```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAXPAROLA 30
#define MAXRIGA 80

int main(int argc, char *argv[])
{
  int freq[MAXPAROLA] ; /* vettore di contatori
  delle frequenze delle lunghezze delle parole */
  char riga[MAXRIGA] ;
  int i, inizio, lunghezza ;
  FILE * f ;

  for(i=0; i<MAXPAROLA; i++)
    freq[i]=0 ;

  if(argc != 2)
  {
    fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
  }
  f = fopen(argv[1], "rt") ;
  if(f==NULL)
  {
    fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
  }

  while( fgets( riga, MAXRIGA, f ) != NULL )
```

High Level Programming

Associative containers

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

License Information

Introduction

- Associative containers support lookup and retrieval by a key
- The two primary associative containers are
  - Maps, whose elements are pairs **key-value**
    - The key is used to access the value
  - Sets, whose elements are just **keys**
    - The set support efficient query as to whether a key is present
- Each assciative container
  - Is either a **map** or a **set**
  - Requires **unique** keys or allows **multiple** keys
  - Stores elements in **order** or **not**

Associative containers

- The word
  - **Multi** indicates multiple keys
  - **Unordered** indicates the use of a hash function

Type	**Meaning**
map	Associative array; hold pairs key-value.
set	Container in which the key is the value.
multimap	A map in whch a key can appear multiple times.
multiset	A set in which a key can appear multiple times.
unordered_map	A map organized using a hash function.
unorderd_set	A set organized using a hash function.
unordered_multimap	Multi map organized using a hash function.
unordered_multiset	Multi set organized using a hash function.

Main operations

- The main operations on associative containers are
  - Insertion, extraction, and access

For a full list of operations (versions), please see the documentation

c is an instance of the container

Operation	Meaning
c.insert(v)	Insert element v in the associative container c.
c.emplace(argv)	Construct an element from argv and insert it in c. For map and set, argv is created and inserted only if the key is **not** already in the container.
c.erase(k)	Removes **every** element with key k from c.
c.erase(b,e)	Removes every element in the range denoted by the iterator b and e.
c[k]	Returns the element with key k. If k is not in c, **adds** a new value (value-initialized) with the key k.
c.at(k)	Check access to the element with key k. Throws an out_of_range exception if k is not in c.

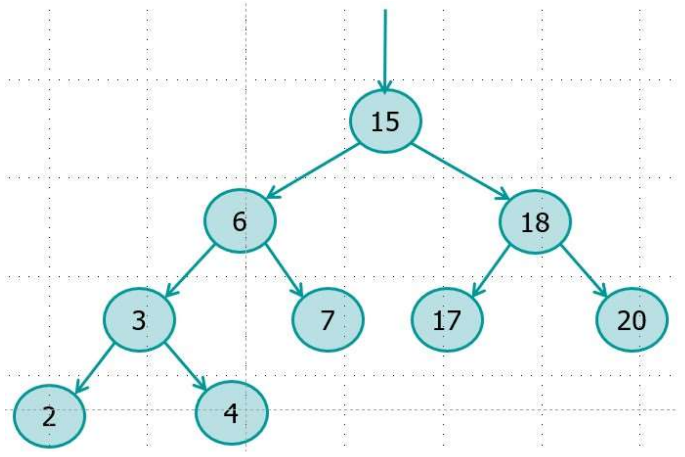
Extra operations

- Extra (more complex) operations on associative containers
  - Search a key or a key range

