STL

CONTAINERS

CODERS SCHOOL

MATEUSZ ADAMSKI

ŁUKASZ ZIOBROŃ

AGENDA

1. Sequence containers
2. Associative containers
3. Adaptors
4. Other containers

REPO `coders-school/stl:cr`

PRESENTATION IN PDF

SOMETHING ABOUT YOU

- Which STL containers are you familiar with?
- Which of them is your favourite?

ŁUKASZ ZIOBROŃ

NOT ONLY A PROGRAMMING XP

- Frontend dev & DevOps @ Coders School
- C++ and Python developer @ Nokia & Credit Suisse
- Team leader & Trainer @ Nokia
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Web developer (HTML, PHP, CSS) @ StarCraft Area

EXPERIENCE AS A TRAINER

- C++ trainings @ Coders School
- Practical Aspects Of Software Engineering @ PWr & UWr
- Nokia Academy @ Nokia
- Internal corporate trainings

PUBLIC SPEAKING EXPERIENCE

- code::dive conference
- code::dive community
- Academic Championships in Team Programming

HOBBIES

- StarCraft Brood War & StarCraft II
- Motorcycles
- Photography
- Archery
- Andragogy

CONTRACT

- 🎰 Vegas rule
- 🗣️ Discussion, not a lecture
- ☕ Additional breaks on demand
- ⏱️ Be on time after breaks

PRE-TEST 1/3 🤯

MARK TRUE STATEMENTS ABOUT STL CONTAINERS

1. In `std::deque`, items can be added from both sides using `push_front()` and `push_back()`, respectively.
2. `std::set` supports random access to contained elements.
3. `std::vector` supports random access to contained elements.
4. `std::deque` supports random access to contained elements.
5. `std::multimap` allows multiple values to be stored for every key value.

PRE-TEST 2/3 🤯

WHICH OF THE FOLLOWING CONTAINERS ARE CONTIGUOUS?

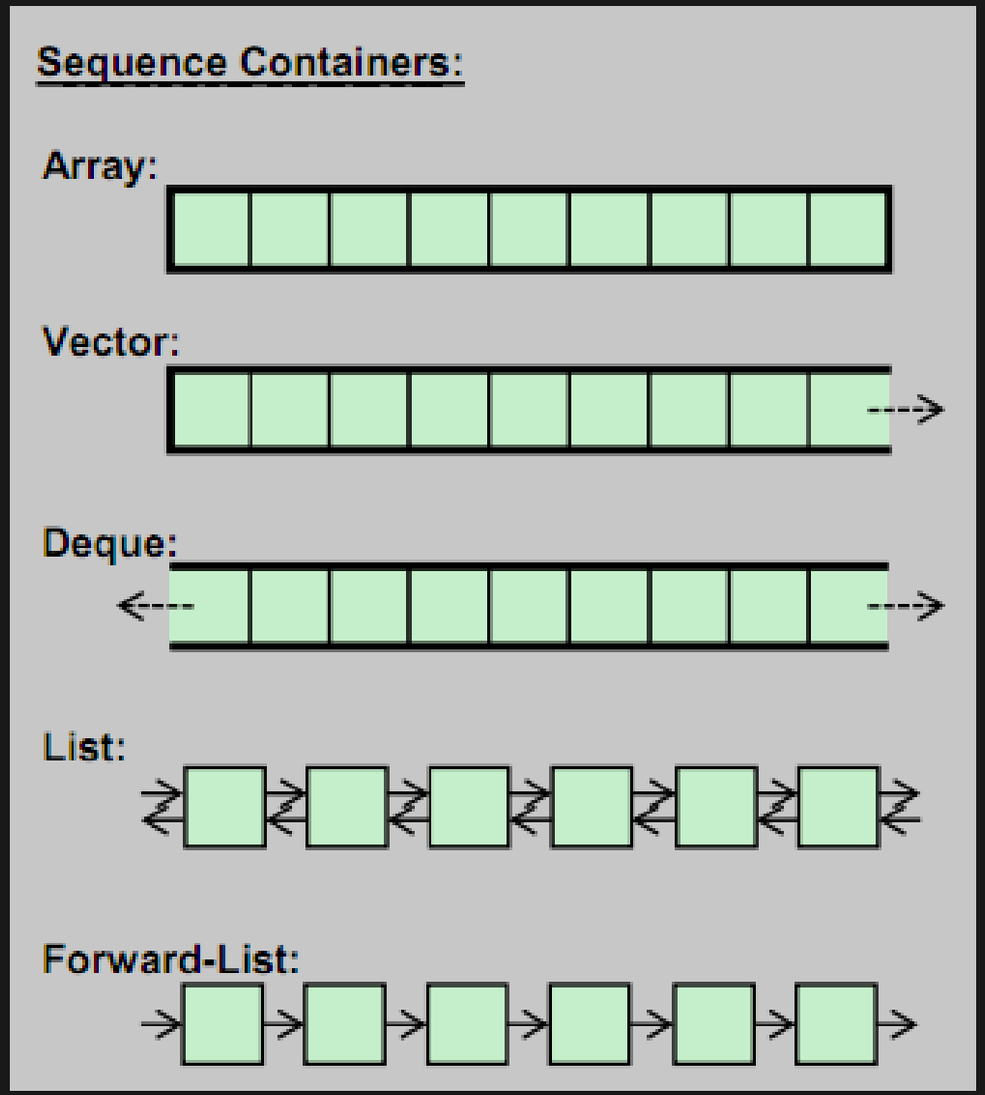
1. `deque`
2. `vector`
3. `unordered_set`
4. `array`
5. `forward_list`

PRE-TEST 3/3 🤯

WHAT IS THE COMPLEXITY OF OPERATIONS ON `std::unordered_map`?

1. Inserting/removing `O(log n)`, accessing `O(1)`, searching `O(log n)`
2. Inserting/removing `O(1)`, accessing `O(1)`, searching `O(1)`
3. Inserting/removing `O(n)`, accessing `O(n)`, searching `O(n)`
4. Inserting/removing `O(nlog n)`, accessing `O(log n)`, searching `O(n)`
5. Inserting/removing `O(nlog n)`, accessing `O(1)`, searching `O(log n)`

SEQUENCE CONTAINERS



WHAT IS A SEQUENCE CONTAINER?

It's a container, in which elements are stored exactly in the order we put them.

Sequence containers implement data structures that can be accessed sequentially.

WHAT ARE THE DIFFERENCES BETWEEN THEM?

LET'S HAVE A DISCUSSION!

Consider:

- Available methods
- Memory usage
- Time complexity of accessing the specific element
- Time complexity of searching for a matching item
- Time complexity of insertion and deletion

Container	Insertion	Access	Erase	Find	Persistent Iterators
vector / string	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)	Sorted: O(log n) Other: O(n)	Pointers only
list / forward_list	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	O(n)	Yes
set / map	O(log n)	-	O(log n)	O(log n)	Yes
unordered_set / unordered_map	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	Pointers only
priority_queue	O(log n)	O(1)	O(log n)	-	-

std::vector<T>

ARRAY WITH DYNAMIC SIZE

CODERS SCHOOL

SIZE VS CAPACITY

`std::vector<T>` TRAITS

- Cache-friendly
- Items are arranged contiguously in memory just like in a regular array
- The complexity of adding a new item:
  - Is constant when `size() < capacity()`
  - Is linear when `size() >= capacity()`
  - The vector allocates additional memory space
  - When there is no more space for additional data in the current area, the entire vector will be copied to another memory location (copy constructor is called for every element)
- Removing the last element from a vector is cheap, but removing the element from the middle or beginning is expensive
- We usually do not have proper intuition about how the cache memory can improve the performance
- Iterators are invalidated after each insertion or deletion
  - You can use `__gnu_debug::vector<T>` from `#include <debug/vector>` to have debug information about invalid iterators

`std::vector<T>` METHODS

`std::vector<T>` ON CPPREFERENCE.ORG

- adding an item: `push_back()`, `emplace_back()`, `insert()`
- modifying/accessing an item: `at()`, `operator[]`
- first/last item: `back()`, `front()`
- size/is the container empty: `size()`, `empty()`
- reserved memory space: `capacity()`
- reserve memory space: `reserve()`
- clearing unused memory: `shrink_to_fit()`
- start/end iterator: `begin()`, `end()`
- reverse iterator: `rbegin()`, `rend()`
- constant iterator: `cbegin()`, `cend()`, `crbegin()`, `crend()`
- clearing the container: `clear()`
- preparing an item for removal: `remove()` (not a method of `std::vector<T>`)
- erasing items from memory: `erase()`
- replacement of the entire container: `swap()`

OPERATIONS COMPLEXITY

- Insertion/deletion
  - `O(1)` - at the end, if `size() < capacity()`
  - `O(n)` - at the end, if `size() >= capacity()`
  - `O(n)` - other
- Access
  - `O(1)`
- Searching
  - `O(n)`

MEMORY USAGE

- Generally: `n * sizeof(T)`
- `O(n)`
- In fact: gradual allocation dependent on the capacity (implementation-defined)
  - You can use `reserve()` to allocate memory upfront
  - You can use `shrink_to_fit()` to release not used memory (shrinks the capacity to the size)
- Additional small constant memory for internal data is used (size, capacity, allocator)

ITERATOR INVALIDATION

- All read-only operations
  - Never
- Insertion
  - If the vector changed capacity, all iterators are invalidated
  - If not, only those at or after the insertion point (including `end()`).
- Deletion
  - Erased elements and all elements after them (including `end()`)

Refer to cppreference.org

(ALMOST) DELETING ITEMS 😉

`std::remove()` FROM `<algorithm>` HEADER

```
1 template< class ForwardIt, class T >
2 ForwardIt remove( ForwardIt first, ForwardIt last, const T& value );
```

Since removing items from the end of the vector is fast, the STL library allows us to prepare `std::vector<T>` to remove elements by moving the correct ones to the beginning of the container.

