SWEG2102 Fall 2026

Fundamentals of Programming II

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia

Lecture 13

Standard Template Library (STL) - Part I

C++ **Standard ISO/IEC 14882**
Programming Language

Outline

Topics Covered

01 Associative Containers (Sorted)

02 Sorted Sets

03 Unordered Containers

04 Container Adapters

05 Advanced String Components

06 Practice & Strategy

Learning Objectives

By the end of this lecture you will be able to:

- **Understand** How maps differ fundamentally from sequences
- Master **sorted containers** how red-black trees ensure O(log n) operations
- Master **unordered containers** handling collisions and load factors
- Use **container adapters** when specialized interfaces enhance code clarity
- **Optimize with strings** advanced std::string functions;
- Make **informed decisions** Choose containers based on operations
- **Avoid common pitfalls** Key immutability, iterator invalidation, hash function design

Categorizing Containers

1. **Sequence Containers (Part I)** — *Maintain insertion order. Access by position/index.*
2. **Associative Containers (Sorted)** — *Organize by key; maintain sorted order. O(log n). Red-black trees.*
3. **Unordered Associative Containers** — *Organize by hash; no guaranteed order. Average O(1). Hash tables.*
4. **Container Adapters** — *Constrain interfaces on sequence containers; specialized semantics.*

Maps: Introduction

std::map<Key, Value> - Key-Value Associative Container

Definition

- Stores unique key-value pairs. Each key maps to exactly one value.
- The building block. Access via .first (key) and .second (value).
- Maintains ascending order by key. Enables range queries and ordered iteration.

```
#include <map>   ->   std::map<K, V>
```

Sorted Map Visualization

Key	Value
apple	5
banana	3
cherry	7
date	2

O(log n) *All operations*

Sorted *Key order*

Unique *Keys only*

Stable *Iterators*

Maps: Internal Mechanics

Red-Black Tree - Self-Balancing BST

Coloring Rule
Every node is colored RED or BLACK.

Black Depth
Every root-to-leaf path has the same number of black nodes.

No Double Red
Red nodes never have red children.

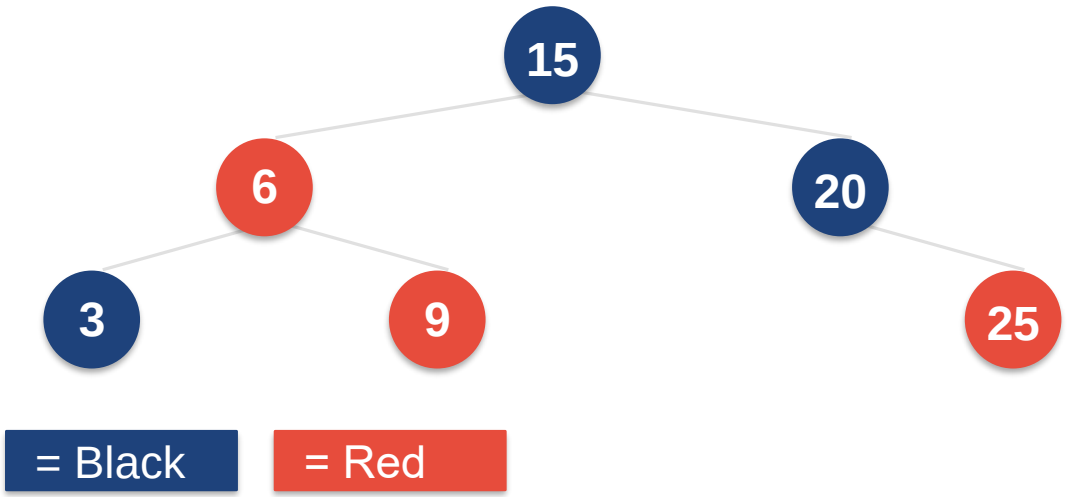
Height Guarantee
These invariants ensure height = O(log n).

Iterator Stability
Insert/delete only invalidates the erased iterator.