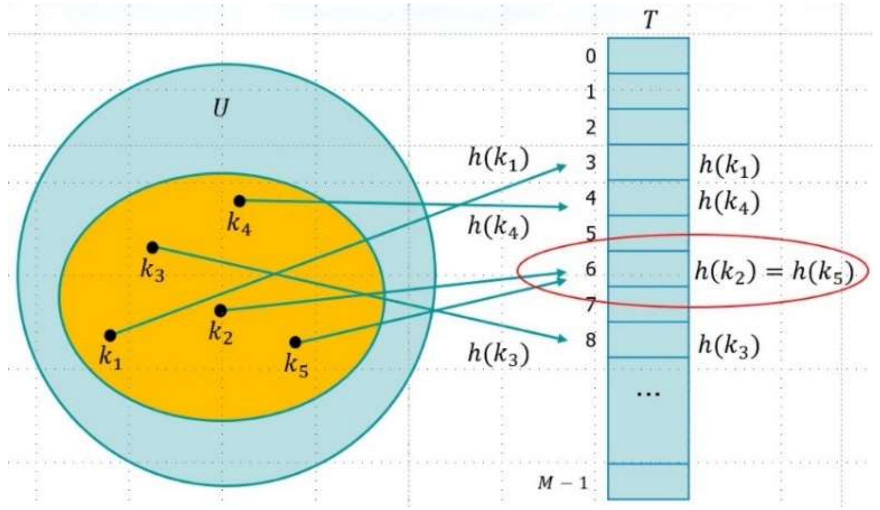
Operation	Meaning
c.find(k)	Returns an iterator to the **first** element with key k. returns an iterator pointing to the end of the container.
c.count(k)	Returns the number of elements with key k.
c.lower_bound(k)	Returns an iterator to the first element with key **not less** than k in c.
c.upper_bound(k)	Returns n iterator to the first element with key **greater** than k in c.
c.equal_range(k)	Returns a pair of iterators denoting the element with key k. If k is not present both members are c.end().

Maps

- Maps are associative containers consisting of pairs key-value
  - In maps, the keys are sorted



  - In unordered maps, there is no order among keys

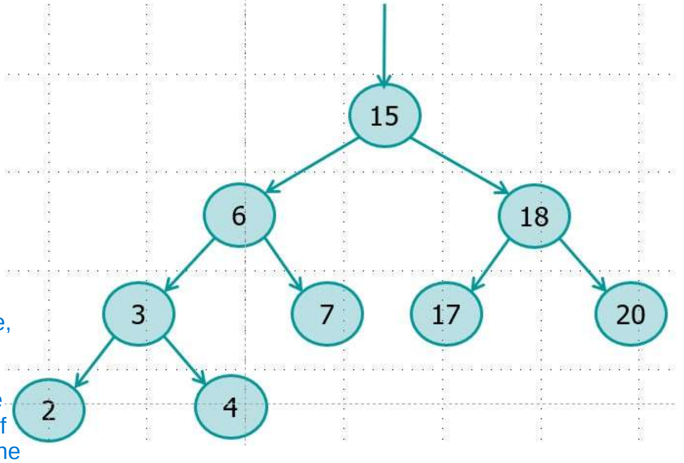


Maps

- Complexity for random access, search, insertion, and removal is
  - $O(\log N)$ for maps
    - Internally are a tree (usually AVL- or R/B-Tree)

In a well-balanced binary search tree, each time you traverse down a level in the tree, you effectively reduce the search space by half.
Here's why:
At each level: You make a decision to go left or right based on a comparison with the current node's value. This halves the search space because you're eliminating one of the subtrees.Each level represents a power of 2: If the tree has n nodes, its height (the maximum number of levels) is approximately log(n). This is because each level of the tree can accommodate twice as many nodes as the previous level.
So, by making binary decisions at each level and effectively halving the search space, the time complexity of operations like searching, inserting, and deleting becomes logarithmic with respect to the number of nodes in the tree.

