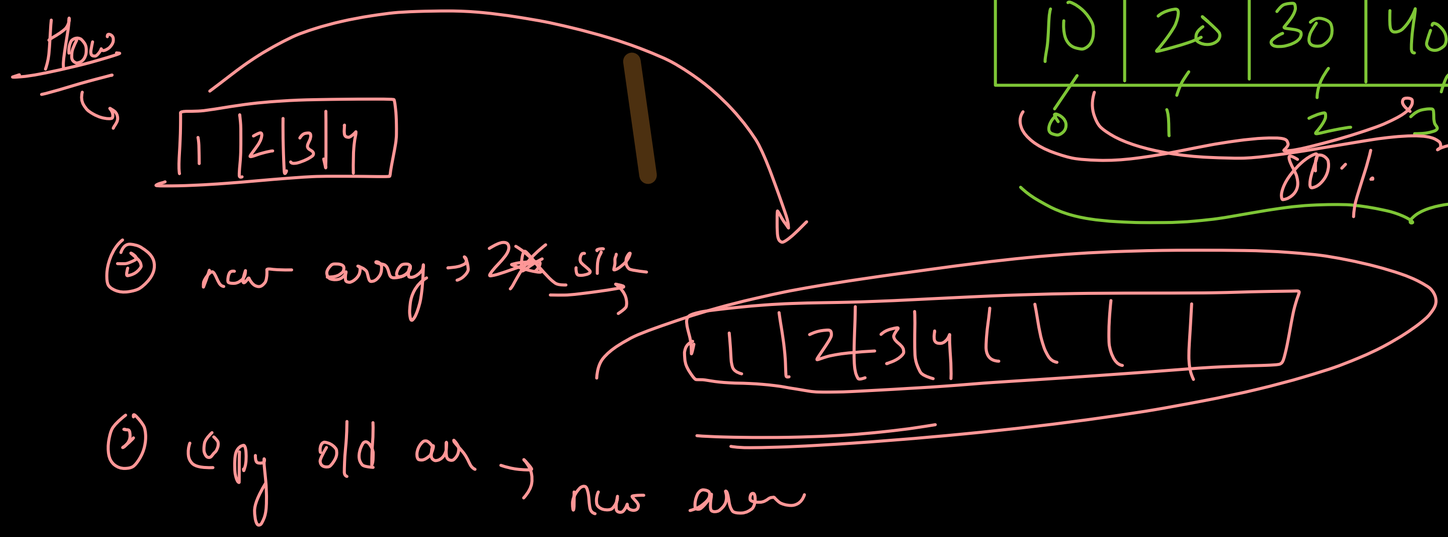
int arr [] = {2, -1, -1, -3}

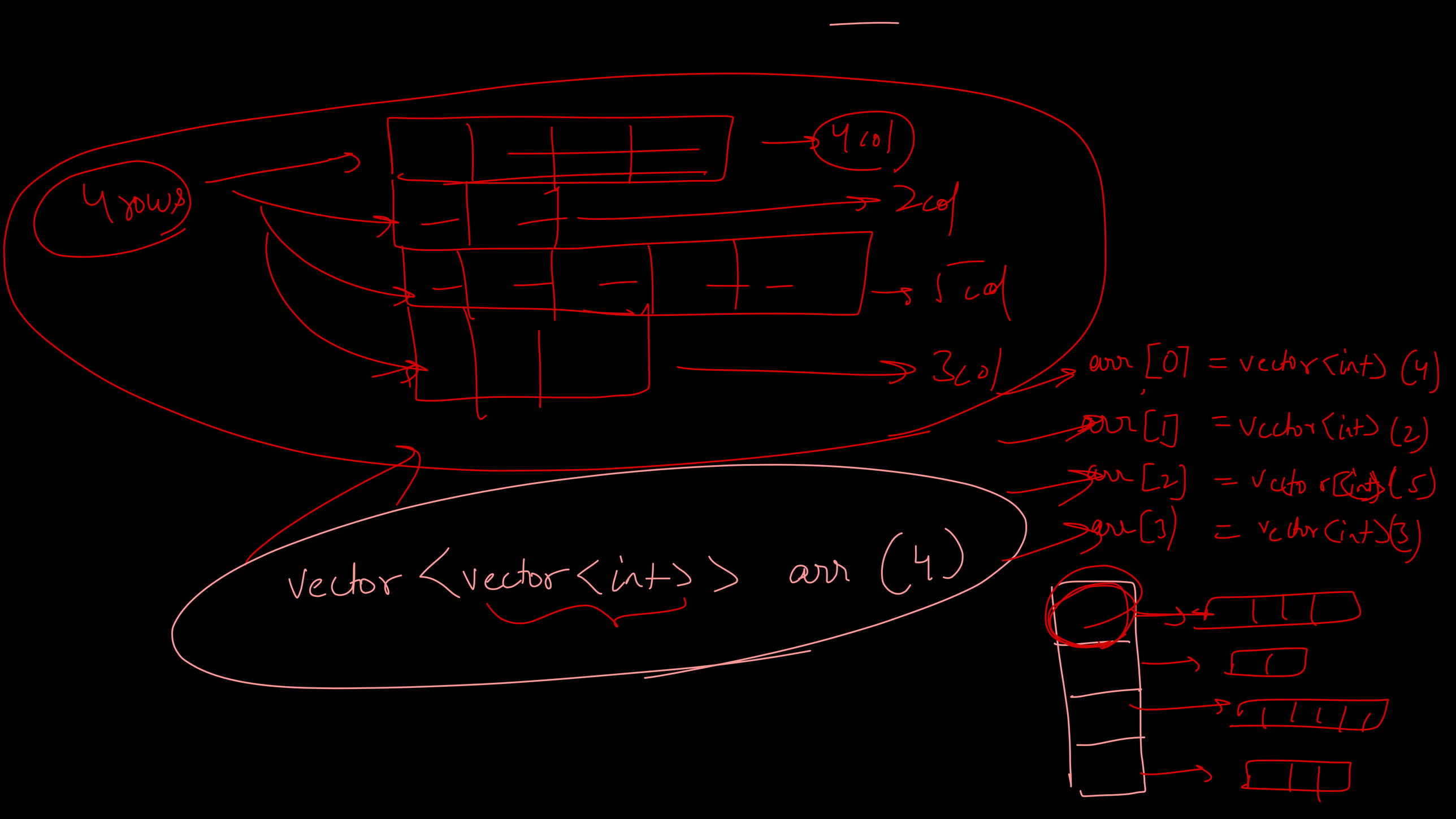
Vector :



- Dynamic array that can grow or shrink in size.
- Allows fast random access to elements.
- Insertion and removal of elements at the end is efficient.
- Suitable for most scenarios when elements need to be stored in a linear sequence.