Operation Complexity

Operation	Complexity
Insertion	O(log n)
Search	O(log n)
Deletion	O(log n)
Range query	O(log n + k)

Simplified Red-Black Tree



Maps: Syntax

```cpp
#include <map>
#include <utility>  // std::pair, std::make_pair

// Default: empty map
std::map<std::string, int> dictionary;

// Initializer list (C++11)
std::map<std::string, int> fruits = {
    {"apple", 5}, {"banana", 3}, {"cherry", 7}
};

// operator[] - creates key if missing!
std::map<int, std::string> ids;
ids[1] = "Alice";

// insert() - returns pair<iterator, bool>
ids.insert({2, "Bob"});
ids.insert(std::make_pair(3, "Charlie"));
```

```cpp
// C++17 structured bindings
auto [it, inserted] = ids.insert({1, "Updated"});
// inserted == false: key 1 already exists

// Safe access: throws out_of_range
try {
    std::cout << fruits.at("grape");
} catch (const std::out_of_range& e) {
    std::cout << "Key not found\n";
}

// find(): never creates entries
auto it2 = fruits.find("banana");
if (it2 != fruits.end())
    std::cout << it2->first << " -> " << it2->second;
```

Maps: Insertion

operator[ ]	insert( )	emplace( )
`m["key"] = value;`	`m.insert({"key", val});`	`m.emplace("key", 42);`
Creates key if missing	Preserves existing keys	Constructs in-place
Returns value reference	Returns pair<iter,bool>	No temporary pair
Convenient but risky	Explicit, safe	Most efficient
Creates missing entries with default value. Dangerous in loops.	*If key exists, fails silently. bool indicates success.*	*C++11. Avoids temporary pair. Best for expensive types.*

operator[ ] -> O(log n) **insert( ) -> O(log n)** **emplace( ) -> O(log n)**

Maps: Search & Access

find(key)

```
auto it = m.find(key);
```

Returns iterator or end(). O(log n).
Never creates entries. Safe.

count(key)

```
if (m.count(key)) { ... }
```

Returns 0 or 1. O(log n).
Concise existence check.

at(key)

```
value = m.at(key);
```

Returns reference. Throws out_of_range.
O(log n). Never creates entries.

Maps: Iteration

```cpp
// Range-based for (C++17 structured bindings)
for (const auto& [key, value] : fruits) {
    std::cout << key << " -> " << value << "\n";
}

// C++14 / no structured bindings:
for (const auto& pair : fruits) {
    std::cout << pair.first << " -> "
              << pair.second << "\n";
}

// Iterator loop (fine-grained control):
for (auto it = m.begin(); it != m.end(); ++it) {
    std::cout << it->first << ": "
              << it->second << "\n";
}

// Reverse iteration (descending key order):
for (auto it = m.rbegin(); it != m.rend(); ++it) {
    // process in reverse
}

// Range query with lower/upper_bound:
auto lo = m.lower_bound("banana");
auto hi = m.upper_bound("date");
for (auto it = lo; it != hi; ++it) {
    std::cout << it->first << "\n";
}
// equal_range returns both as pair:
auto [lo2, hi2] = m.equal_range("cherry");
```

Maps: Performance

Operation	map / set	unordered_map	vector	list
Insert	O(log n)	O(1) avg	O(n)	O(1)*
Search	O(log n)	O(1) avg	O(n)	O(n)
Delete	O(log n)	O(1) avg	O(n)	O(1)*
Ordered iter?	Yes	No	N/A	N/A

25,000x faster

1M elements: vector search ~500,000 comparisons vs. map ~20 comparisons.

Worst-case guarantee

O(log n) is guaranteed always. unordered_map worst-case is O(n).

Use map when

Sorted access, range queries, or worst-case guarantees matter.

Maps: Example Program

```cpp
#include <map>
#include <iostream>

int main() {
  // Registry: employee name -> department
  std::map<std::string,std::string> registry;

  registry["Alice"] = "Engineering";
  registry["Bob"]   = "Marketing";
  registry["Carol"] = "Engineering";

  // Lookup — O(log n)
  if (registry.count("Alice"))
    std::cout << registry["Alice"]; // Engineering

  // Iterate in sorted key order
  for (auto& [name, dept] : registry)
    std::cout << name << ": " << dept;
}
```

Key Observations

operator[]

Creates a default value if key is missing.

count(key)

Returns 0 or 1 (keys are unique). Safe way to check membership.