As a result, values to be removed are overwritten with values from the end of the vector that should not be removed.

Therefore, "garbage" remains at the end of the vector, which must be erased from memory.

```
1 std::vector<int> vec{1, 4, 2, 4, 3, 4, 5};
2 auto it = std::remove(vec.begin(), vec.end(), 4);
3 // for example: vec {1, 2, 3, 5, 3, 4, 5}
4                                   ^it
```

`std::remove()` returns an iterator that points to the beginning of the data to be erased.

ERASE

std::vector<T>::erase()

```
template< class T, class Alloc, class U >
constexpr typename std::vector<T,Alloc>::size_type
    erase(std::vector<T, Alloc>& c, const U& value);
```

Thanks to the `erase()` function, we can now remove unnecessary data from the container:

```
std::vector<int> vec{1, 4, 2, 4, 3, 4, 5};
auto it = std::remove(vec.begin(), vec.end(), 4);
vec.erase(it, vec.end());
// vec {1, 2, 3, 5}
```

We can also write it all on one line (Erase-Remove Idiom)

```
vec.erase(std::remove(vec.begin(), vec.end(), 4), vec.end());
```

TASK

During this session, we will write a "framework" for performance testing of STL containers. We will start with only basic functionality and later we will implement additional features. Tasks for now are:

1. Measure the time of accessing the middle element of the vectors of ints of size 100K and 1M. Use `std::chrono::high_resolution_clock()`. Print the measurement result and the description in one line.

2. We are going to measure accessing the middle element for other containers as well. Please refactor the code. Extract a `measure` function that takes 2 parameters:

   - a description of measurement as a string
   - a function without parameters (so far) - use a template parameter for the function

   The `measure` function should measure the execution time of the given function. The measurement result and the description should be printed in one line.

std::array<T, N>

FIXED SIZE ARRAY

CODERS SCHOOL

`std::array<T, N>` TRAITS

- Cache-friendly
- Type `<N>` denotes the size of the array, which must be known at compile time
- Items are arranged contiguously in memory just like in a regular array
- `std::array` is the most primitive wrapper on regular arrays, the efficiency is the same
- We cannot add a new item, we can only modify the existing items (size does not change)
- We cannot remove items (size does not change)
- Reading and modifying elements are very fast

`std::array<T, N>` METHODS

`std::array<T, N>` ON CPPREFERENCE.ORG

- adding an item: impossible
- modifying/accessing an item: `at()`, `operator[]`
- first/last item: `back()`, `front()`
- size/is container empty: `size()`, `empty()`
- start/end iterator: `begin()`, `end()`
- reverse iterator: `rbegin()`, `rend()`
- constant iterator: `cbegin()`, `cend()`, `crbegin()`, `crend()`
- clearing the container: you can't, but there is a method `fill` which can reset all elements
- preparing item for removal: impossible
- erasing items from memory: impossible
- replacement of the entire container: `swap()`

OPERATIONS COMPLEXITY

- Insertion/deletion
  - Not available
- Access
  - `O(1)`
- Searching
  - `O(n)`

MEMORY USAGE

- `n * sizeof(T)`
- `O(n)`
- No additional memory overhead
- Size is kept as a template parameter, not a member

ITERATOR INVALIDATION

- Never
- During `swap`, the iterator will continue to point to the same array element, and will thus change its value

PASSING `std::array<T, N>` TO FUNCTION

Because `std::array<T, N>` has 2 template parameters, some may have a problem passing it to a function

```
1  template <size_t SIZE>
2  void print(const std::array<int, SIZE>& arr) {
3      for (const auto& el : arr) {
4          std::cout << el << ' ';
5      }
6      std::cout << '\n';
7  }
8
9  int main() {
10     std::array<int, 10> arr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11     print(arr);
12
13     return 0;
14 }
```

TASK

Develop our performance testing framework.

3. Measure the time of accessing the middle element of the `std::array` of ints of size 100K and 1M.

std::list<T>

DOUBLY-LINKED LIST

CODERS SCHOOL

`std::list<T>` TRAITS

- Elements are scattered around memory (cache-unfriendly)
- Each element (node) has a pointer to the previous and the next element (mind the memory consumption)
- Adding a new element is cheap, if you already have the iterator. The necessary memory for the node is allocated and proper pointers (next/prev) are set.
- Deleting an element is fast, the allocated memory is freed and neighbouring nodes have their pointers changed
- Searching for a node (e.g. to remove or insert a new element after it) is expensive. We have to iterate through all the nodes one by one until we find the wanted one (even if we know exactly that it is e.g. the 40th element in the list)
- Iterators are not invalidated after each insertion or deletion
- Only the iterator to the deleted node is invalidated

`std::list<T>` METHODS

`std::list<T>` ON CPPREFERENCE.ORG

- adding an item: `push_back()`, `emplace_back()`, `push_front()`, `emplace_front()`, `insert()`
- modify/access an item: via iterators
- first/last item: `back()`, `front()`
- size/is container empty: `size()`, `empty()`
- start/end iterator: `begin()`, `end()`
- reverse iterator: `rbegin()`, `rend()`
- constant iterator: `cbegin()`, `cend()`, `crbegin()`, `crend()`
- clearing the container: `clear()`
- sorting: `sort()`
- reversing: `reverse()`
- removing duplicates: `unique()`
- removing items: `remove()`
- erasing items: `erase()`
- replacement of the entire container: `swap()`

OPERATIONS COMPLEXITY

- Insertion/deletion
  - `O(1)` - at the end or the beginning
  - `O(1)` - with an iterator, anywhere
  - `O(n)` - other
- Access
  - `O(1)` - at the end or at the beginning
  - `O(1)` - with an iterator, anywhere
  - `O(n)` - other
- Searching
  - `O(n)`

MEMORY USAGE

- `n * (sizeof(T) + 2 * sizeof(X*))`
- `O(n)`
- Additional small constant memory for internal data is used (size, head, tail, allocator)

ITERATOR INVALIDATION

- Adding, removing and moving the elements within the list or across several lists does not invalidate the iterators or references
- An iterator is invalidated only when the corresponding element is deleted

std::list<T>::remove() && std::list<T>::erase()

`std::list` has its own `remove()` method. We don't need to use `erase()`.

```
1 std::list<int> list{1, 4, 2, 4, 3, 4, 5};
2 list.remove(4);
3 // list {1, 2, 3, 5}
```

`erase()` is used in the same way as for `std::vector<T>`

```
1 std::list<int> list{1, 2, 3, 4, 5, 6, 7, 8};
2 auto it = list.begin();
3 std::advance(it, 3); // like on pointer ptr += 3
4 list.erase(list.begin(), it);
5 // list {4, 5, 6, 7, 8}
```

`std::advance()` is used to increment iterators. In our case, we are moving 3 elements forward.