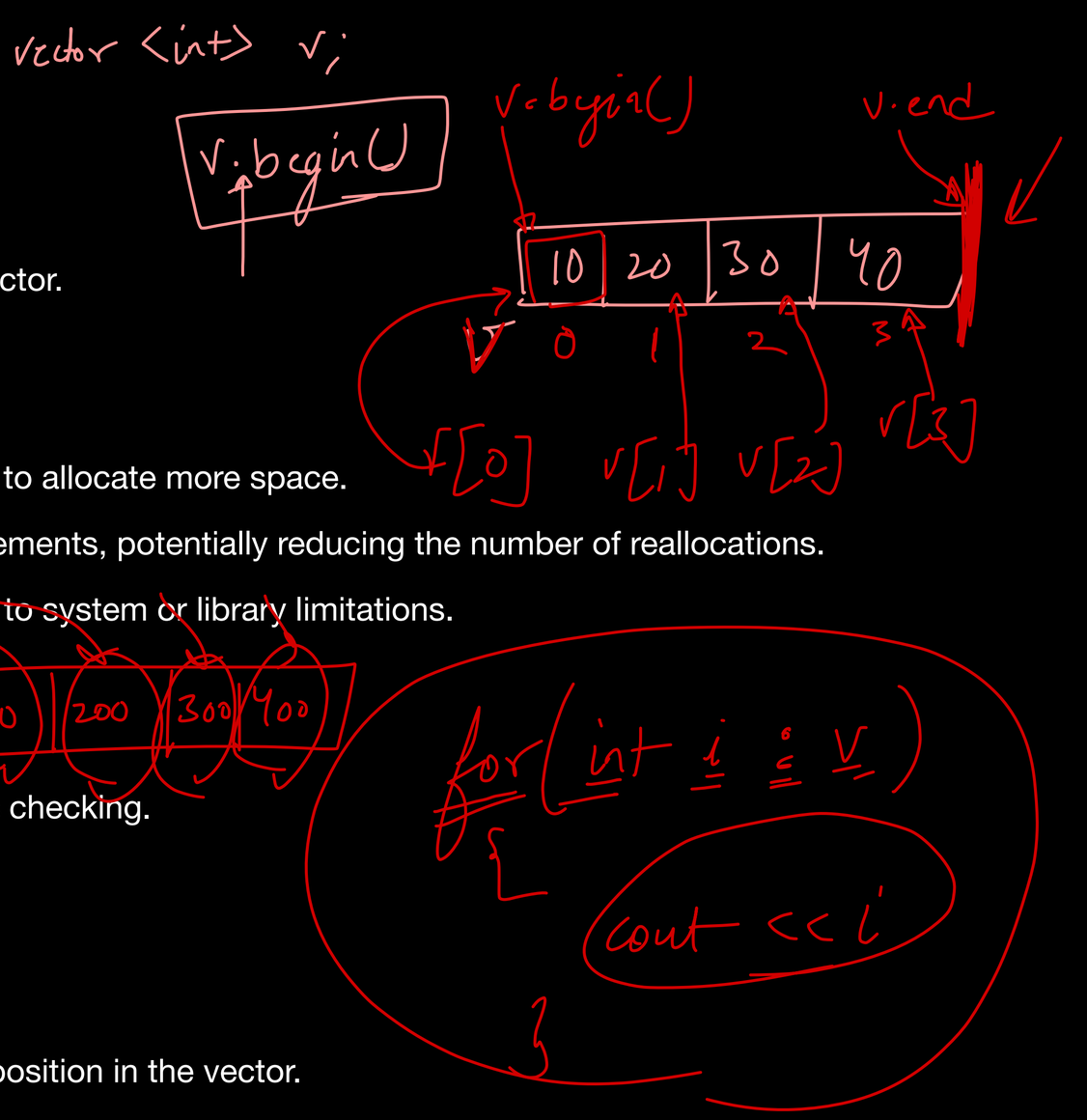




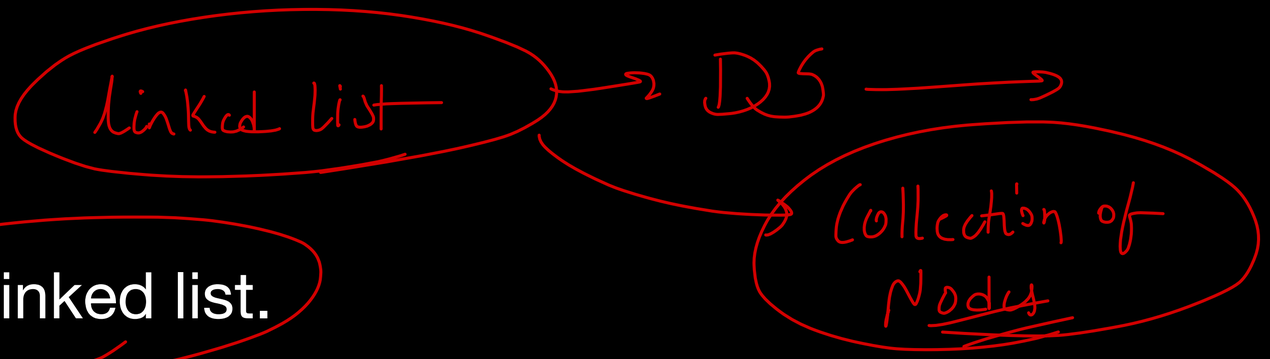

4 rows
4 col
2 col
5 col
3 col
vector<vector<int>> arr (4)
arr[0] = vector<int> (4)
arr[1] = vector<int> (2)
arr[2] = vector<int> (5)
arr[3] = vector<int> (3)

Member Functions:

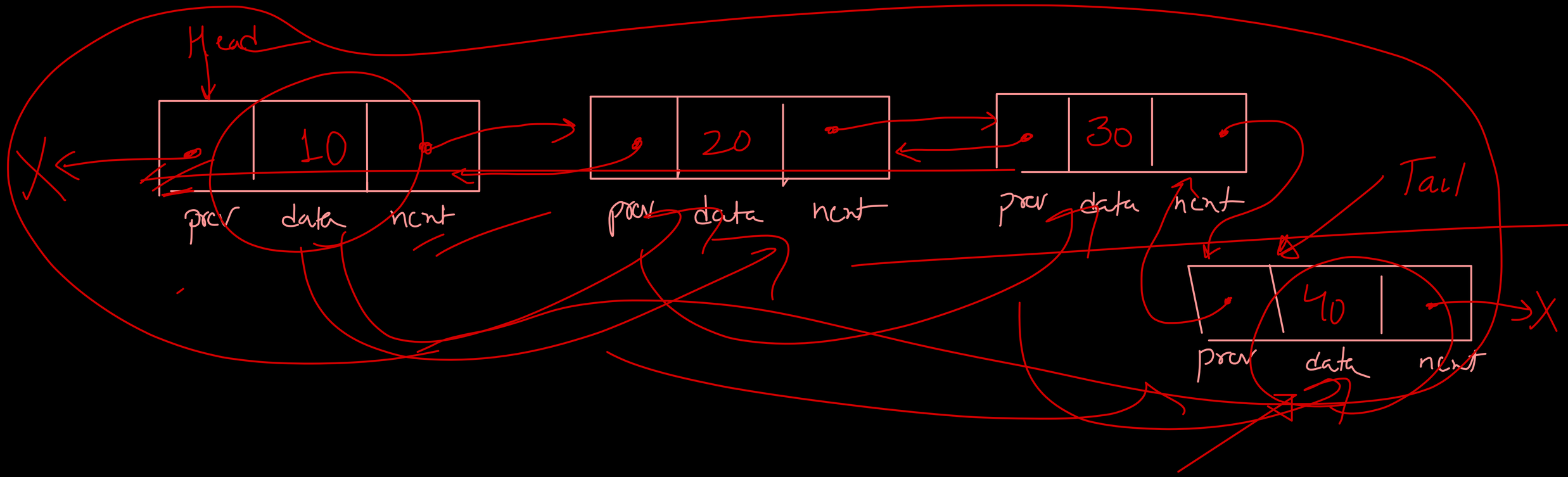
- begin(): Returns an iterator pointing to the first element in the vector.
- end(): Returns an iterator pointing to the position just after the last element in the vector.
- size(): Returns the number of elements in the vector.
- empty(): Checks if the vector is empty (i.e., whether its size is 0).
- capacity(): Returns the number of elements that the vector can hold before needing to allocate more space.
- reserve(size_type n): Requests that the vector capacity be increased to at least n elements, potentially reducing the number of reallocations.
- max_size(): Returns the maximum number of elements that the vector can hold due to system or library limitations.
- front(): Accesses the first element in the vector.
- back(): Accesses the last element in the vector.
- operator[](size_type n): Accesses the element at the specified index without bounds checking.
- at(size_type n): Accesses the element at the specified index with bounds checking.
- push_back(const T& value): Adds an element to the end of the vector.
- pop_back(): Removes the last element from the vector.
- insert(iterator position, const T& value): Inserts a new element before the specified position in the vector.
- erase(iterator position) or erase(iterator first, iterator last): Removes one or more elements from the vector starting at the specified position.
- clear(): Removes all elements from the vector, which are destroyed, and leaves it with a size of 0.
- swap(vector& x): Swaps the contents of the vector with those of another vector of the same type, including their sizes and capacities.