Sorted output

Iterating always yields keys in ascending order.

Erase

registry.erase("Bob") removes key in O(log n).

Multimap: Introduction

Feature	std::map	std::multimap
Duplicate keys	No	Yes
operator[]	Yes	No
insert() result	pair<iter,bool>	iterator
find() result	One element	One element
Get all by key	Direct	equal_range()
Count per key	0 or 1	0, 1, or more
Complexity	O(log n)	O(log n)

Multimap: Operations

```cpp
#include <map>  // also covers multimap

std::multimap<std::string, int> m;

// insert() always succeeds (no bool!)
m.insert({"apple", 5});
m.insert({"apple", 3});  // Allowed!
m.insert({"apple", 7});  // Three apples

// find() -> only ONE element
auto it = m.find("apple");
// it points to just one apple, not all three

// equal_range() -> ALL elements with key
auto [lower, upper] = m.equal_range("apple");
for (auto it2 = lower; it2 != upper; ++it2) {
    std::cout << it2->first << " -> "
              << it2->second << "\n";
}
// Prints: apple->5, apple->3, apple->7

// count() returns actual multiplicity
int n = m.count("apple");  // n == 3

// lower_bound / upper_bound still work
auto lb = m.lower_bound("apple"); // first apple
auto ub = m.upper_bound("apple"); // first non-apple
// [lb, ub) == equal_range("apple")
```

Multimap State Visualization

apple	5
apple	3
apple	7
banana	2
banana	6

insert() -> iter only

No bool: multimap never rejects duplicates.

find() limitation

Returns one; use equal_range() for all.

count() usefulness

Returns actual count (not just 0 or 1).

equal_range() idiom

[lower, upper) is the canonical access pattern.

Multimap: Example Program

```cpp
#include <map>
#include <iostream>

int main() {
  // dept -> multiple employee names
  std::multimap<std::string,std::string> dept;

  dept.insert({"Eng", "Alice"});
  dept.insert({"Eng", "Bob"});   // duplicate key OK
  dept.insert({"HR",  "Carol"});
  dept.insert({"HR",  "Dave"});

  // Retrieve all Eng employees
  auto [lo, hi] = dept.equal_range("Eng");
  for (auto it = lo; it != hi; ++it)
    std::cout << it->second; // Alice, Bob

  std::cout << dept.count("Eng"); // 2
```

Multimap Mechanics

insert() always inserts

Always adds a new entry, never rejecting duplicate keys.

equal_range() idiom

Returns the start and end boundaries to loop through a key's values.

count(key)

Returns the exact number of entries for a key, not just 0 or 1.

Quick Check: Maps vs. Multimaps

Q1 What is the main difference between std::map and std::multimap?

A) Both allow duplicate keys

B) Maps have unique keys; multimaps allow duplicates

C) Multimaps are faster

D) Maps can't use find()

Answer: B

Q2 Why doesn't std::multimap support operator[]?

A) Performance reasons

B) Iterator stability

C) Ambiguous: which value for duplicate keys?

D) operator[] is deprecated in C++17

Answer: C

Q3 How do you retrieve all values for a multimap key?

A) Use find() and loop

B) Use equal_range() to get [lower, upper)

C) Use operator[]

D) Iterate the whole container

Answer: B

Sets: Introduction

std::set<T> - Sorted Unique Collection

Definition

Stores unique elements in sorted order.
Conceptually: a map with only keys, no values.

Declaration - std::set<ElementType>

Uniqueness - Duplicate insertions are rejected.

Sorted Order - in ascending order

Set vs Map - Key Distinction

```
std::set<int>  s;
```

2	5	7	11	13	17

-> Each element appears exactly once, in sorted order

Feature	std::set	std::map
Template params	1 (element)	2 (key, value)
operator[]	No	Yes
Value immutable	Yes	Keys only
Complexity	O(log n)	O(log n)

Sets: Internal Mechanics

Red-Black Tree Implementation

Like maps, sets use red-black trees for O(log n) insertion, deletion, and search.
The height of the tree stays logarithmic, ensuring all operations remain efficient.

Immutability Constraint

Elements in a set are const. You cannot modify them in-place.
Doing so could violate sort order and corrupt the tree structure.