TASK

4. Measure the time of accessing the middle element of the lists of size 100K and 1M.
5. Measure if accessing a vector or an array via an index is faster than accessing it via advancing the begin iterator.
6. Refactor the code if necessary.

std::forward_list<T>

SINGLY-LINKED LIST

CODERS SCHOOL

`std::forward_list<T>` TRAITS

- The same properties as `std::list<T>`
- Less memory consumption - only "next" pointer, no "prev" pointers
- Due to different memory layout it is considered as faster than `std::list<T>`
- Backward iteration is not possible
- There is no `size()` method

`std::forward_list<T>` METHODS

`std::forward_list<T>` ON CPPREFERENCE.ORG

- adding an item: `push_front()`, `emplace_front()`, `insert_after()`, `emplace_after()`, NO `push_back()`, NO `emplace_back()`
- modify/access an item: via iterators
- first/last item: `front()`, NO `back()`
- size/is the container empty: NO `size()`, `empty()` is available
- start/end iterator: `begin()`, `end()`
- an iterator that points to the element before `begin()`: `before_begin()`
- constant iterator: `cbegin()`, `cend()`, `cbefore_begin()`
- clearing the container: `clear()`
- sorting: `sort()`
- reversing: `reverse()`
- removing duplicates: `unique()`
- removing items: `remove()`
- erasing items: `erase_after()`
- erase items using `<algorithm>`: `erase()`
- replacement of the entire container: `swap()`

OPERATIONS COMPLEXITY

- The same as for `std::list<T>`
- Insertion/deletion
  - `O(1)` - at the end or the beginning
  - `O(1)` - with an iterator, anywhere
  - `O(n)` - other
- Access
  - `O(1)` - at the end or at the beginning
  - `O(1)` - with an iterator, anywhere
  - `O(n)` - other
- Searching
  - `O(n)`

MEMORY USAGE

- `n * (sizeof(T) + sizeof(X*))`
- `O(n)`
- Additional small constant memory for internal data is used (head, allocator)

ITERATOR INVALIDATION

- Adding, removing and moving the elements within the list or across several lists does not invalidate the iterators or references.
- An iterator is invalidated only when the corresponding element is deleted.

std::forward_list<T>::insert_after() && std::forward_list<T>::before_begin()

A singly-linked list allows us to insert an element after a specific node. If we want to put an item at the beginning of the list using the `insert_after`, we need to give it a special iterator `before_begin` that points to the element before the first node of the list. Thus, the method `insert_after()` will insert the desired value exactly as the first item in the list.

```
std::forward_list<int> list {1, 2, 3, 4, 5, 6};
list.insert_after(list.begin(), 10);
print(list);
list.insert_after(list.before_begin(), 0);
print(list);
```

Output:

```
1 10 2 3 4 5 6
0 1 10 2 3 4 5 6
```

std::forward_list<T>::remove()

`std::forward_list` has its own `remove()` method. We don't need to use `erase()`.

```
1 std::forward_list<int> list {1, 4, 2, 4, 3, 4, 5};
2 list.remove(4);
3 // {1, 2, 3, 5}
```

std::forward_list<T>::erase_after()

`erase_after()` is used to remove nodes from the position after the element indicated by the first iterator to the point indicated by the second iterator (without the element pointed to by it).

```
1 std::forward_list<int> list {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2 auto it = list.begin();
3 std::advance(it, 4);
4 std::cout << "it: " << *it << std::endl;
5 auto it2 = list.begin();
6 std::advance(it2, 7);
7 std::cout << "it2: " << *it2 << std::endl;
8 list.erase_after(it, it2);
9 print(list);
```

Output:

```
it: 5
it2: 8
1 2 3 4 5 8 9 10
```

TASK

7. Measure the time of accessing the middle element of the forward lists of size 100K and 1M.
8. Print the `sizeof()` of each container.

std::deque<T>

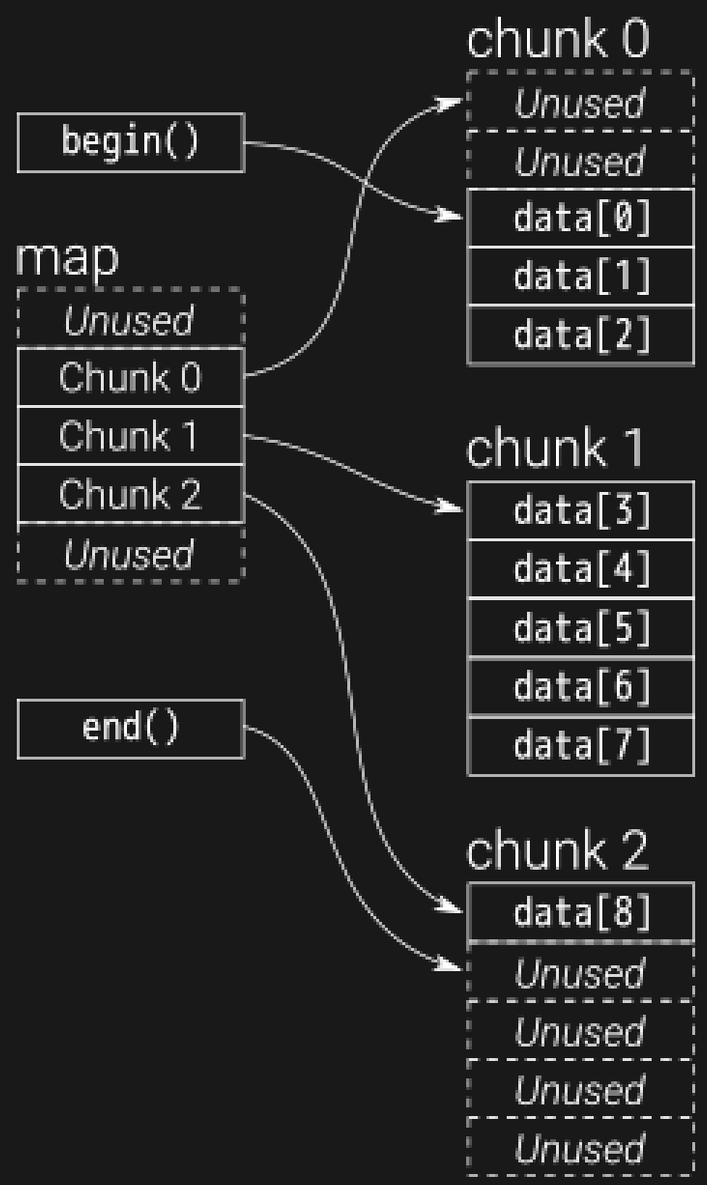
DOUBLE-ENDED QUEUE

CODERS SCHOOL

`std::deque<T>` TRAITS

- The hybrid of a list and a vector
- `deque` is divided into chunks of arrays, which are scattered around the memory
- The size of chunks in up to the implementation
- Additionally, deque comes with one more vector of pointers that indicate the start of each chunk in memory.
- This way we get 2 things:
  - Adding a new item is faster because we always allocate memory for the whole chunk and we will not move items like in `std::vector<T>` when we run out of space to allocate additional memory
  - The data from one chunk is cache-friendly, but the whole `std::deque` is not

std::deque<T> STRUCTURE



`std::deque<T>` TRAITS CONT.

- Each chunk is represented in memory like an array, while chunks themselves are not adjacent to each other and are spread out like list nodes
- Adding a new item is cheap
  - If a given chunk still has free space, we save it at the end/beginning
  - If not, a new chunk is allocated and the new element is put there
- Removal from the beginning and end is cheap - it only shifts the iterators `begin()` or `end()`
- Removing elements from the inside is expensive
- Reading and modification is cheap
  - We know the chunk size, so we know exactly from which element in our mapping vector we should read the chunk address
  - We also know which element to read the data from, because the chunk is arranged like an array.

ACCESSING THE ITEM

Mathematically speaking: if chunk has 16 elements and we want to get to the 100th element then:

- chunk no: `x = 100 / 16 -> x = 6`
- position: `y = 100 % 16 -> y = 4`

So we know that this is the 4th element in the 6th chunk.

This knowledge is completely unnecessary when using `std::deque`. The container takes care of this automatically.

`std::deque<T>` METHODS

`std::deque<T>` ON CPPREFERENCE.ORG

- adding an item: `push_back()`, `emplace_back()`, `push_front()`, `emplace_front()`, `insert()`
- modifying/accessing an item: `at()`, `operator[]`
- first/last item: `back()`, `front()`
- size/is the container empty: `size()`, `empty()`
- clearing unused memory: `shrink_to_fit()`
- start/end iterator: `begin()`, `end()`
- reverse iterator: `rbegin()`, `rend()`
- constant iterator: `cbegin()`, `cend()`, `crbegin()`, `crend()`
- clearing the container: `clear()`
- preparing an item for removal: `remove()` (not `std::deque` method),
- erasing items from memory: `erase()`
- replacement of the entire container: `swap()`

OPERATIONS COMPLEXITY

- Insertion/deletion
  - `O(1)` - at the end or at the beginning
  - `O(n)` - other
- Access
  - `O(1)`
- Searching
  - `O(n)`

MEMORY USAGE

- `n * sizeof(T) + n / chunk_size`
- `O(n)`
- Additional memory for vector of pointers is used
- Additional internal data is used (size, begin, end, allocator)

ITERATOR INVALIDATION

- Insertion and deletion at either end of a deque never invalidates pointers or references to the rest of the elements.
- Otherwise - all iterators are invalidated (including the past-the-end iterator).

TASK

9. Add measurements for accessing the middle elements of std::deque of size 100K and 1M.
10. Refactor the code, eg. create `measureBoth` function that takes 2 containers of the same type to simplify the code.