List :

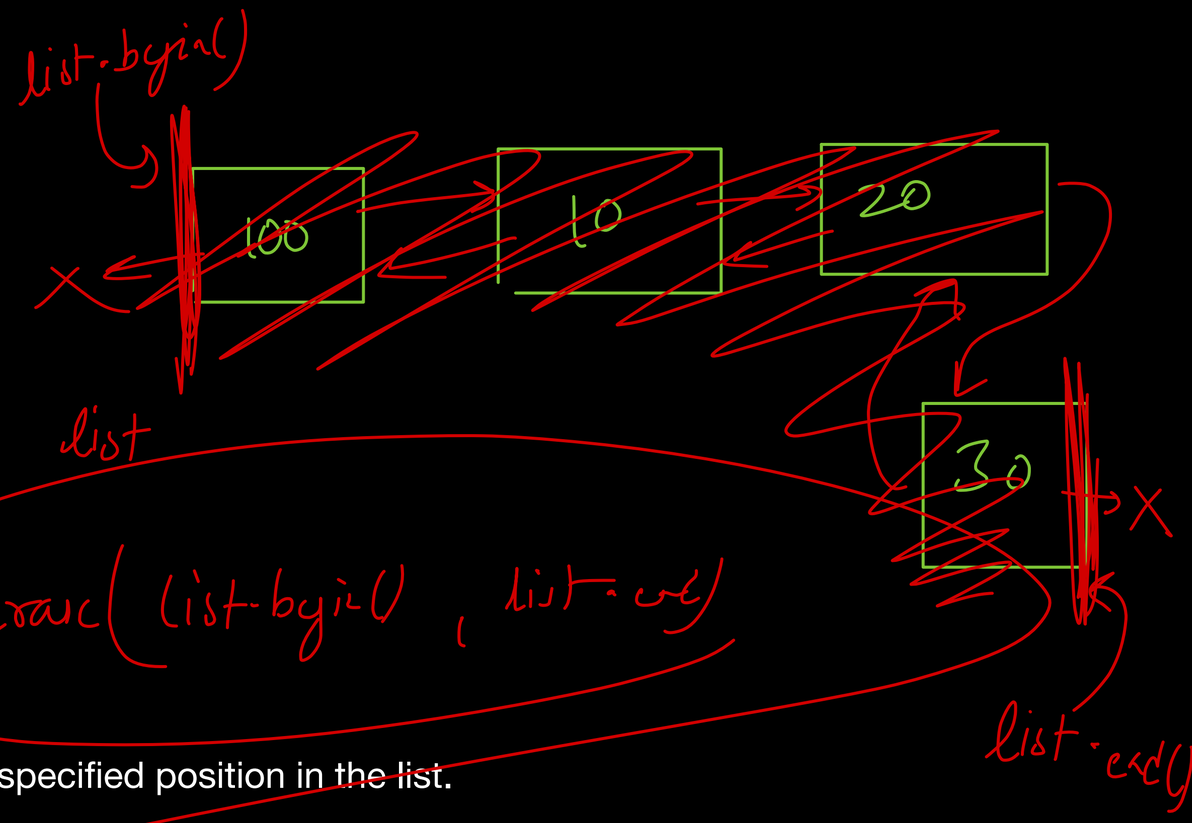


- Doubly-linked list.
- Allows fast insertions and removals anywhere in the list.
- No random access like vectors



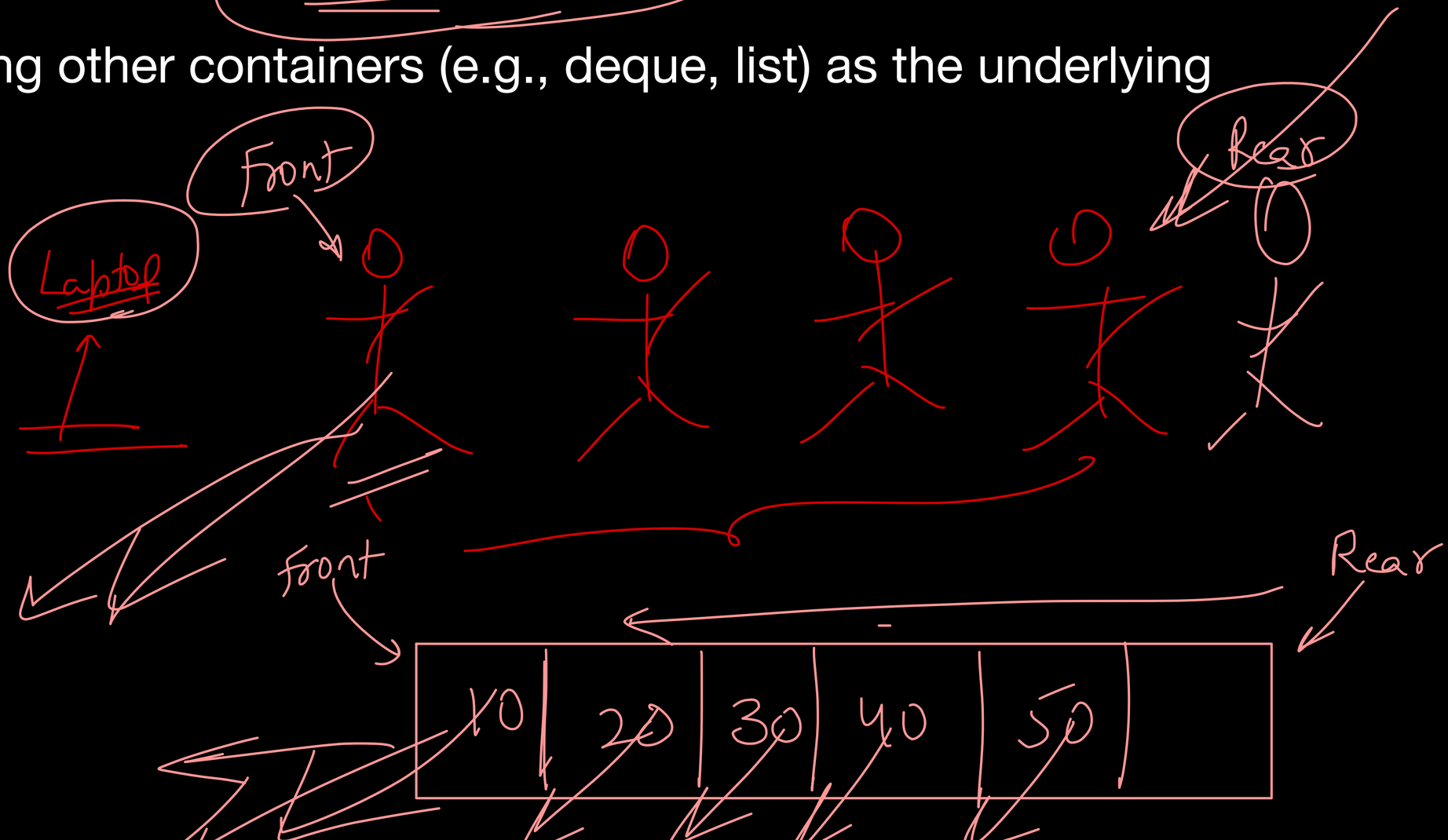
Member Functions:

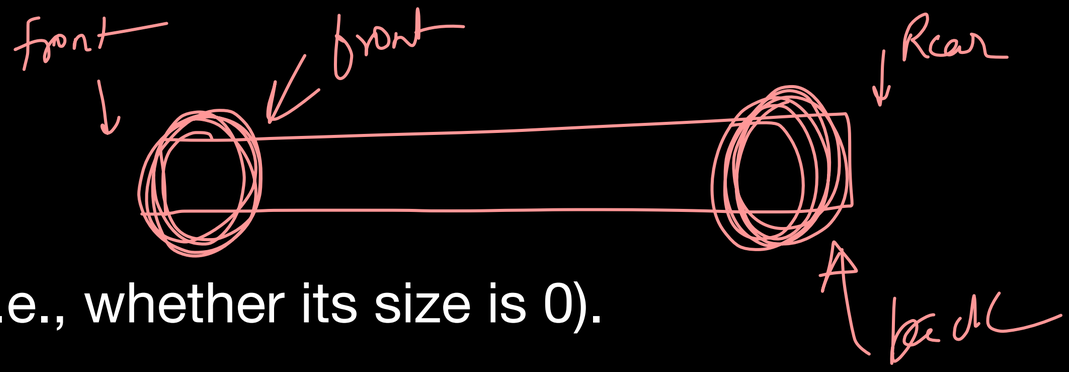
- begin(): Returns an iterator pointing to the first element in the list.
- end(): Returns an iterator pointing to the past-the-end element in the list.
- size(): Returns the number of elements in the list.
- empty(): Checks if the list is empty (i.e., whether its size is 0).
- front(): Accesses the first element in the list.
- back(): Accesses the last element in the list.
- push_back(const T& value): Adds an element to the end of the list.
- pop_back(): Removes the last element from the list.
- insert(iterator position, const T& value): Inserts a new element before the specified position in the list.
- erase(iterator position) or erase(iterator first, iterator last): Removes one or more elements from the list starting at the specified position.
- clear(): Removes all elements from the list, which are destroyed, and leaves it with a size of 0.
- swap(list& x): Swaps the contents of the list with those of another list of the same type, including their sizes.
- pop_front(): Removes the first element from the list.
- push_front(const T& value): Adds an element to the beginning of the list.
- remove(const T& value): Removes all elements from the list that are equal to the specified value.



Queue:

- Adaptor class that provides a First-In, First-Out (FIFO) data structure.
- Implemented using other containers (e.g., deque, list) as the underlying storage.





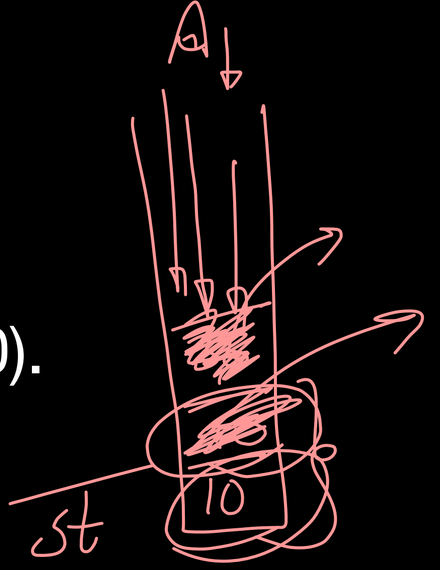
Member Functions:

- empty(): Checks if the queue is empty (i.e., whether its size is 0).
- size(): Returns the number of elements in the queue.
- front(): Accesses the first element in the queue, which is the next element to be removed.
- back(): Accesses the last element in the queue, which is the most recently added element.
- push(const T& value): Adds an element to the end of the queue.
- pop(): Removes the first element from the queue.
- swap(queue& x): Swaps the contents of the queue with those of another queue of the same type.

Stack:

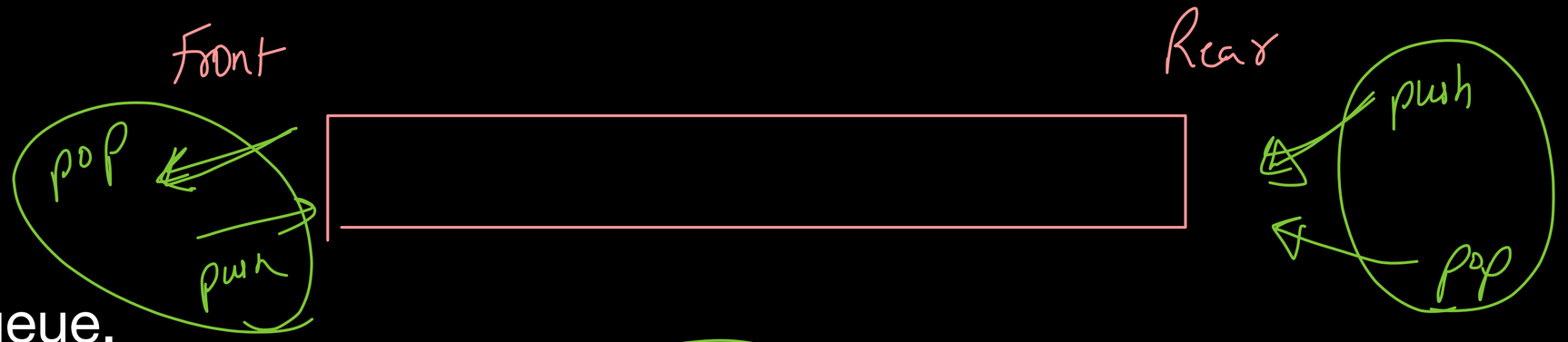
- Adaptor class that provides a Last-In, First-Out (LIFO) data structure.
- Implemented using other containers (e.g., vector, deque, list) as the underlying storage.

Member Functions:

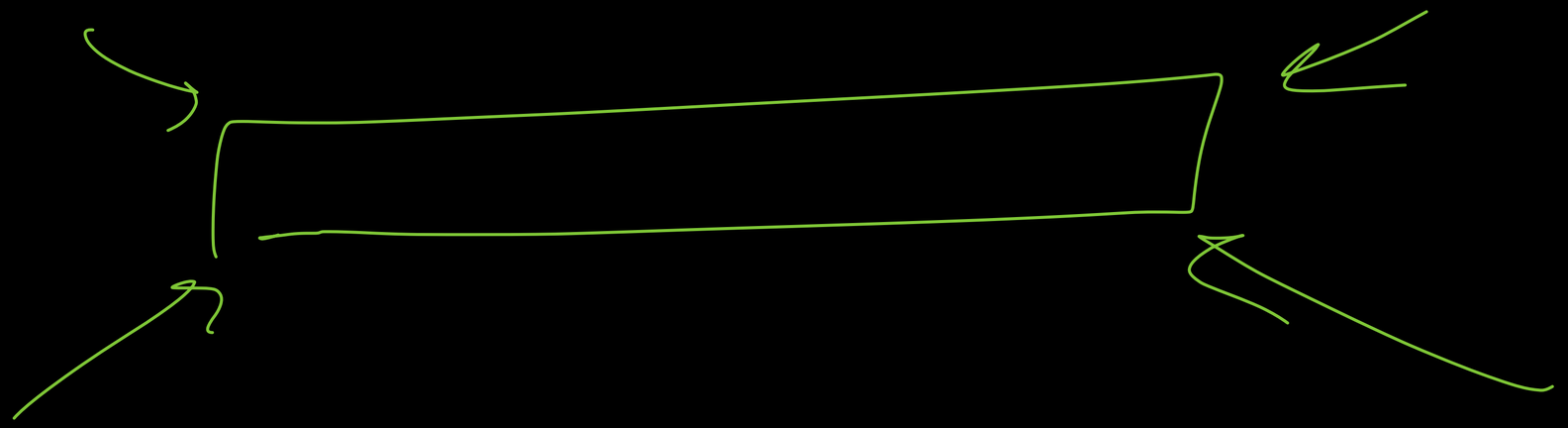


- empty(): Checks if the stack is empty (i.e., whether its size is 0).
- size(): Returns the number of elements in the stack.
- top(): Accesses the top element of the stack, which is the most recently added element.
- push(const T& value): Adds an element to the top of the stack.
- pop(): Removes the top element from the stack.
- swap(stack& x): Swaps the contents of the stack with those of another stack of the same type.