Solution: Erase & Re-insert

To 'change' an element: erase the old one (O(log n)), then insert the new value (O(log n)).
This is the only safe path.

Sorting Customization

Default uses std::less<T> (ascending). Supply std::greater<int> or a custom comparator for descending or custom ordering.

Sets: Common Operations

Operation	Returns	Key Behaviour
insert(x)	pair<it,bool>	Rejected silently if duplicate; bool=false
find(x)	iterator	Returns end() if not found; O(log n)
count(x)	0 or 1	Quick membership test idiom
erase(x)	size_t	By value, by iterator, or by range
begin/end	iterator	Sorted ascending order always

Uniqueness

Duplicates fail to insert and return false.

Always Sorted

Elements always iterate in ascending order.

count() vs find()

Use count() checks existence;, and find() returns the iterator.

Sets: Math-like Operations

`std::set_union()`

{1,2,3,4,5}

All elements from either set

`std::set_intersection()`

{3}

Elements common to both sets

`std::set_difference()`

{1,5}

In first set, NOT in second

`std::set_symmetric_difference()`

{1,2,4,5}

In one set or the other, not both

Sets: Multiset

std::set<T>

- ✓ Unique elements only
- ✓ Sorted ascending
- ✓ Elements are const (immutable)

insert() returns pair<it, bool>
count() returns 0 or 1
e.g.: { 2, 3, 5, 7 }

std::multiset<T>

- ✓ Duplicate elements allowed
- ✓ Sorted ascending
- ✓ Elements are const (immutable)

insert() returns only an iterator
count() returns 0, 1, 2 … n
e.g.: { 2, 3, 3, 5, 5, 5, 7 }

```cpp
std::multiset<int> m;
m.insert(5); m.insert(5); m.insert(5); // Three 5s — all accepted!

int c = m.count(5);
// c = 3

auto [lo, hi] = m.equal_range(5);
// [lo, hi) spans all three 5s
for (auto it = lo; it != hi; ++it) std::cout << *it; // 5 5 5
```

Sets: Example Program

```cpp
std::vector<int> data = {5,2,8,2,9,5,1,8,3,5};

// 1. Deduplicate & sort — one liner!
std::set<int> s(data.begin(), data.end());
// s = {1,2,3,5,8,9}

// 2. Membership test  O(log n)
if (s.count(8)) { /* 8 exists */ }

// 3. Range query — elements in [3, 8)
auto lo = s.lower_bound(3), hi = s.lower_bound(8);
for (auto it=lo; it!=hi; ++it) cout << *it; // 3 5

// 4. Set union with another set
std::set<int> other={1,4,6,9}, u;
std::set_union(s.begin(),s.end(),other.begin(),other.end(),
    std::inserter(u, u.begin())); // {1,2,3,4,5,6,8,9}
```

Original

5 2 8 2 9 5 1 8 3 5

Deduplicated

1 2 3 5 8 9

Range [3,8)