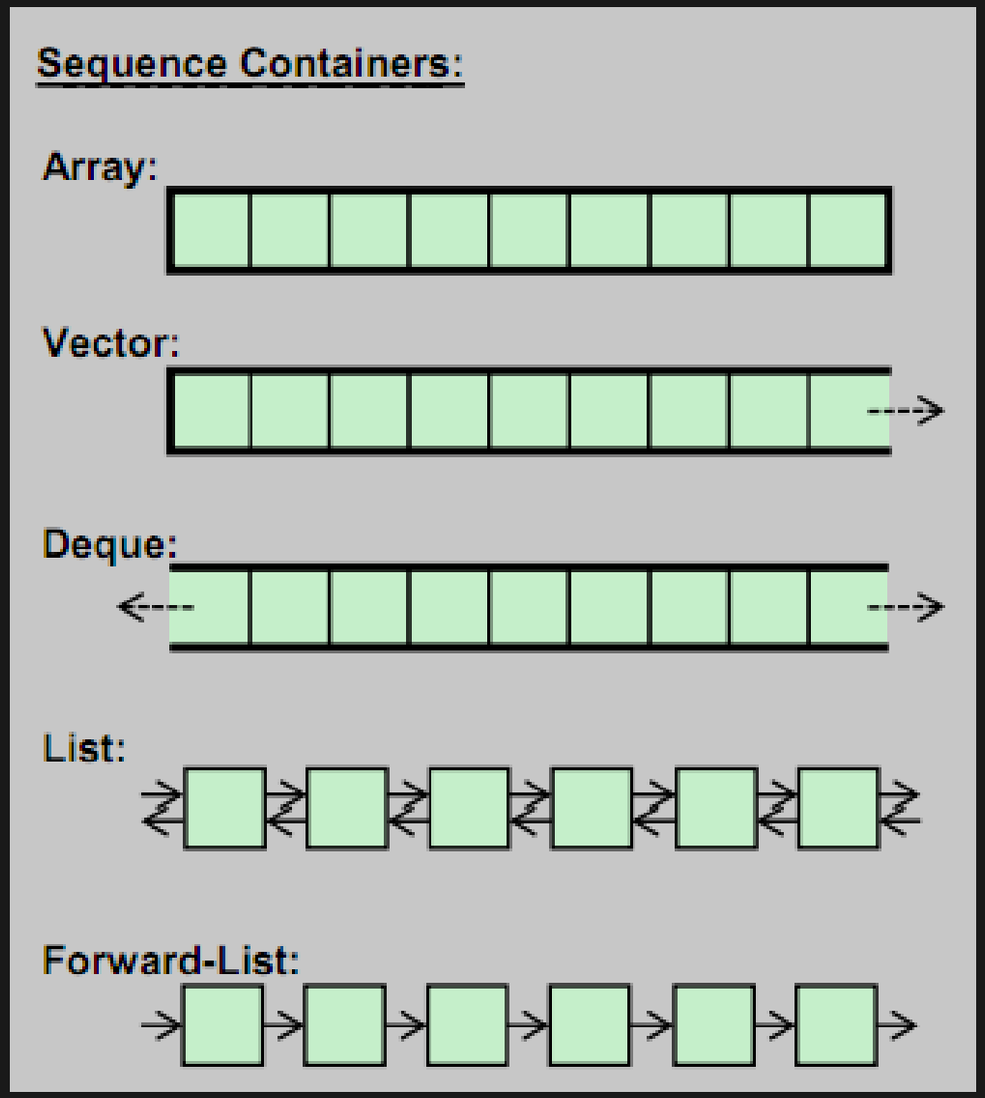
SEQUENCE CONTAINERS

SUMMARY

CODERS SCHOOL

WHAT DO YOU REMEMBER ABOUT SEQUENCE CONTAINERS?

1. 5 containers - mention them
2. Which of them are cache-friendly?
3. Operations complexity
4. Memory consumption
5. Iterators invalidation


Sequence Containers:
Array:
Vector:
Deque:
List:
Forward-List:

Container	Insertion	Access	Erase	Find	Persistent Iterators
vector / string	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)	Sorted: O(log n) Other: O(n)	Pointers only
list / forward_list	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	O(n)	Yes
set / map	O(log n)	-	O(log n)	O(log n)	Yes
unordered_set / unordered_map	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	Pointers only
priority_queue	O(log n)	O(1)	O(log n)	-	-

TASK

11. Fill all containers with unique values - eg. use `std::iota()`
12. Add measurements of inserting the data in the middle of all containers.

ASSOCIATIVE CONTAINERS

CODERS SCHOOL

WHAT IS AN ASSOCIATIVE CONTAINER?

It's a container, in which elements are stored in a predefined order.

ORDERED

- `set`
- `multiset`
- `map`
- `multimap`

Associative containers implement sorted data structures that can be quickly searched - `O(log n)` complexity.

UNORDERED (HASHED)

- `unordered_set`
- `unordered_multiset`
- `unordered_map`
- `unordered_multimap`

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched - `O(1)` average-case complexity, `O(n)` worst-case complexity.

C++ Standard Library Associative Containers

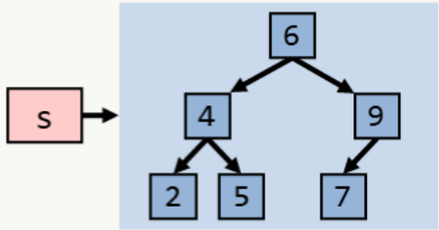
set<*Key*>

unique, ordered keys

multiset<*K*>

(non-unique) ordered keys

```
std::set<int> s;
s.insert(7); …
s.insert(5);
auto i = s.find(7);  // → iterator
if(i != s.end())     // found?
  cout << *i;        // 7
if(s.contains(7)) {…}   C++20
```



usually implemented as balanced binary tree (red-black tree)

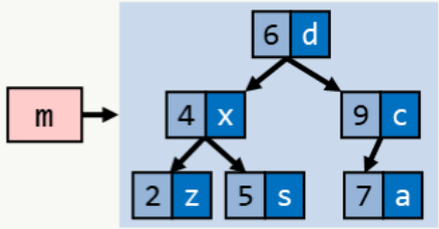
map<*Key*,*Value*>

unique key → value-pairs; ordered by keys

multimap<*K*,*V*>

(non-unique) key → value-pairs, ordered by keys

```
std::map<int,char> m;
m.insert({7,'a'});  …
m[4] = 'x';          // insert 4→x
auto i = s.find(7);  // → iterator
if(i != s.end())     // found?
  cout << i->first   // 7
       << i->second; // a
if(s.contains(7)) {…}   C++20
```



usually implemented as balanced binary tree (red-black tree)

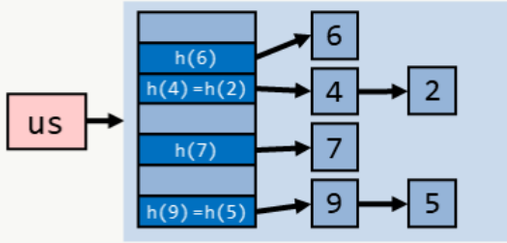
unordered_set<*Key*>

unique, hashable keys

unordered_multiset<*Key*>

(non-unique) hashable keys

```
std::unordered_set<int> us;
us.insert(7); …
us.insert(5);
auto i = us.find(7); // → iterator
if(i != us.end())    // found?
  cout << *i;        // 7
if(s.contains(7)) {…}   C++20
```



hash table for key lookup, linked nodes for key storage

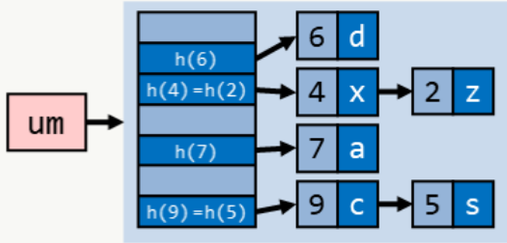
unordered_map<*Key*,*Value*>

unique key → value-pairs; hashed by keys

unordered_multimap<*Key*,*Value*>

(non-unique) key → value-pairs; hashed by keys

```
unordered_map<int,char> um;
um.insert({7,'a'});  …
um[4] = 'x';         // insert 4→x
auto i = um.find(7); // → iterator
if(i != um.end())    // found?
  cout << i->first   // 7
       << i->second; // a
if(s.contains(7)) {…}   C++20
```



hash table for key lookup, linked nodes for (key, value) pair storage

WHAT ARE THE DIFFERENCES BETWEEN THEM?

LET'S HAVE A DISCUSSION!