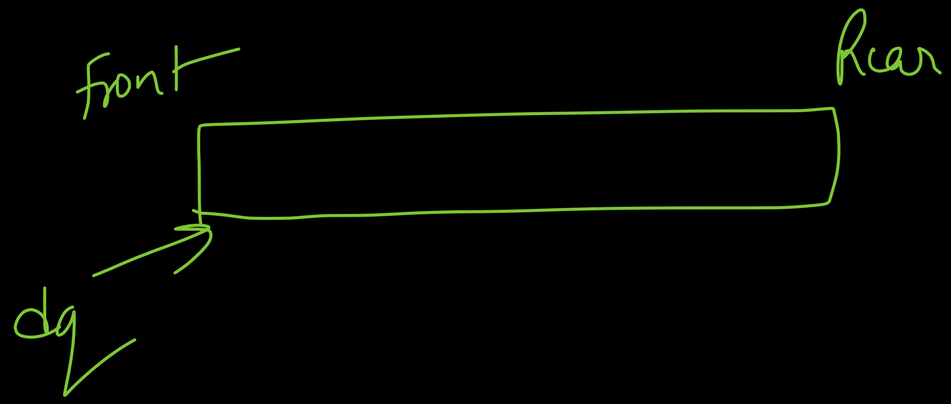
Deque:



- Double-ended queue.
- Similar to vectors but allows efficient insertion and removal at both ends.
- Suitable when elements need to be inserted or removed frequently from the front or back.



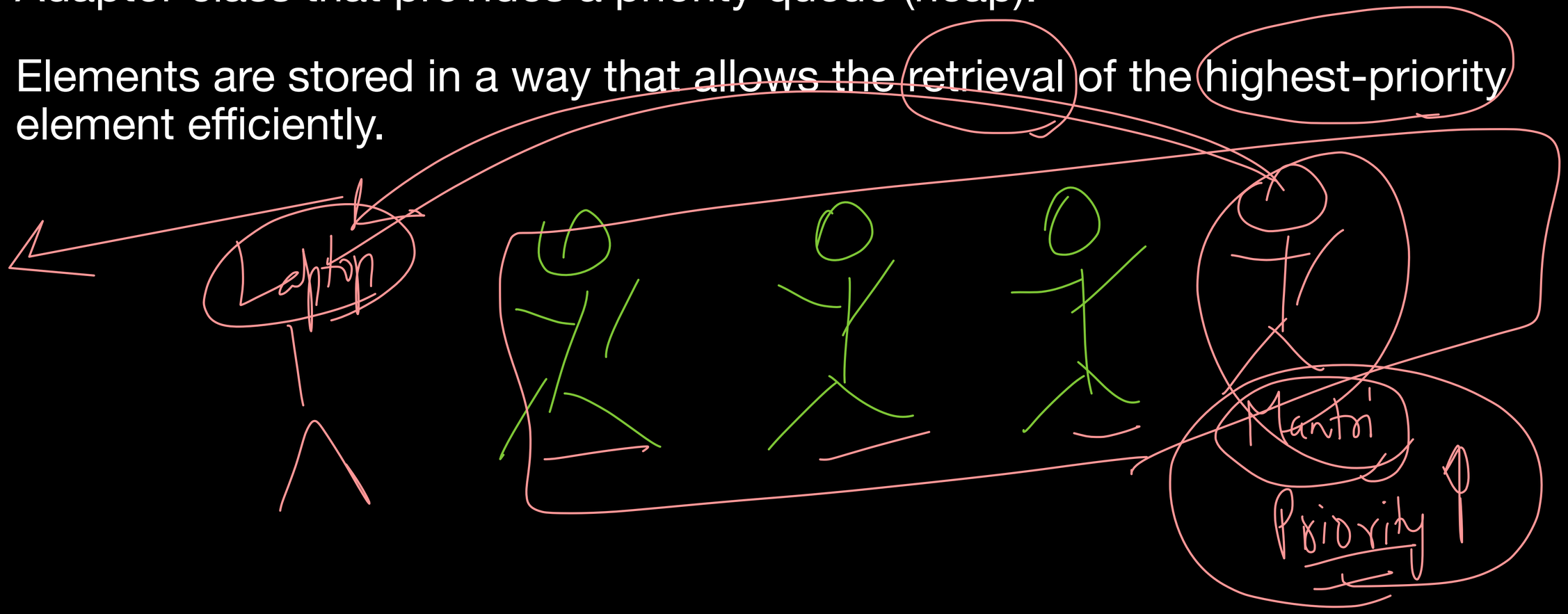
Member Functions:



- begin(): Returns an iterator pointing to the first element in the deque.
- end(): Returns an iterator pointing to the past-the-end element in the deque.
- size(): Returns the number of elements currently in the deque.
- empty(): Checks if the deque is empty (i.e., whether its size is 0).
- front(): Accesses the first element in the deque.
- back(): Accesses the last element in the deque.
- operator[](size_type n): Accesses the element at the specified index without bounds checking.
- at(size_type n): Accesses the element at the specified index with bounds checking.
- push_back(const T& value): Adds an element to the end of the deque.
- pop_back(): Removes the last element from the deque. pop_front(): Removes the first element from the deque.
- push_front(const T& value): Adds an element to the beginning of the deque.
- insert(iterator position, const T& value): Inserts a new element before the specified position in the deque.
- erase(iterator position) or erase(iterator first, iterator last): Removes one or more elements from the deque starting at the specified position.
- clear(): Removes all elements from the deque, which are destroyed, and leaves it with a size of 0.
- swap(deque& x): Swaps the contents of the deque with those of another deque of the same type, including their sizes.

Priority queue:

- Adaptor class that provides a priority queue (heap).
- Elements are stored in a way that allows the retrieval of the highest-priority element efficiently.



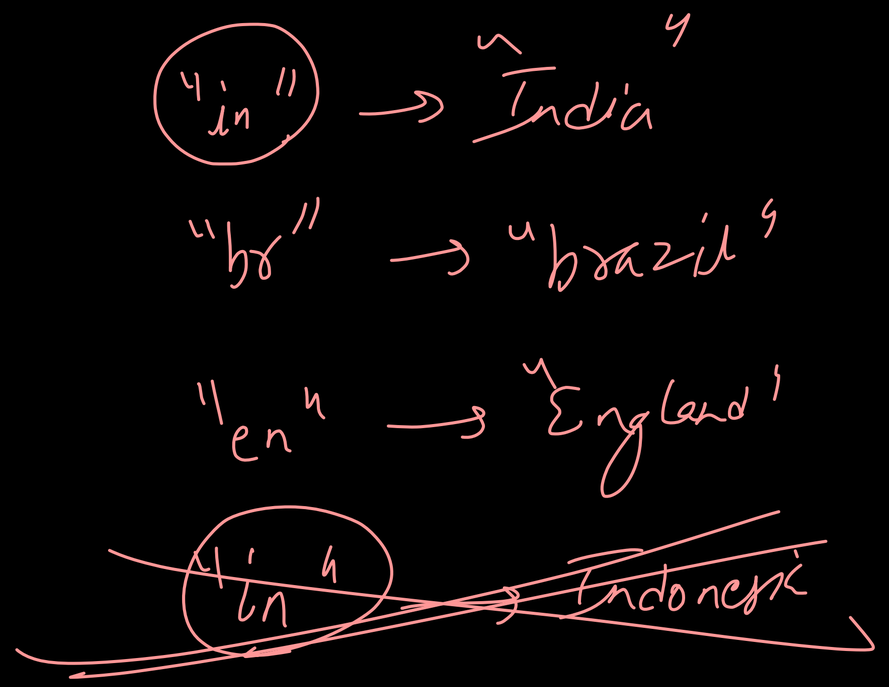
Member Functions:

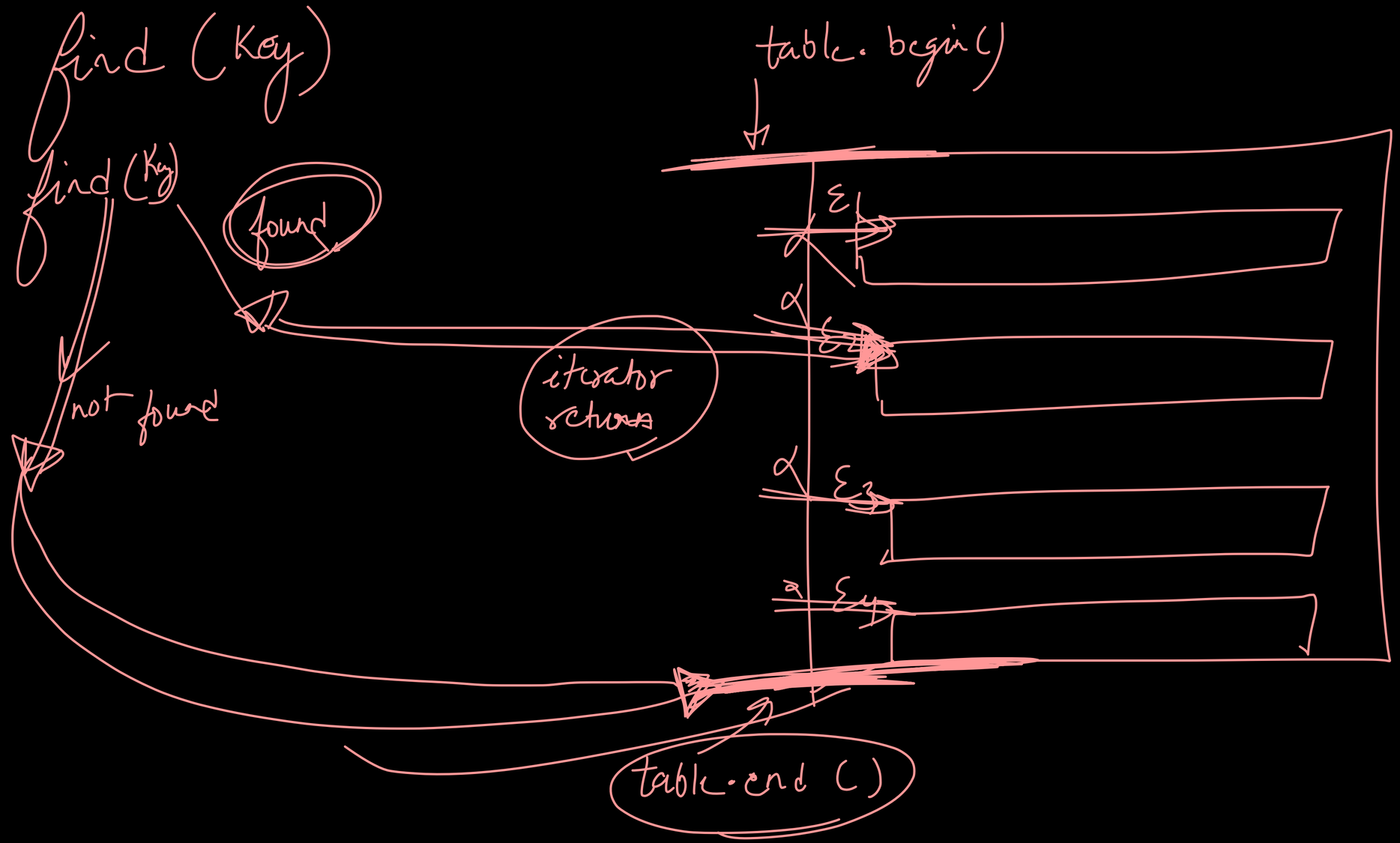
- empty(): Checks if the priority queue is empty (i.e., whether its size is 0).
- size(): Returns the number of elements currently in the priority queue.
- top(): Accesses the element at the top of the priority queue, which is the largest (or highest priority) element, depending on the comparator used.
- push(const T& value): Adds an element to the priority queue and reorders it to maintain the heap property.
- pop(): Removes the top element from the priority queue, which is the largest (or highest priority) element, and reorders the remaining elements to maintain the heap property.
- swap(priority_queue& x): Swaps the contents of the priority queue with those of another priority queue of the same type, including their underlying containers and comparators.

Map:

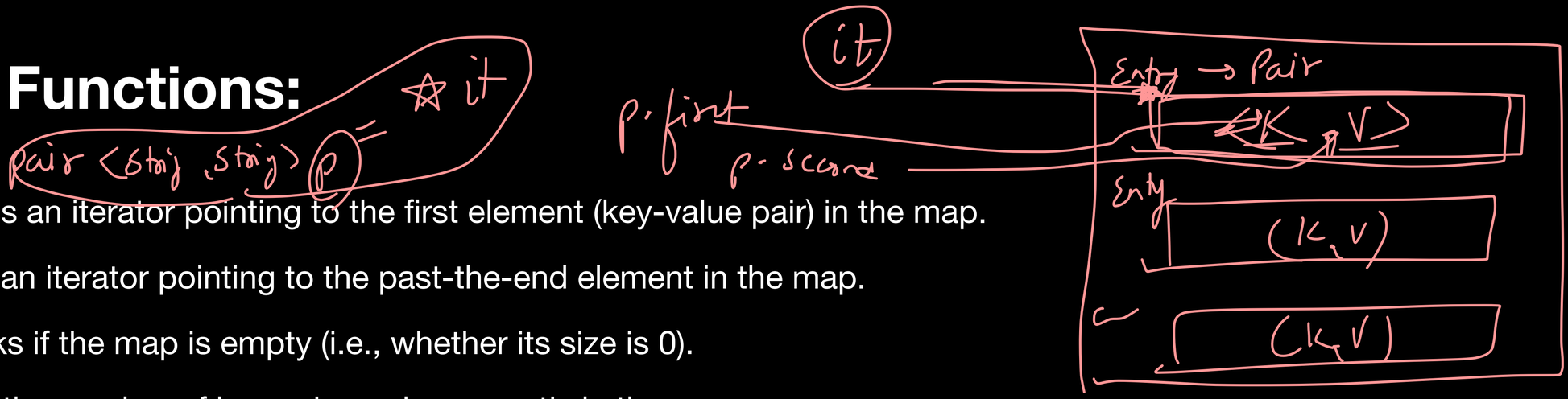
Table → Collection of Key-value Pairs

- Associative container that stores key-value pairs.
- Allows efficient retrieval and modification of values based on keys.
- Keys are unique within the map.