3 5

Union

1 2 3 4 5 6 8 9

Unordered Containers: Overview

Ordered Containers

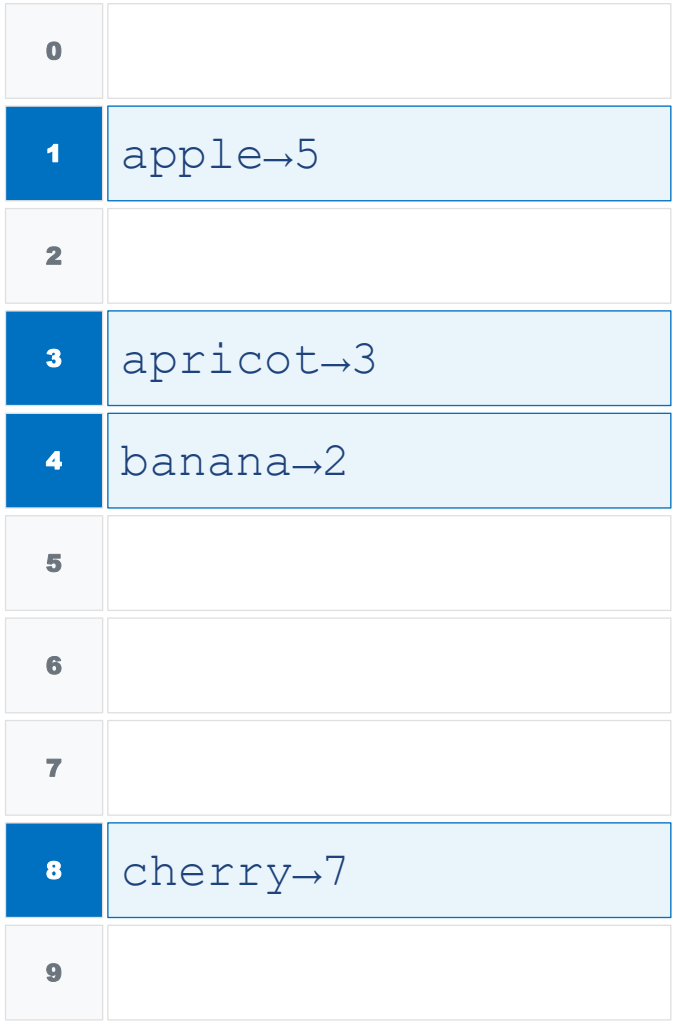
- **Engine:** Red-Black Tree with guaranteed O(log n) operations.
- **Benefits:** Automatically sorted keys, efficient range queries, and stable iterators.
- **When to use:** When you need sorted order, range searches, or strict O(log n) performance.

Unordered Containers

- **Engine:** Hash Table with average O(1) operations, but worst-case O(n) if keys collide.
- **Behavior:** Undefined iteration order with no support for range queries.
- **When to use:** When you need maximum lookup speed and element order does not matter.

💡 Choose Ordered when: sorted iteration, range queries, or worst-case guarantees matter.
Choose Unordered when: pure lookup speed on large datasets and order is irrelevant.

Unordered: Mechanics



Hash Function

h(key) → bucket index. A good function distributes keys uniformly, minimising collisions.

Collision Resolution

Multiple keys → same bucket? Resolved by chaining (list per bucket) or open addressing.

Load Factor λ = n / buckets

λ < 0.75 → sparse, fast. λ > 1.0 → chains grow, lookups degrade. Rehash triggers when threshold exceeded.

Rehashing

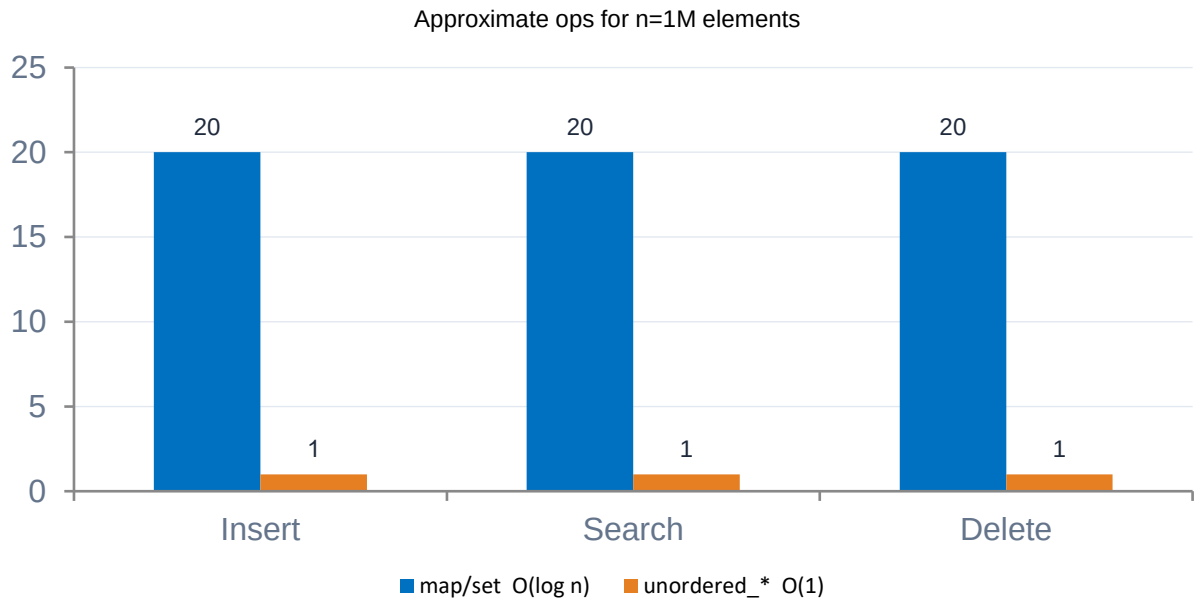
Allocates a larger bucket array and recomputes all hashes. Expensive O(n) but rare — amortised O(1).