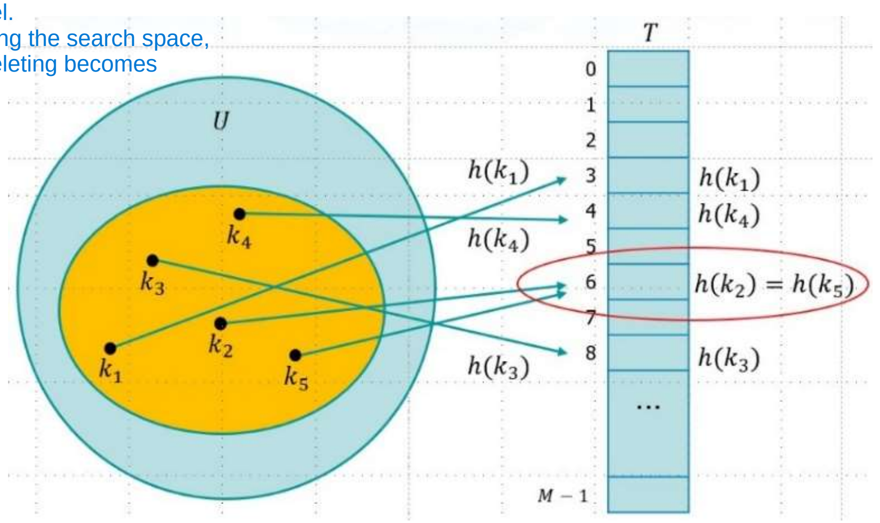
N is the number of element stored in the container



  - $O(1)$ for unordered maps
    - Internally are a hash–table using a hash–function (h)

Unordered maps are typically implemented using hash tables, also known as hash maps or hash dictionaries.
A hash table is a data structure that stores key-value pairs, where each key is hashed to a unique index in an array.

In an ideal scenario with a well-designed hash function and a sufficiently large array, the time complexity for key-based operations (search, insertion, and removal) in a hash table is O(1), which is constant time.



Maps

- Main characteristics
  - Defined in the header **map** or **unordered_map**
  - Maps and unordered maps have a very similar user interface
    - In a **map** there is no way to access keys or values in order
  - Keys are required to be **unique**
  - To check if a key exists, we may use the function **count**

Examples

Definitions

```
#include <map>

map<string,int> mymap;              // Empty map

// Map with three elements
map<string, string> authors = {
        {"Joyce","James"},
        {"Austen","Jane"},
        {"Dickens","Charles"}};
```

Examples

Insertions

Empty map

Equivalent insertion

```
map<string,size_t> word_count;

word_count.insert({"this",1});
word_count.insert(make_pair{"this",1});
word_count.insert(pair<string,size_t> ("this",1));
word_count.insert(map<string,size_t>::value_type("this",1));
```

The last line utilizes the insert function of the word_count map to add a new key-value pair constructed using the value_type alias, representing a pair with a key of type const string and a value of type size_t, with the key "this" and value 1. This approach is functionally equivalent to the other lines, providing a concise way to insert elements into the map.

See documentation for: key_type, mapped_type, value_type

- Elements of a map are objects of pair type
  - A pair holds two data members
  - Unlike other library types, these two data members are public
    - They are named **first** and **second**, respectively

The pair type

❖ Pairs have their own set of operations

Operation	Meaning
pair<T1,T2> p;	Defines a new pair p. The members of the pair are initialized following type T1 and T2.
pair<T1,T2> p(v1,v2);	Defines a new pair p. The members of the pair are initialized with v1 an v2.
make_pair(v1,v2)	Returns a pair initialized with v1 and v2.
p.first	Returns the (public) member of p named first (i.e., v1).
p.second	Returns the (public) member of p, named second (i.e., v2).
p1==p2 p1!=p2	Two pairs are equal if their first and second member are respectively equal.

Examples

Compute the absolute frequency of input words

```
#include <map>

map<string, size_t> word_count;
string word;

while (cin >> word)
  ++word_count[word];

for (const auto &w : word_count) {
  cout << w.first << "occurs " << w.second <<
     "time(s)." << endl;
}
```

Empty map

Insertion through subcripting

Access to the pair

Examples

Compute the absolute frequency of input words

```
#include <map>

map<string, size_t> word_count;
string word;

while (cin >> word)
  ++word_count[word];
```

Insertion through subcripting

Equivalent insertion: more verbose

```
while (cin >> word) {
  auto ret = word_count.insert ({word,1});
  if (!ret.second)
    ++ret.first->second;
}
```

The code reads input words one by one from the standard input cin using while (cin>> word)

For each word it attempts to insert the word into the map with initial count as one. The insert returns a pair of iterator ('ret.first') and a boolean ret.second. The first one points to the inserted element and the second one indicates whether the insertion took place successfully or not.

if the insertion was unsuccessful (the word already exists. Then it increments the count associated with the word.