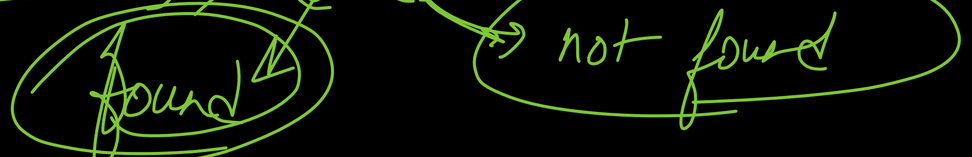



find (Key)
find (Key)
found
not found
iterator
table.begin()
table.end()

Member Functions:

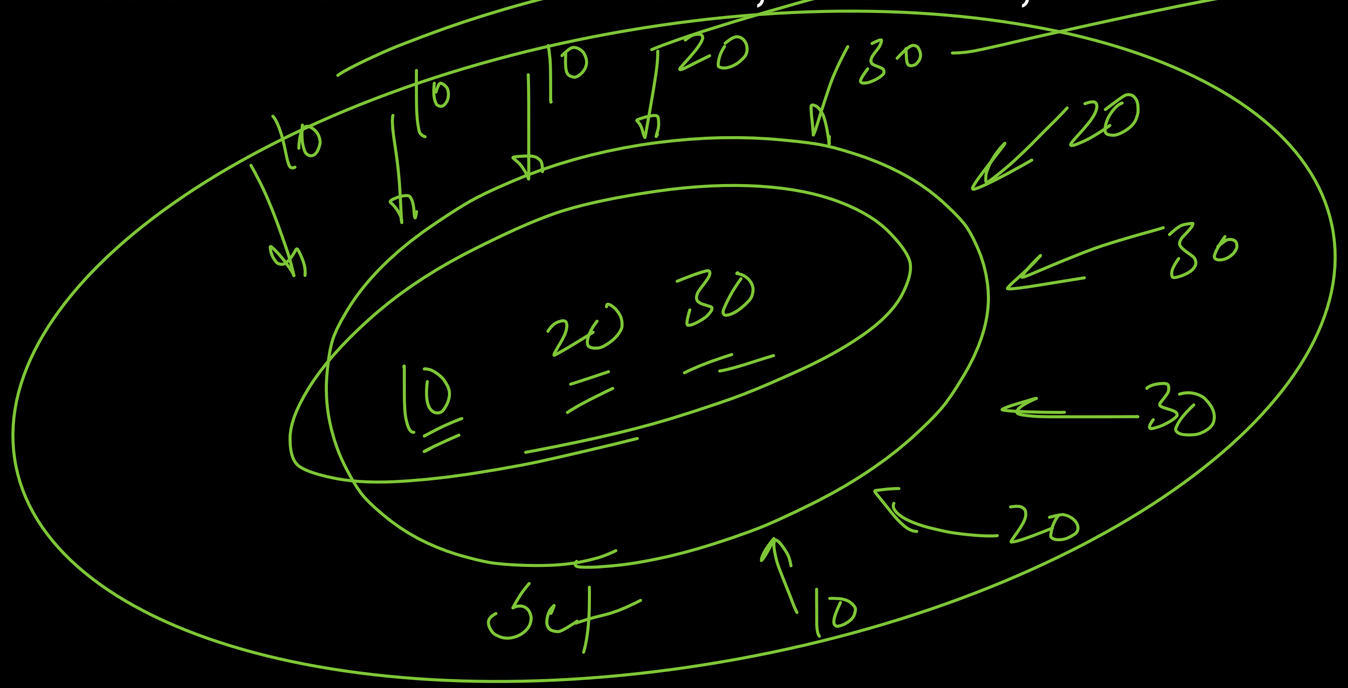


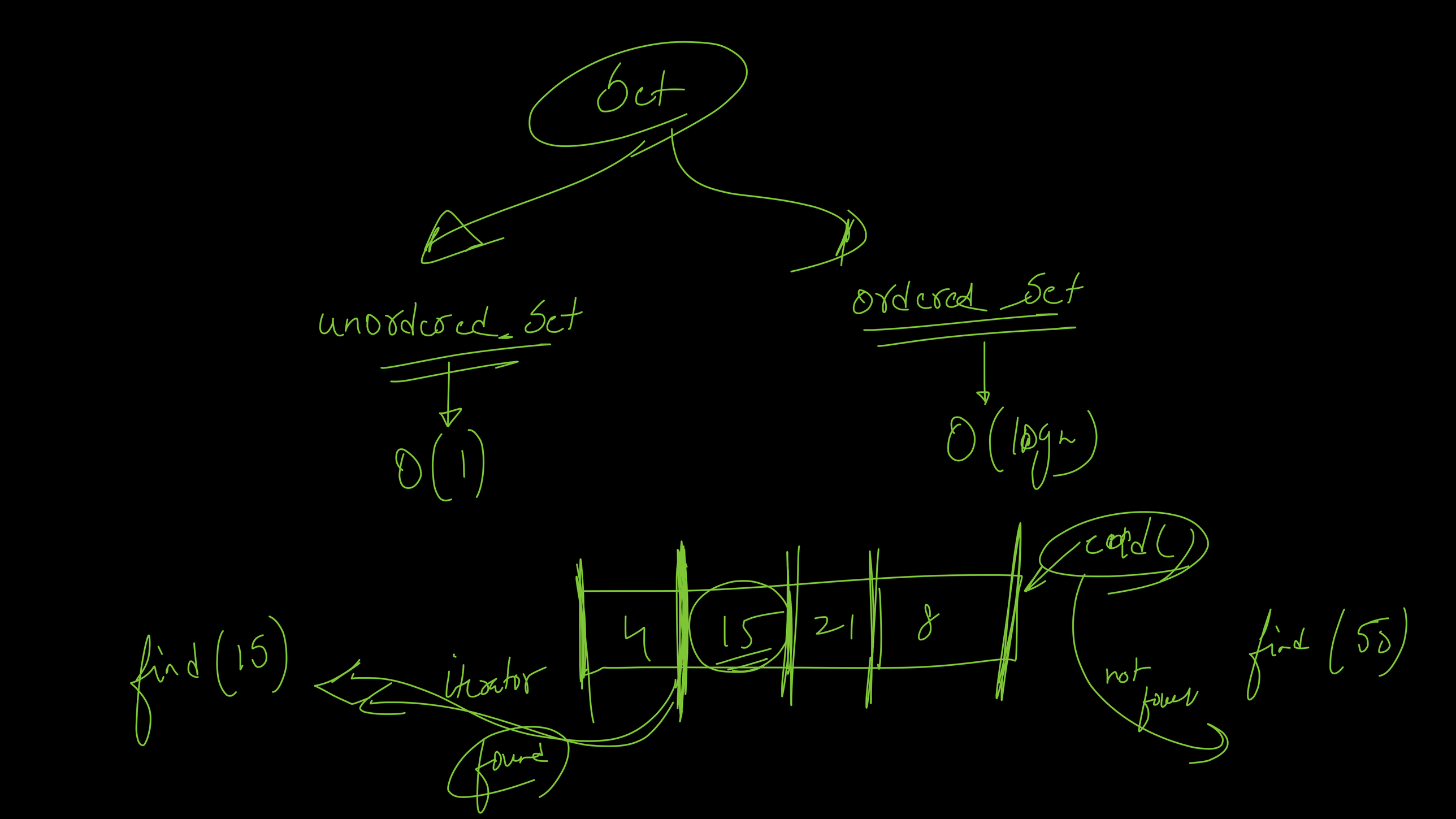
- begin(): Returns an iterator pointing to the first element (key-value pair) in the map.
- end(): Returns an iterator pointing to the past-the-end element in the map.
- empty(): Checks if the map is empty (i.e., whether its size is 0).
- size(): Returns the number of key-value pairs currently in the map.
- operator[](const Key& key): Accesses the value associated with the key, inserting a new element if the key does not exist.
- at(const Key& key): Accesses the value associated with the key, throwing an exception if the key is not found.
- insert(const std::pair<Key, Value>& value) or insert(iterator hint, const std::pair<Key, Value>& value): Inserts a new key-value pair into the map; with a hint, it can potentially improve insertion efficiency.
- erase(const Key& key) or erase(iterator position) or erase(iterator first, iterator last): Removes one or more elements from the map specified by key or position.
- clear(): Removes all key-value pairs from the map, which are destroyed, and leaves it with a size of 0.
- find(const Key& key): Returns an iterator to the element with the given key, or end() if the key is not found.
- count(const Key& key): Returns the number of elements with the specified key (1 or 0 since std::map does not allow duplicate keys).



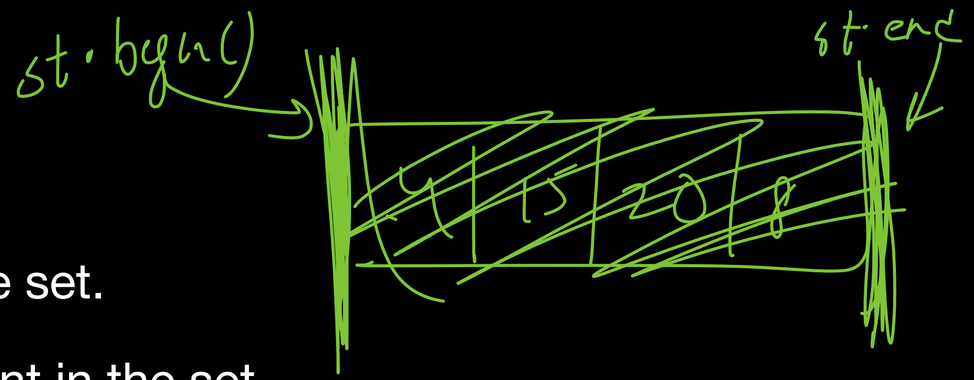
Set:

- Sorted collection of unique elements.
- Elements are stored in sorted order, and duplicates are automatically removed.
- Provides efficient insertion, deletion, and search operations.




Set
unordered_set
ordered_set
O(1)
O(log n)
find(15)
iterator
found
4
15
21
8
end()
not found
find(55)

Member Functions:



- begin(): Returns an iterator pointing to the first element in the set.
- end(): Returns an iterator pointing to the past-the-end element in the set.
- empty(): Checks if the set is empty (i.e., whether its size is 0).
- size(): Returns the number of elements currently in the set.
- insert(const T& value): Inserts a new element into the set, maintaining the order and uniqueness of elements.
- erase(const T& key) or erase(iterator position) or erase(iterator first, iterator last): Removes one or more elements from the set specified by key or position.
- clear(): Removes all elements from the set, which are destroyed, and leaves it with a size of 0.
- find(const T& key): Returns an iterator to the element with the given key, or end() if the key is not found.
- count(const T& key): Returns the number of elements with the specified key (1 or 0, since std::set does not allow duplicate keys).

erase(st.begin(), st.end())

found

Not found

Algorithms in STL:

The C++ Standard Template Library (STL) includes a wide range of algorithms that operate on various container types (such as vectors, lists, sets, and maps) and provide essential functionality for data manipulation. These algorithms are defined in the <algorithm> header and are part of the STL's core functionality.

Iterators and Iterating Algorithms:

- std::for_each: Applies a function to each element in a range. std::find: Searches for a specific element in a range.
- std::find_if: Searches for the first element that satisfies a given predicate.
- std::count: Counts the occurrences of a value in a range.
- std::count_if: Counts the elements that satisfy a given predicate.
- std::sort: Sorts the elements in a range in ascending order.
- std::reverse: Reverses the order of elements in a range.
- std::rotate: Rotates elements in a range.
- std::unique: Removes duplicate elements from a sorted range.
- std::partition: Divides elements in a range into two groups based on a predicate.

Numeric Algorithms:

- std::accumulate: Computes the sum of elements in a range.
- std::inner_product: Computes the inner product of two ranges.
- std::partial_sum: Computes the partial sums of a range.
- std::iota: Fills a range with incrementing values.

Searching and Finding Algorithm:

- std::binary_search: Checks if a value exists in a sorted range.
- std::lower_bound: Finds the first element greater or equal to a value in a sorted range.
- std::upper_bound: Finds the first element greater than a value in a sorted range.
- std::equal_range: Finds a range of elements equal to a value in a sorted range.

Min and Max Algorithm:

- std::min: Returns the smaller of two values.
- std::max: Returns the larger of two values.
- std::min_element: Finds the smallest element in a range.
- std::max_element: Finds the largest element in a range.

Heap Algorithm:

- std::make_heap: Converts a range into a max-heap.
- std::push_heap: Inserts an element into a max-heap.
- std::pop_heap: Removes the largest element from a max-heap.
- std::sort_heap: Sorts a range that represents a max-heap.

Set Algorithm:

- std::set_union: Computes the union of two sorted ranges.
- std::set_intersection: Computes the intersection of two sorted ranges.
- std::set_difference: Computes the difference between two sorted ranges.
- std::set_symmetric_difference: Computes the symmetric difference of two sorted ranges.

Functors:

- Functors, short for "function objects," are a fundamental concept in C++ and play a significant role in the C++ Standard Template Library (STL). Functors are objects that behave like functions and can be called with the same syntax as regular functions. They are implemented as classes or structs that overload the operator().

- In the context of the C++ STL, functors are often used as custom comparators or custom operations when working with various algorithms and containers

Usage in algorithms:

- Functors are commonly used as arguments to STL algorithms, providing custom behavior for sorting, searching, and other operations.
- For example, when sorting a container of custom objects, you can provide a functor that defines the sorting criteria based on specific object attributes.

Random Access Iterators:

```cpp
int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // Creating a random access iterator
    std::vector<int>::iterator it = numbers.begin();

    // Using the iterator to read and write values
    std::cout << "Forward traversal: ";
    for (; it != numbers.end(); ++it) {
        std::cout << *it << " ";
    }

    // Using the iterator to traverse backward
    std::cout << "\nBackward traversal: ";
    for (it = numbers.end() - 1; it >= numbers.begin(); --it) {
        std::cout << *it << " ";
    }

    // Using iterator arithmetic
    std::cout << "\nUsing iterator arithmetic: ";
    it = numbers.begin() + 2;  // Move to the third element (index 2)
    std::cout << *it << " ";

    return 0;
}
```

Function Call Operator operator():

- Functors define the behavior of their function call operator operator().
- When a functor object is called like a function, the operator() is executed, allowing you to encapsulate custom behavior within the functor.

Custom Comparators:

- Functors are frequently used to provide custom comparison logic when sorting or searching in STL containers.
- For example, when using the std::sort algorithm, you can pass a custom functor to specify the sorting order.

```cpp
#include <iostream>

// Functor for comparing integers in descending order
struct DescendingComparator {
    bool operator()(int a, int b) const {
        return a > b;
    }
};

int main() {
    DescendingComparator cmp;

    int x = 5, y = 3;

    if (cmp(x, y)) {
        std::cout << x << " is greater than " << y << std::endl;
    } else {
        std::cout << x << " is not greater than " << y << std::endl;
    }

    return 0;
}
```

Advantages of Functors:

- Functors provide a more flexible way to customize behavior compared to function pointers [illegible]
- They are type-safe, allowing the compiler to catch type-related errors at compile-time.
- Functors can carry additional data or state, making them versatile for various tasks.

Advantages of Functors:

- Functors provide a more flexible way to customize behavior compared to function pointers or lambda expressions.
- They are type-safe, allowing the compiler to catch type-related errors at compile-time.
- Functors can carry additional data or state, making them versatile for various tasks.