Unordered Map/Set: Operations

```cpp
#include <unordered_map>
#include <unordered_set>

std::unordered_map<std::string,int> wc;
wc["apple"]=5; wc["banana"]=3; wc["cherry"]=7;

// Search  O(1) avg
auto it = wc.find("banana");
if (it != wc.end()) cout << it->second; // 3

// Iteration — order UNDEFINED
for (const auto& [k,v] : wc) cout << k << v;
// order may be: cherry, apple, banana …

// unordered_set
std::unordered_set<int> nums = {10,20,30};
nums.erase(20); // O(1) avg
if (nums.count(10)) { /* found */ }
```

Operation	map/set	unordered_*
insert / operator[]	O(log n)	**O(1) avg**
find / count	O(log n)	**O(1) avg**
erase	O(log n)	**O(1) avg**
Range query	**O(log n+k)**	O(n)
Sorted iteration	**✓ Yes**	✗ No
Worst-case guarantee	**O(log n)**	O(n)

Ordered vs. Unordered Performance



Practical Benchmark (n = 1,000,000 random integers)

map::find() ~40 comparisons (balanced tree depth ≈ 20)

unordered_map::find() ~1 operation (direct hash lookup)

Range query [1000,2000]: map O(log n + k) vs. unordered_map O(n)

Choose map / set when …

- Sorted iteration needed
- Range queries (lower/upper_bound)
- Worst-case O(log n) required
- Hash function unreliable

Choose unordered_* when …

- Fastest average lookup
- No ordering needed
- Large datasets (1M+ elements)
- Workload: insert/search/delete only

Unordered: Example Program

```cpp
std::unordered_map<std::string,int> word_freq;
std::vector<std::string> words = {"apple","banana","apple",
    "cherry","banana","apple","date","apple"};

// Count frequencies — O(1) per increment
for (const auto& w : words) word_freq[w]++;

// Query membership  O(1) avg
if (word_freq.count("cherry"))
    cout << word_freq["cherry"]; // 1

// 1M-element lookup benchmark
std::unordered_set<int> big_set;
for (int i=0; i<1'000'000; ++i) big_set.insert(i);
auto t0 = chrono::high_resolution_clock::now();
bool found = big_set.count(500000) > 0;
auto t1 = chrono::high_resolution_clock::now();
// Elapsed: ~10-50 nanoseconds  (essentially O(1)!)
```