This is achieved by accessing the iterator (ret.first) while points to the inserted element and then incrementing the second value component that it points to.

Word is in.
Increase the counter

Insert returns a pair.
The first member is an iterator to the element, the second is a bool.
If the key is not present, then the element is inserted and the bool is true.
If the key is already in the container, insert does nothing and the bool is false.

Examples

Iterating through a map

```
#include <map>

map<string, size_t> word_count;

...

auto it = word.count.begin();
while (it != word_count.end()) {
  cout << it->firt << occurs << it->second << times << endl;
  it++;
}
```

Map iterator

For a vector you have to do it like this but not for maps as in maps ->second and ->first is the element

```
auto it = authors.begin();
while (it != authors.end()) {
   cout << *it;
   it++;
}
```

In an unordered map, keys are in random order

Examples

Given a map {author,title}
Print all books by a specific author

```
#include <map>

map<string, string> books =
  {{"A1","B1"},{"A2","B2"},{"A3","B3"},...};

string my_author("Tolkien");

auto entries = books.count (my_author);
auto iter = books.find (my_author);

while (entries) {
  cout << iter->second << endl;
  ++iter;
  --entries;
}
```

Number of books with that author

Returns the number of elements with key k.

Variable iter is an interator to all books with the same author

Wow, read this.

This can't be possible with ordered maps cuz we can't have more than one key with the same value. So we use unordered maps.

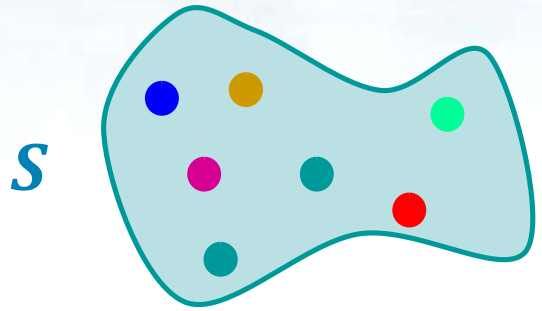
Examples

Operations with an unordered_map

```
#include <unordered_map>

std::unordered_map<std::string,double> um
{{"maier", 1.3}, {"huber", 2.7}, {"schmidt", 5.0}};

cout << um["huber"];           // Displays 2.7
couut << um.at("schmidt");     // Displays 5.0

auto search = um.find("schmidt");
if (search != um.end()) {
  // Returns an iterator pointing to a pair!
  string s = search->first;                // This is "schmidt"
  float val = search->second;              // This is 5.0
}

int n1 = um.count("schmidt");    // == 1
int n2 = um.count("blafasel");   // == 0
```

Sets

- A set is simply a collection of keys



- They are useful when we want to know whether a value is present
  - Sets are usually implemented using **trees** and traversed using iterators
  - Unordered sets are often implementd using **hash tables**

Examples

Definitions and basic operations

```
#include <set>

set<int> is = {0,1,2,3,4,5,6,7,8,9};

auto it1 = is.find(1);    // Returns an iterator
                         // to element with key=1

auto it2 = is.find(11);   // Returns an iterator to is.end()

auto n1 = is.count(1);    // Returns 1

auto n2 = is.count(11);   // Returns 0
```

if the set contains an element with the specified value it returns one else it returns 0. 1 exists in the set and 11 doesn't

Examples

Definitions

```
#include <set>

set<int> myset;

vector<int> iv = {2,4,6,8,2,4,6,8,2,4,6,8};

myset.insert(iv.begin(),iv.end());
// myet includes 4 elements {2,4,6,8}

myset.insert({1,3,5,7,1,3,5,7});
// myet now includes 8 elements {1,2,3,4,5,6,7}
```

Empty set

Do not forget that keys are unique

Iterators on sets

```
set<int> is = {0,1,2,3,4,5,6,7,8,9};
set<int>::iterator it = is.begin();
while (it |= is.end()) {
  cout << *it << endl;
  it++;
}
```

Displays 0,1,2,…,9 in order. In unordered_set the order is undefined.

Examples

Map

Compute the absolute frequency of input words

Set

Excluding a few words

List initialization

```
#include <map>
#include <set>

map<string, size_t> word_count;
set<string> exclude = { "The", "But", "And", "Or", "An", "A",
                        "the", "but", "and", "or", "an", "a"};
string word;

while (cin >> word)
  if (exclude.find(word)==exclude.end())
    ++word_count[word];

for (const auto &w : word_count) {
  cout << w.first << "occurs " << w.second <<
    "time(s)." << endl;
}
```

Or
exclude.count(word)==0

Exercise: Word Frequency Counter

- Write a C++ program that reads a paragraph of text (a line of text) from the user
  - Tokenize the input paragraph into words
    - Ignore punctuation, consider only alphabetic characters, and transform characters in lowercase
  - Create a map in which
    - Words (in the text) are keys
    - Values are frequencies (of that word in the text)
  - Display the list of unique words and their frequencies alphabetically
  - Find and display the total number of unique words in the paragraph

Exercise: Word Frequency Counter

- Prompt the user to enter a word and then search the map to display the frequency of that word
- Create a set containing the unique words from the paragraph
- Display the unique words in the set sorted alphabetically

Example

Input

```
This is a simple example. This is a paragraph. It has some words.
```

Output

```
Unique words and their frequencies:
a: 2
example: 1
has: 1
is: 2
...

Total number of unique words: 10

Enter a word: is
is appears 2 times

Unique words sorted alphabetically:
a
example
has
...
```

Input

Solution

Main: Part 1

std::getline is a function that reads a line of input from a stream. In the context of reading from the standard input (std::cin), std::getline will read an entire line of text, up to and including the newline character (\n). This is different from std::cin >>, which reads input up to the first whitespace character it encounters (like a space or a newline).

```
#include <iostream>
#include <string>
...

using ...

int main() {
    string paragraph;
    string word;
    set<string> words;
    map<string, int> freq_map;

    // Read a paragraph
    cout << "Enter a paragraph of text: ";
    getline (cin, paragraph);
```

Read an entire inpuy line

std::string s;
std::cin >> s;

If the user enters "Hello, World!", s will be "Hello," because std::cin >> stops reading at the first whitespace character.

On the other hand, if we write it like this
string s;
getline(cin, s);
If the user enters "Hello, World!", s will be "Hello, World!" because std::getline reads the entire line, up to and including the newline character.

Solution

Main: Part 2

Istringstrem, ostringstream, stringstream are like fstream but for in-memory string IO

For example, you can use std::istringstream to split a std::string into words like this:
```
std::string s = "Hello, World!";
std::istringstream iss(s);
std::string word;
while (iss >> word) {
    std::cout << word << '\n';
}
```

```
std::stringstream ss(paragraph);


while (ss >> word) {
  // Remove punctuation and convert to lowercase
  std::string clean_word;
  for (char c : word) {
    if (std::isalpha(c)) {
      clean_word += std::tolower(c);
    }
  }
  words.insert(clean_word);
  freq_map[clean_word]++;
}
```

Reading from a string stream instead than from IO (overloading)

Remove punctuation and convert to lowercase

Insert word in set (unique words)

Count word frequency

Solution

Display unique words and their frequencies

Main: Part 3

```
  std::cout << "\nUnique words and their frequencies:\n";
  for (const auto &pair : freq_map) {
    std::cout << pair.first << ": " << pair.second << std::endl;
  }
```

Display total number of unique words

```
  std::cout << "\nTotal number of unique words: «
    << words.size() << std::endl;
```

Search a word and display its frequency

```
  cout << "\nEnter a word: ";
  cin >> word;
  cout << word << "appears " <<
    freq_map[word] << " times.\n";

  std::cout << "\nUnique words sorted alphabetically:\n";
  for (const std::string &word : words)
    std::cout << word << std::endl;

  return 0;
}
```

Display unique words sorted alphabetically