Consider:

- Ordering
- Key Type
- Uniqueness
- Memory usage
- Time complexity of accessing the specific element
- Time complexity of searching for a matching item
- Time complexity of insertion and deletion

OPERATIONS COMPLEXITY

Container	Insertion	Access	Erase	Find	Persistent Iterators
vector / string	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)	Sorted: O(log n) Other: O(n)	Pointers only
list / forward_list	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	O(n)	Yes
set / map	O(log n)	-	O(log n)	O(log n)	Yes
unordered_set / unordered_map	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	Pointers only
priority_queue	O(log n)	O(1)	O(log n)	-	-

MAP AND MULTIMAP

CODERS SCHOOL

`std::map<K, T>` AND `std::multimap<K, T>` TRAITS #1

- Usually implemented as a binary tree (red-black tree)
- Cache-unfriendly
- Hold key-value pairs ordered by key
- Unique keys (`map`) or repeating keys are allowed (`multimap`)
- An alternative to multimap is `std::map<key, std::vector<value>>`

std::map<K, T> AND std::multimap<K, T> TRAITS #2

In red-black trees, the time to insert, delete or add an element is `O(log n)`.

Advantages:

- A quick search for items - sequence containers have `O(n)` complexity,
- Adding and removing elements is relatively cheap
  - `std::vector<T>` adds items in the middle in time `O(n)`
  - `std::list<T>` (if an iterator is given) adds items in `O(1)`
  - Hence the map is relatively fast in both adding and removing
- Perfect when we frequently search for data and less frequently add or delete it.

If we use it as a regular container, we will lose efficiency. The map should be used when you want to have key-value pairs and search for them often. Otherwise, we can use `std::vector<pair<K, V>>`.

`std::map<K, T>` AND `std::multimap<K, T>` METHODS

`std::map<K, T>` on cppreference.org

`std::multimap<K, T>` on cppreference.org

- adding an item: `insert()`, `emplace()`, `emplace_hint()`. Additionally, the map has: `insert_or_assign()`, `try_emplace()` and `operator[]` (for inserting)
- modifying/accessing an item: `at()`, `operator[]` (`multimap` does not have such methods)
- size/is the container empty: `size()`, `empty()`
- clear unused memory: none
- first/last item: none
- start/end iterator: `begin()`, `end()`

`std::map<K, T>` AND `std::multimap<K, T>` METHODS #2

- reverse iterator: `rbegin()`, `rend()`
- constant iterator: `cbegin()`, `cend()`, `crbegin()`, `crend()`
- cleaning the container: `clear()`
- preparing an item for removal: none
- erasing items from memory: `erase()`
- replacement of the entire container: `swap()`
- counting elements matching a given key: `count()` (returns only 0 or 1 for `map` or any positive number from 0 to n for `multimap`)
- finding an element with a given key: `find()`
- contains a given key (C++20): `contains()`

OPERATIONS COMPLEXITY

- Insertion/deletion
  - `O(log n)`
- Access
  - `O(log n)`
- Searching
  - `O(log n)`

MEMORY USAGE

- `n * (sizeof(std::pair) + 2 * sizeof(X*))`
- `O(n)`
- Additional small constant memory for internal data is used (root, compare, allocator)

ITERATOR INVALIDATION

- Adding, removing and moving the elements within the map or across several maps does not invalidate the iterators or references.
- An iterator is invalidated only when the corresponding element is deleted.

EXAMPLE OF `emplace_hint` USAGE

```
1 int main() {
2     std::map<int, std::string> map;
3
4     auto it = map.begin();
5     map.emplace_hint(it, 10, "Ten");
6
7     std::cout << map[10] << '\n';
8 }
```

Output: `Ten`

We suggest the place where the element should be inserted in the map. Thanks to that, such an operation can have `O(1)` complexity.

However, if we give an incorrect hint, the insertion time is `O(log n)`. Rather rarely used 🙂

EXAMPLE OF `count` USAGE

```
int main() {
    std::multimap<int, std::string> map;

    map.insert({5, "Five"});
    map.insert({5, "Funf"});
    map.insert({5, "Piec"});
    map.insert({5, "Cinq"});
    std::cout << map.count(5) << '\n';
}
```

Output: 4

TASK

13. Measure the time of accessing/inserting the middle element of the `std::map` and `std::multimap` of size 100K and 1M.
14. Measure the time of accessing/inserting data with `operator[]` into the `std::map` when the key exists and when it does not exist.

SET AND MULTISET

`std::set<K>` AND `std::multiset<K>` TRAITS

- The same properties as `std::map<K, V>` and `std::multimap<K, V>`
- The only difference is, that `set` and `multiset` hold only ordered keys (not key-value pairs)
- Perfect when you want to always have sorted values

If we do not want the container to be always sorted, it might be better to use `std::vector<T>` and sort it when necessary.

If we also care about unique values, we can use `std::unique()`

`std::set<K>` AND `std::multiset<K>` METHODS #1

`std::set<K, T>` on cppreference.org

`std::multiset<K, T>` on cppreference.org

- adding an item: `insert()`, `emplace()`, `emplace_hint()`
- modifying / accessing an item: via iterators
- size / is the container empty: `size()`, `empty()`
- clear unused memory: none
- first / last item: none
- start / end iterator: `begin()`, `end()`
- reverse iterator: `rbegin()`, `rend()`

`std::set<K>` AND `std::multiset<K>` METHODS #2

- constant iterator: `cbegin()`, `cend()`, `crbegin()`, `crend()`
- clearing the container: `clear()`
- preparing item for removal: none
- erasing items from memory: `erase()`
- replacement of the entire container: `swap()`
- counting elements matching a given key: `count()` (returns only 0 or 1 for `map` or any positive number from 0 to n for `multimap`)
- finding an element with a given key: `find()`
- contains a given key (C++20): `contains()`

OPERATIONS COMPLEXITY

- Insertion/deletion
  - `O(log n)`
- Access
  - `O(log n)`
- Searching
  - `O(log n)`

MEMORY USAGE

- `n * (sizeof(K) + 2 * sizeof(X*))`
- `O(n)`
- Additional small constant memory for internal data is used (root, compare, allocator)

ITERATOR INVALIDATION

- Adding, removing and moving the elements within the map or across several maps does not invalidate the iterators or references.
- An iterator is invalidated only when the corresponding element is deleted.

EXAMPLE OF `std::set<K>` USAGE

```
std::vector<int> v = {5, 4, 3, 2, 1, 0, 6, 8, 7};
std::set<int> set {v.begin(), v.end()};
for (const auto el : set) {
    std::cout << el << ' ';
}
std::cout << '\n';

std::set<int, std::greater<int>> set2 {v.begin(), v.end()};
for (const auto el : set2) {
    std::cout << el << ' ';
}
std::cout << '\n';
```

Output:

```
0 1 2 3 4 5 6 7 8
8 7 6 5 4 3 2 1 0
```

EXAMPLE OF `std::multiset<K>` USAGE

```
std::vector<int> v = {5, 4, 3, 2, 1, 0, 6, 8, 7, 1, 2, 3, 4, 5, 6};
std::multiset<int> set {v.begin(), v.end()};
for (const auto el : set) {
    std::cout << el << ' ';
}
std::cout << '\n';

std::multiset<int, std::greater<int>> set2 {v.begin(), v.end()};
for (const auto el : set2) {
    std::cout << el << ' ';
}
std::cout << '\n';
```

Output:

```
0 1 1 2 2 3 3 4 4 5 5 6 6 7 8
8 7 6 6 5 5 4 4 3 3 2 2 1 1 0
```

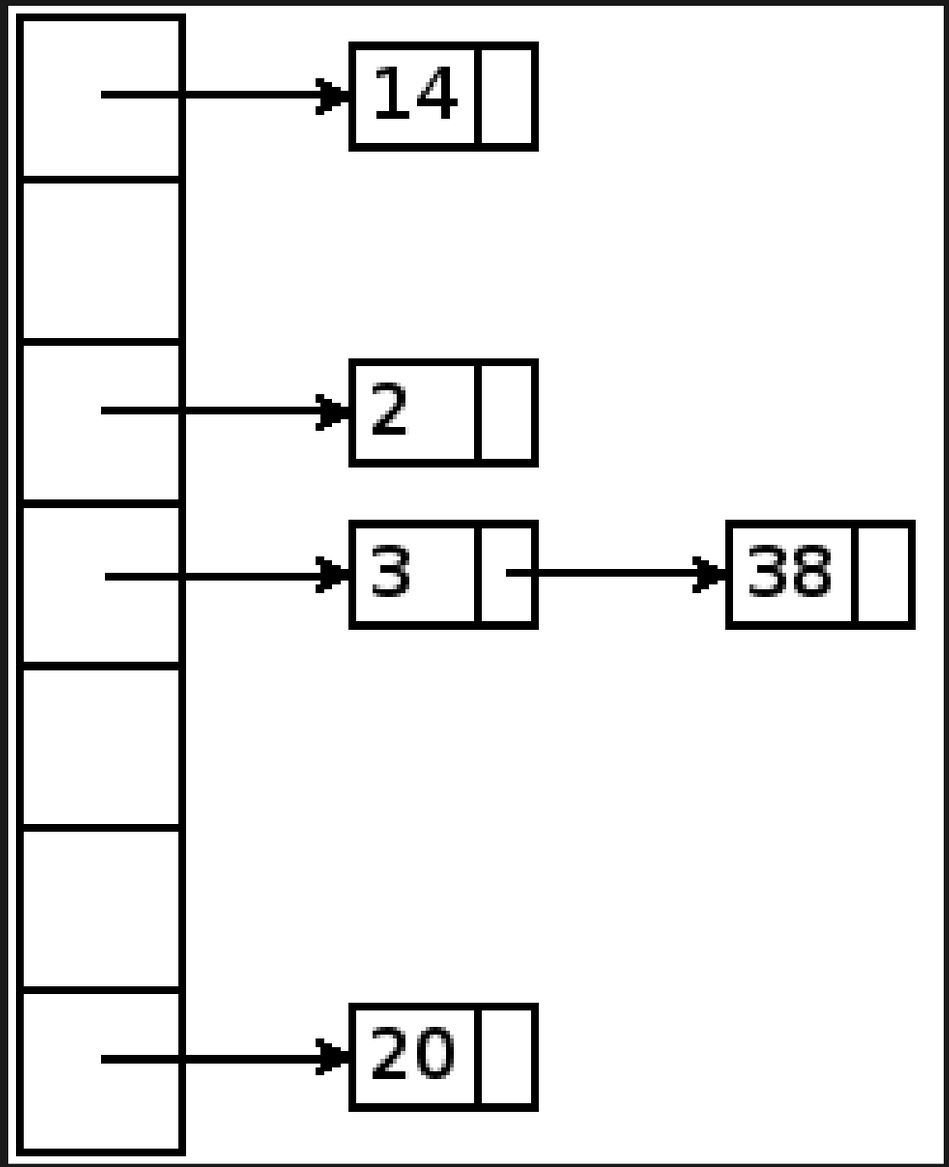
TASK

15. Measure the time of accessing the middle element of the `std::set` and `std::multiset` of size 100K and 1M.
16. Add measurements of inserting the data in the middle of these containers.

HASHING CONTAINERS

CODERS SCHOOL

HASH TABLE



HASH FUNCTION

It is any function that can be used to map data of arbitrary size to fixed-size values, eg. an index of an array.

RATE BELOW HASH FUNCTIONS

```
size_t hash(const std::string& str) { return str.size(); }
```

```
size_t hash(const std::string& str) {
    size_t index = 0;
    for (size_t i = 0 ; i < str.size() ; ++i) {
        index += (int)str[i];
    }
    return index;
}
```

```
size_t hash(const std::string& str) {
    size_t index = 0;
    for (size_t i = 0 ; i < str.size() - 1 ; ++i) {
        index += ((int)str[i] * int(str[i + 1]) * (i + 5)) & (((int)str[i] + int(str[i + 1]) * i *
    }
    return index * str.size();
}
```

`std::unordered_set<K>` AND `std::unordered_multiset<K>` TRAITS #1

- Implemented as a hash table
- It may or may not be cache friendly
  - Hash tables are often created as a hybrid of `std::vector<T>` and `std::list<T>`
- Values are not sorted
- Unique keys (`set`) or repeating keys are (`multiset`)

`std::unordered_set<K>` AND `std::unordered_multiset<K>` TRAITS #2

The hash table provides the average time of adding, removing, accessing and modification of `O(1)`. The worst time for all operations is `O(n)`.

Advantages:

- Very fast searching
- Constant time of adding and removing items...
- Assuming that we have a good hash function

Perfect for read-only containers. Reading time is `O(1)`. Great for types with a good hash function.

std::unordered_set<K> AND std::unordered_multiset<K> METHODS

std::unordered_set<K> on cppreference.org

std::unordered_multiset<K> on cppreference.org

- adding an item: `insert()`, `emplace()`, `emplace_hint()`
- modifying/accessing an item: none
- first/last item: none
- size/is the container empty: `size()`, `empty()`
- clear unused memory: none
- start/end iterator: `begin()`, `end()`
- reverse iterator: none

`std::unordered_set<K>` AND `std::unordered_multiset<K>` METHODS #2

- constant iterator: `cbegin()`, `cend()`
- container cleaning: `clear()`
- preparing item for removal: none
- erasing items from memory: `erase()`
- replacement of the entire container: `swap()`
- counting elements matching a given key: `count()` (returns only 0 or 1 for `map` or any positive number from 0 to n for `multimap`)
- finding an element with a given key: `find()`

OPERATIONS COMPLEXITY

- Insertion/deletion
  - `O(1)` - average case
  - `O(n)` - worst case
- Access
  - `O(1)` - average case
  - `O(n)` - worst case
- Searching
  - `O(1)` - average case
  - `O(n)` - worst case

MEMORY USAGE

- `n * (sizeof(K) + sizeof(X*)) + n / bucket_size`
- `O(n)`
- Additional memory for vector of pointers is used (bucket list)
- Additional small constant memory for internal data is used (begin, end, size, hash, key_eq, allocator)

ITERATOR INVALIDATION

- All read only operations, `swap`, `std::swap`: never
- `clear()`, `rehash()`, `reserve()`, `operator=()`: always
- `insert()`, `emplace()`, `emplace_hint()`, `operator[]()`: only if causes rehash
- `erase()`: only to the erased element

EXAMPLE OF std::unordered_set<K> AND std::unordered_multiset<K> USAGE

```
std::unordered_set<std::string> set{"Ala", "Ma", "Kota",
                                    "A", "Kot", "Ma", "ALE"};
for (const auto el : set) {
    std::cout << el << ' ';
}
std::cout << '\n';
set.insert("Ala");
set.insert("Ala");
set.insert("Ala");
for (const auto el : set) {
    std::cout << el << ' ';
}
std::cout << '\n';
```

`unordered_map<K, T>` AND `unordered_multimap<K, T>`

- Guess the features
- Guess the implementation

TASK

17. Measure the time of accessing/inserting the middle element of the `std::unordered_set` and `std::unordered_multiset` of size 100K and 1M.

ASSOCIATIVE CONTAINERS

SUMMARY

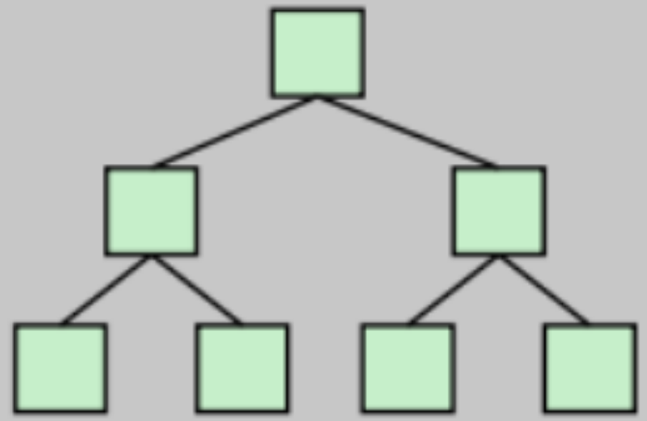
CODERS SCHOOL

WHAT DO YOU REMEMBER ABOUT ASSOCIATIVE CONTAINERS?

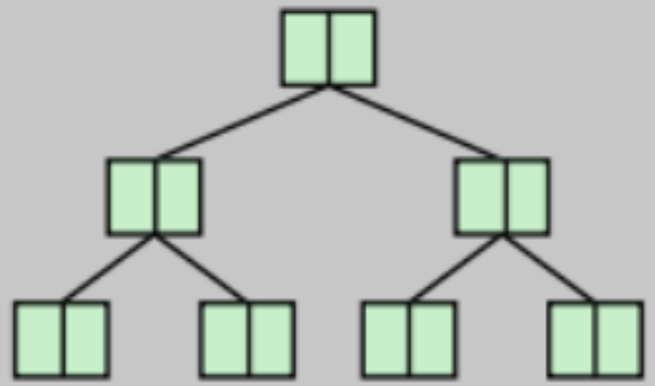
1. 8 containers - mention them
2. Which of them are cache-friendly?
3. Operations complexity
4. Memory consumption
5. Iterators invalidation

Associative Containers:

Set/Multiset:



Map/Multimap:



Unordered Containers:

Unordered Set/Multiset:

Unordered Map/Multimap:

Container	Insertion	Access	Erase	Find	Persistent Iterators
vector / string	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)	Sorted: O(log n) Other: O(n)	Pointers only
list / forward_list	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	O(n)	Yes
set / map	O(log n)	-	O(log n)	O(log n)	Yes
unordered_set / unordered_map	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	Pointers only
priority_queue	O(log n)	O(1)	O(log n)	-	-

CONTAINER ADAPTORS

CODERS SCHOOL

WHAT IS A CONTAINER ADAPTOR?

Container adaptors provide a different interface for sequential containers.

They are NOT a new, different kind of containers.

CONTAINER ADAPTORS

- stack<T>
- queue<T>
- priority_queue<T>

WHAT ARE THE DIFFERENCES BETWEEN THEM?

LET'S HAVE A DISCUSSION!

Consider:

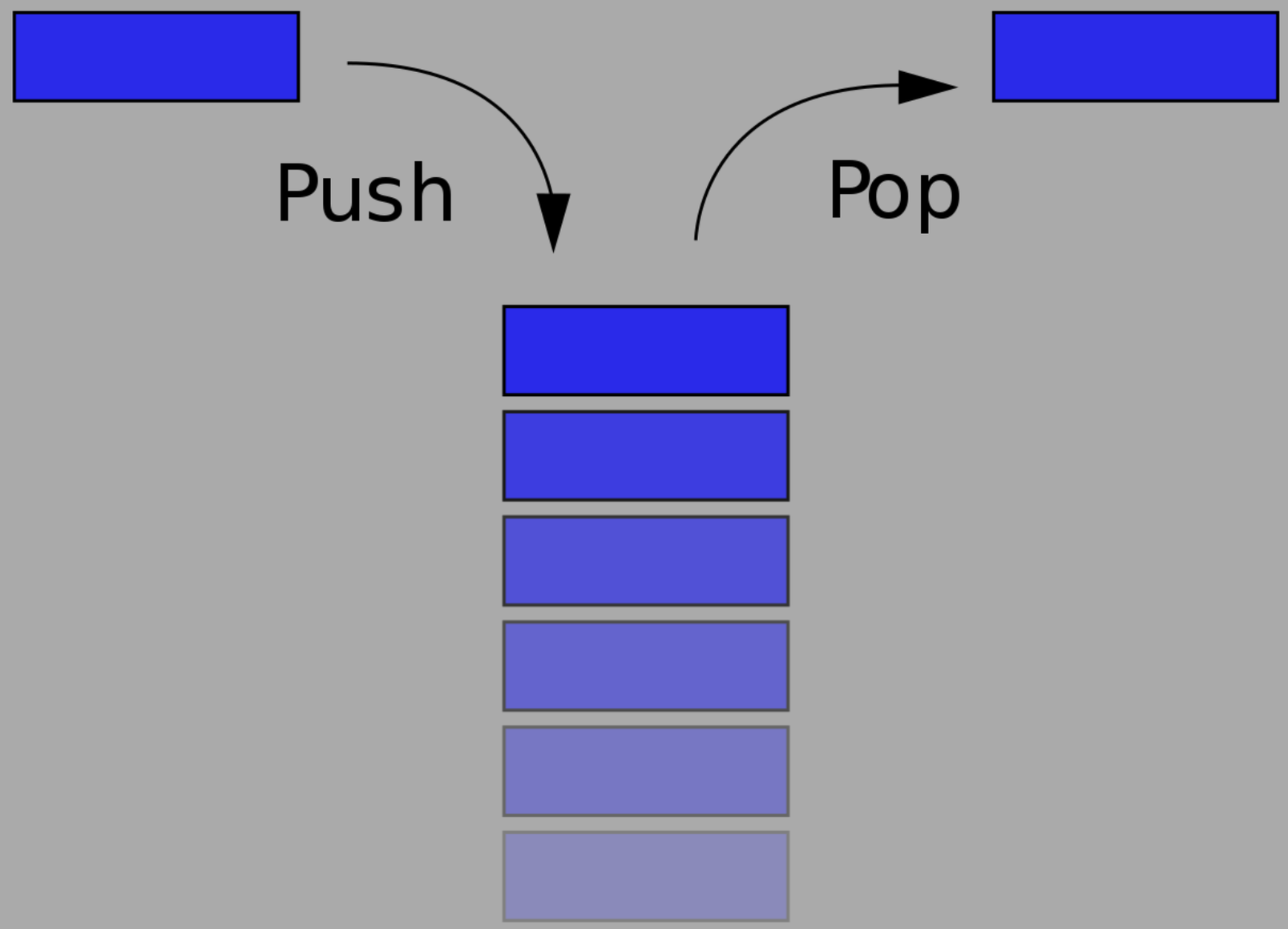
- Available operations
- Default underlying container
- Possible underlying containers

std::stack<T>

LIFO DATA STRUCTURE

CODERS SCHOOL

LIFO



`std::stack<T>` TRAITS

- The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.
- `std::deque<T>` is the default underlying container
- The underlying container can be any of:
  - `std::vector<T>`
  - `std::deque<T>`
  - `std::list<T>`

`std::stack<T>` METHODS

`std::stack<T>` ON CPPREFERENCE.ORG

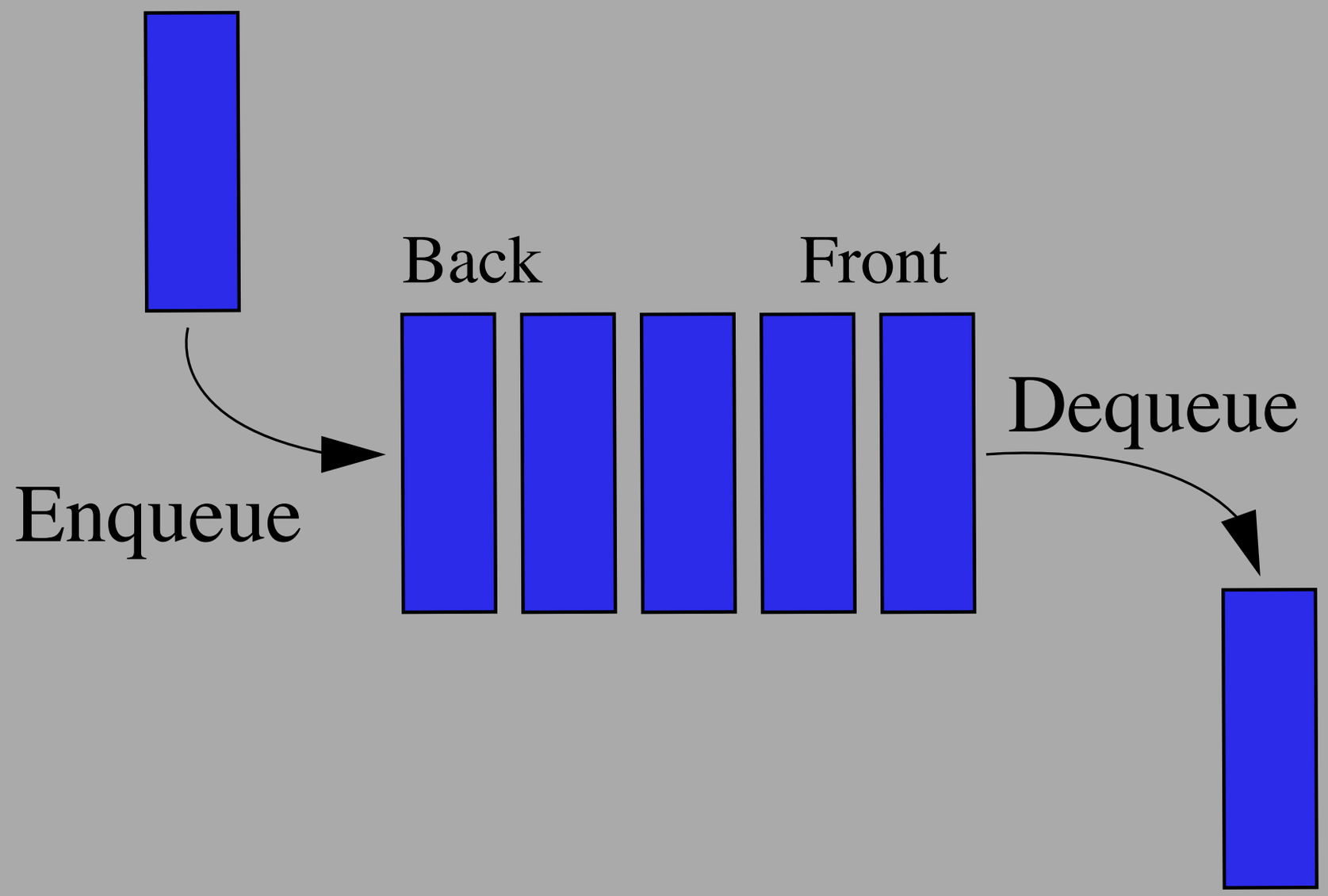
- adding an item: `push()`, `emplace()`
- accessing an item: `top()`
- removing an item: `pop()`
- size/is the container empty: `size()`, `empty()`

std::queue<T>

FIFO DATA STRUCTURE

CODERS SCHOOL

FIFO



`std::queue<T>` TRAITS

- The queue pushes the elements on the back of the underlying container and pops them from the front.
- `std::deque<T>` is the default underlying container
- The underlying container can be any of:
  - `std::deque<T>`
  - `std::list<T>`

`std::queue<T>` METHODS

`std::queue<T>` ON CPPREFERENCE.ORG

- adding an item: `push()`, `emplace()`
- accessing an item: `front()`, `back()`
- removing an item: `pop()`
- size/is the container empty: `size()`, `empty()`

std::priority_queue<T>

HEAP DATA STRUCTURE

CODERS SCHOOL

`std::priority_queue<T>` TRAITS

- Provides constant time lookup of the largest (by default) element, at the expense of logarithmic insertion
- A user-provided comparator can be supplied to change the ordering
- Elements are sorted on insertion
- Heap == sorted stack
- `std::vector<T>` is the default underlying container
- The underlying container can be any of:
  - `std::vector<T>`
  - `std::deque<T>`

`std::priority_queue<T>` METHODS

`std::priority_queue<T>` ON CPPREFERENCE.ORG

- adding an item: `push()`, `emplace()`
- accessing an item: `top()`
- removing an item: `pop()`
- size/is the container empty: `size()`, `empty()`

OTHER CONTAINERS

CODERS SCHOOL

OTHER CONTAINERS

- std::string/std::wstring
- std::bitset<N>
- std::valarray<T>
- std::tuple<T>

std::bitset<N>

CODERS SCHOOL

`std::bitset<N>` TRAITS

- The class for representing and manipulating arrays of bits
- It supports testing if any/all/none bits are set
- It supports setting/resetting/flipping single bits
- It keeps all elements on stack
- `std::vector<bool>` is an alternative, but it keeps elements on heap and does not provide fancy functions

std::bitset<N> METHODS

std::bitset<N> ON CPPREFERENCE.ORG

EXAMPLE

```
std::bitset<4> b{"0011"};
std::cout << "bitset" << b << '\n';

// bitset supports bitwise operations:
b |= 0b0100;                    assert(b == 0b0111);
b &= 0b0011;                    assert(b == 0b0011);
b ^= std::bitset<4>{0b1100};    assert(b3 == 0b1111);

// operations on the whole set:
b3.reset();                     assert(b3 == 0);
b3.set();                       assert(b3 == 0b1111);

// operations on individual bits:
b3.set(1, true);                assert(b3 == 0b0010);
b3.set(1, false);               assert(b3 == 0);
b3.flip(2);                     assert(b3 == 0b0100);
b3.reset(2);                    assert(b3 == 0);

// subscript operator[] is supported:
b3[2] = true;                   assert(true == b3[2]);
```

std::valarray<T>

CODERS SCHOOL

`std::valarray<T>` TRAITS

- The class for representing and manipulating arrays of values.
- It supports element-wise mathematical operations and various forms of generalized subscript operators, slicing and indirect access.
- Expressions such as `v1 = a * v2 + v3;` are expanded to loops that executes `v1[i] = a * v2[i] + v3[i];`

std::valarray<T> METHODS

std::valarray<T> ON CPPREFERENCE.ORG

EXAMPLE

```
#include <iostream>
#include <valarray>

template <typename Container>
void print(const Container& c) {
    std::cout << '\n';
    for (auto element : c) {
        std::cout << element << ' ';
    }
    std::cout << '\n';
}

int main()
{
    std::valarray<int> data = {0,1,2,3,4,5,6,7,8,9};
    std::cout << "Initial valarray:";
    print(data);

    data[data > 5] = -1;
    // the type of data > 5 is std::valarray<bool>
    // the type of data[data > 5] is std::mask_array<int>
```

STL CONTAINERS

POST-WORK

CODERS SCHOOL

TASK DESCRIPTION

`interval_map<K,V>` is a data structure that efficiently associates intervals of keys of type K with values of type V.

`interval_map<K, V>` is implemented on top of `std::map`.

Each key-value-pair (`k`,`v`) in the `interval_map` means that the value `v` is associated with the interval from `k` (including) to the next key (excluding).

EXAMPLE

```
std::map m = {{0,'A'}, {3,'B'}, {5,'A'}};
```

represents the mapping

```
0 -> 'A'
1 -> 'A'
2 -> 'A'
3 -> 'B'
4 -> 'B'
5 -> 'A'
6 -> 'A'
7 -> 'A'
... all the way to numeric_limits<int>::max()
```

The representation in the map must be canonical, that is, consecutive map entries must not have the same value: `..., (0,'A'), (3,'A'), ...` is not allowed.

Initially, the whole range of K is associated with a given initial value, passed to the constructor of the `interval_map<K,V>` data structure.

DOWNLOAD THE CODE

Your task is to implement the assign member function of this data structure.

POST-TEST

STL containers post-test

The link will be sent in the next week with all materials.

STL CONTAINERS

RECAP

CODERS SCHOOL

HOW DID YOU ENJOY THIS TRAINING SESSION?

WHAT WAS THE MOST SURPRISING FOR YOU?

RECAP

Mention containers from these groups:

1. Sequence containers
2. Associative containers
3. Adaptors
4. Other containers

WHAT IS THE DIFFERENCE BETWEEN `std::stack` AND `std::priority_queue`?

`std::priority_queue` is a sorted `std::stack`. Sorting is done during insertion.

WHAT IS THE COMPLEXITY OF OPERATIONS OF `std::deque`?

- Insertion/deletion
  - `O(1)` - at the end or at the beginning
  - `O(n)` - other
- Access
  - `O(1)`
- Searching
  - `O(n)`

WHAT IS THE ROLE OF `operator[]` IN `std::map`?

It returns a reference to value under the provided key. If it does not exist, it is created with the default constructor.

WHICH CONTAINER SHOULD YOU CHOOSE?

SEQUENTIAL #1

- Use `std:vector` as your default sequential container, especially as an alternative to built-in arrays
- If size is known in advance, use `std::array` instead of a built-in array
- If you add or remove elements frequently at both the front and back of a container, use `std::deque`
- Use a `std::list` instead of `std::deque` if you need to insert/remove elements in the middle of the sequence
- Always prefer `std::forward_list` over `std::list`
- Do not use `std::list/std::forward_list` if you need random access to objects
- Prefer `std::vector` over `std::list` (cache memory)

WHICH CONTAINER SHOULD YOU CHOOSE?

SEQUENTIAL #2

- If all your elements require a significant amount of memory do not use `std::array` (stack limitations)
- The smaller your elements are, the more probable you should use `std::vector`
- The bigger your elements are, the more probable you should consider `std::list/std::forward_list`
- If copying elements is very expensive and insertions will be frequent, consider `std::list/std::forward_list`
- `std::string` is almost always better than a C-string

WHICH CONTAINER SHOULD YOU CHOOSE?

ASSOCIATIVE

- Prefer unordered versions of associative containers. Only if element order matters use ordered versions.
- If you need multiple entries for the same key, use `std::unordered_multimap`, or if element order matters, `std::multimap`
- If you need all your elements to be sorted all the time use `std::set`, `std::multiset` or `std::priority_queue`
- If you need sorting occasionally, consider using `std::vector` and sort it with `std::sort()` when necessary

WHICH CONTAINER SHOULD YOU CHOOSE?

GENERAL ADVICE

- By default use `std::vector`
- Measure, if you think that something else might be better.
- Use `std::map` if you need to store key and value pairs and you like accessing it with `operator[]`, but for better performance on small amount of elements you can use `std::vector<std::pair<Key, Value>>`

Why you should avoid Linked Lists - Bjarne Stroustrup

PRE-TEST ANSWERS 1/3 🤯

MARK TRUE STATEMENTS ABOUT STL CONTAINERS

1. In `std::deque`, items can be added from both sides using `push_front()` and `push_back()`, respectively.
2. `std::set` supports random access to contained elements.
3. `std::vector` supports random access to contained elements.
4. `std::deque` supports random access to contained elements.
5. `std::multimap` allows multiple values to be stored for every key value.

PRE-TEST ANSWERS 2/3 🤯

WHICH OF THE FOLLOWING CONTAINERS ARE CONTIGUOUS?

1. `deque`
2. `vector`
3. `unordered_set`
4. `array`
5. `forward_list`

PRE-TEST ANSWERS 3/3 🤯

WHAT IS THE COMPLEXITY OF OPERATIONS ON `std::unordered_map`?

1. Inserting/removing `O(log n)`, accessing `O(1)`, searching `O(log n)`
2. Inserting/removing `O(1)`, accessing `O(1)`, searching `O(1)` average-case
3. Inserting/removing `O(n)`, accessing `O(n)`, searching `O(n)` worst-case
4. Inserting/removing `O(nlog n)`, accessing `O(log n)`, searching `O(n)`
5. Inserting/removing `O(nlog n)`, accessing `O(1)`, searching `O(log n)`

RATE THIS TRAINING SESSION

THANK YOU!