Word Frequencies

```
apple: 4
banana: 2
cherry: 1
date: 1
(order undefined)
```

Unique Words Count

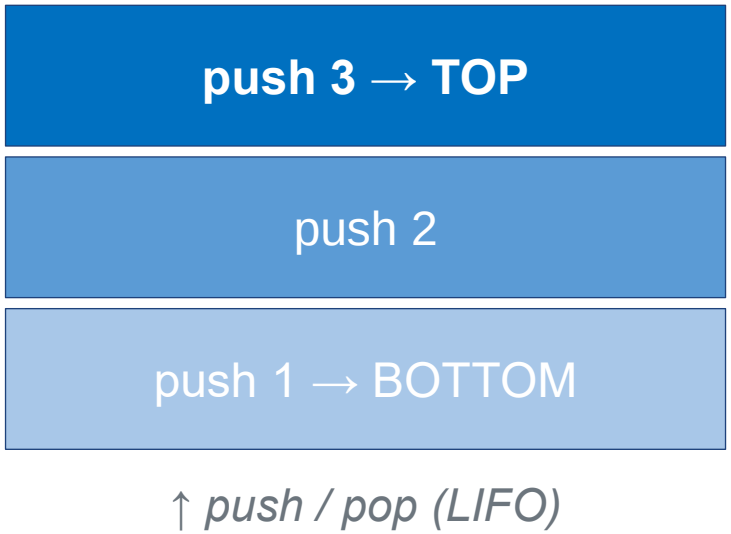
```
4 unique words
```

1M Lookup Time

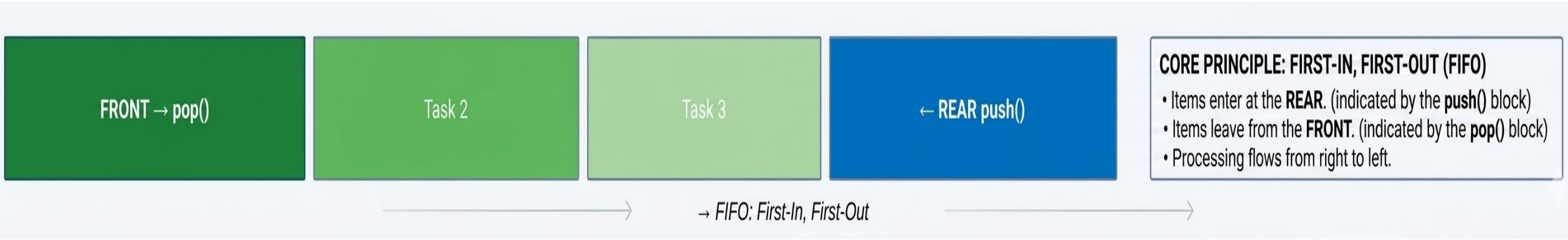
```
~10-50 nanoseconds
vs. map: ~400-600 ns
```

Container Adapters: Stack

Operation	Behaviour	Complexity
push(v)	Add to top	O(1) amortized
pop()	Remove top	O(1) amortized
top()	Read top	O(1)
size()	Count	O(1)
empty()	Check empty	O(1)



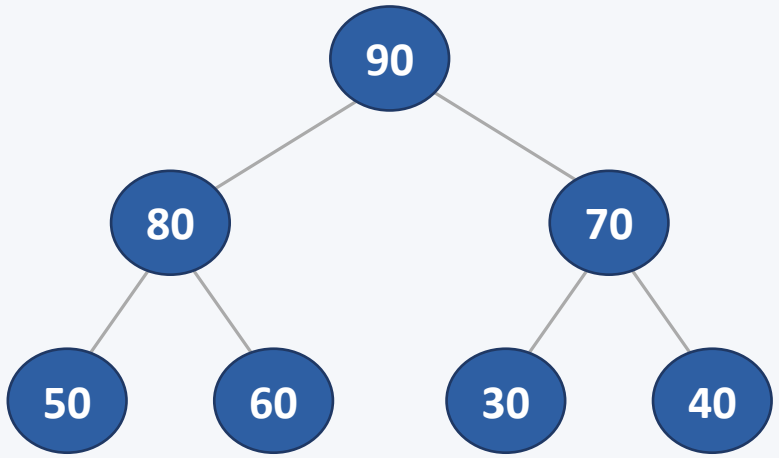
Container Adapters: Queue



Operation	Behaviour	Complexity
push(v)	Add to rear	O(1)
pop()	Remove front	O(1)
front()	Read front	O(1)
back()	Read rear	O(1)
empty()	Check empty	O(1)

Container Adapters: Priority Queue

Max-Heap Visualisation



↑ *top() always returns the largest element*

```
Stored as: [90, 80, 70, 50, 60, 30, 40]
```

Operation	Behaviour	Complexity
push(v)	Insert + re-heap	O(log n)
pop()	Remove top + re-heap	O(log n)
top()	Read maximum (or min)	O(1)
size()	Count elements	O(1)
empty()	Check if empty	O(1)

String Components

substr(pos, len) · O(k)

Extract k characters starting at pos. Returns a new string.

find(str, pos) · O(n·m)

Find first occurrence. Returns position or string::npos if not found.

replace(p,l,str) · O(n)

Replace substring at [p, p+l) with str.

compare(str) · O(n)

Lexicographic comparison. Returns <0, 0, or >0.

c_str() · O(1)

Returns const char* to internal buffer. Valid until string is modified.

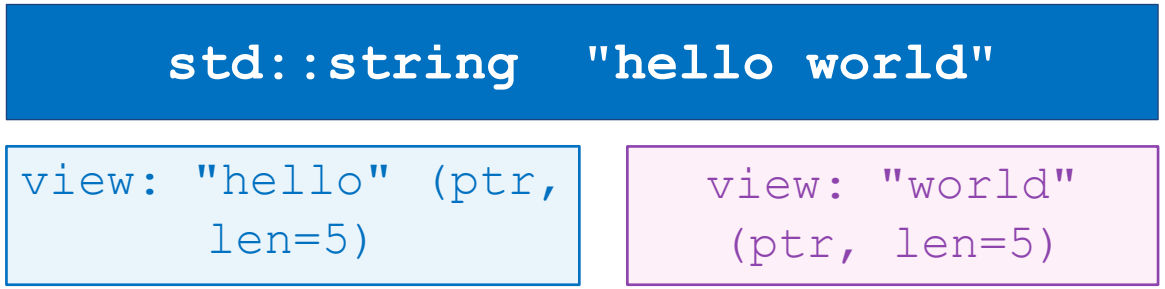
reserve(n) · O(n)

Pre-allocate capacity to avoid reallocations.

```cpp
std::string s = "hello world";
string sub  = s.substr(6, 5);   // "world"
size_t pos  = s.find("world");   // 6
s.replace(6, 5, "STL");          // s = "hello STL"
if (pos != std::string::npos) { /* found */ }
```

String View

Memory Layout



Key Advantage: Zero-Copy

string_view is just a pointer + length (16 bytes). No allocation, no copy — even for substrings.

⚠ Danger: Dangling Views

If the source string is modified or destroyed, the view becomes a dangling reference — undefined behaviour!

```cpp
// Without string_view – allocates copy!
void process(const std::string& s);
process(bigStr.substr(0, 5)); // ❌ heap allocation

// With string_view – zero copy ✅
void process(std::string_view sv);
process(std::string_view(bigStr).substr(0,5)); // no allocation

// Safe usage – source must outlive view!
std::string src = "hello world";
std::string_view sv = src;  // OK while src alive
```

Common Mistakes

1. Modifying Keys In-Place

```
m[5]->first = 10;  // ❌ COMPILE ERROR
```

```
string v = m[5]; m.erase(5); m[10] = v;// ✅
```

2. Using Erased Iterators

```
m.erase(it);  cout << it->first;  // ❌ undefined behaviour
```

```
it = m.erase(it);  // ✅ erase() returns next valid iterator
```

3. Erasing While Iterating

```
for (auto it=m.begin(); it!=m.end(); ++it)
if (cond) m.erase(it);  // ❌ it is now dangling
```

```
if (cond) it = m.erase(it); else ++it; // ✅
```

4. Hash Order Assumptions

```
Assuming unordered_map iterates in insertion order ❌
```

```
Use map/set if you need ordered iteration ✅
```

Best Practices

1. Need sorted order / range queries?	YES → map / set NO → unordered_map / set
2. Key-value pairs or just keys?	YES → map / unordered_map NO → set / unordered_set
3. Multiple values per key?	YES → multimap / multiset NO → map / set
4. Need LIFO, FIFO, or priority order?	YES → stack / queue / priority_queue NO → sequence or assoc. container
5. Need random access by index?	YES → vector NO → associative container

References

Textbooks

- **C++ How to Program** [10th edition], Deitel, P. & Deitel, H., Global Edition, Global Edition (2017).
- **Problem Solving With C++** [10th edition], Walter Savitch, University of California, San Diego, 2018.

Reference Books

- **Programming: Principles and Practice Using C++** by Bjarne Stroustrup, Addison-Wesley, 2014.
- **An Introduction to Programming with C++** (8th Edition), Diane Zak, Cengage Learning, 2016

Online Resources

- https://www.geeksforgeeks.org/cpp/c-plus-plus/
- https://www.w3schools.com/cpp/default.asp
- https://programiz.pro/resources/cpp
- https://www.hackerrank.com/domains/cpp
- https://cplusplus.com/doc/tutorial/

💡 ***Study Tip:*** *Don't just read the code! Retype the examples from these slides and resources into your IDE, compile them, and modify them to see what happens.*

Thank You!

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia