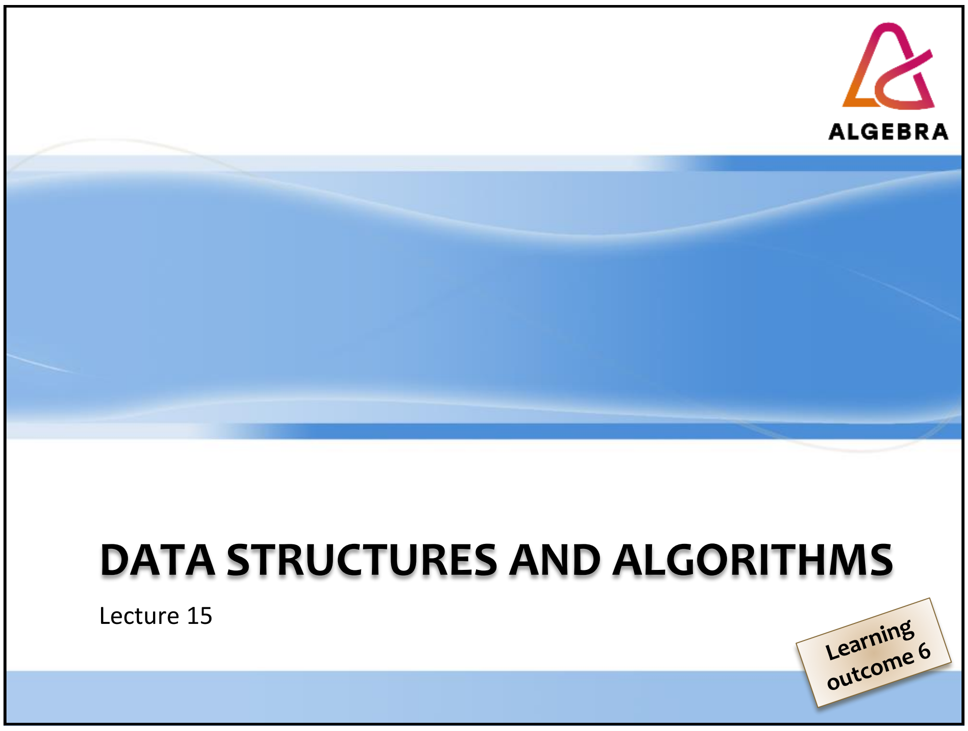
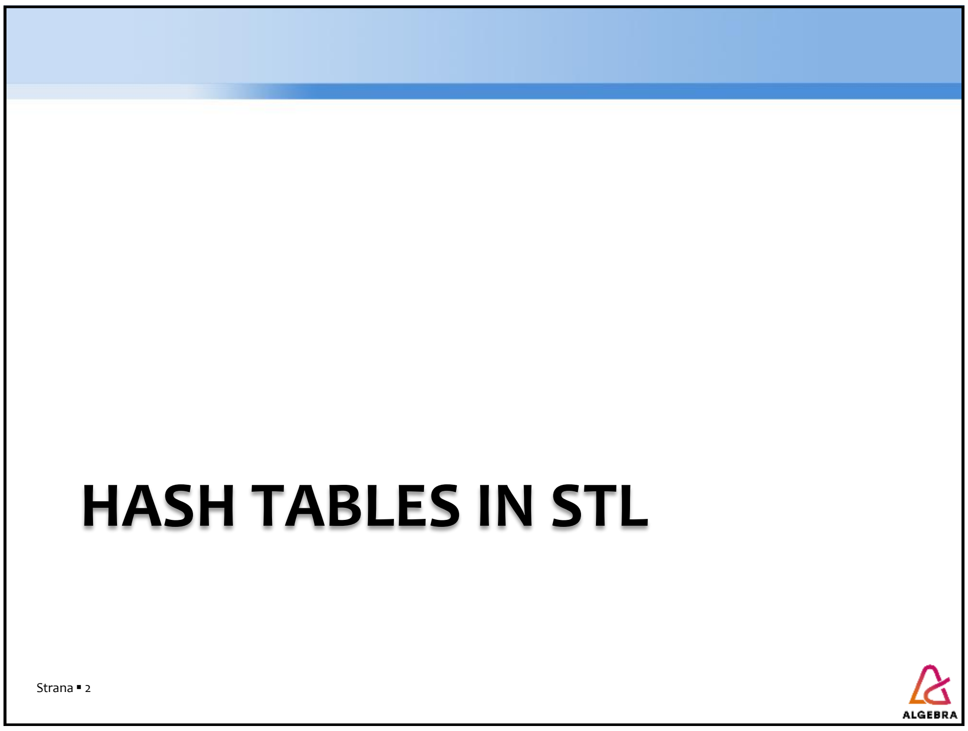


DATA STRUCTURES AND ALGORITHMS

Lecture 15

Learning outcome 6



HASH TABLES IN STL

Strana ▪ 2

Introduction

- The C ++ standard allows any implementation of hash tables, as long as the following applies:
  - The default maximum load factor must be 1.0
  - The hash table is guaranteed not to grow until the load factor exceeds the maximum load factor
- The result is that all implementations use chaining as a collision resolution method

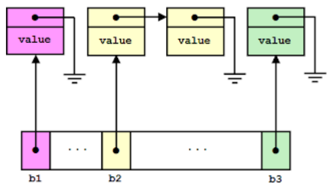


Image taken from: bannalia.blogspot.hr

Problem with classic chaining

- Classic chaining is an approach in which each slot has its own linked list
- This approach has a problem:
  - The standard requires that the hash table define the iterator
  - The standard requires that the iterator increment be $O(1)$
- The above requirement cannot be achieved with classical chaining
  - The solution is for all the elements from all the buckets to be interconnected

Improved chaining

- Implementation in Visual Studio uses one double-linked list for all buckets
  - Each element knows about the previous and next element
  - Each bucket stores two pointers: the first and the last element in the bucket

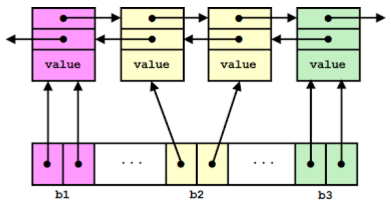


Image taken from: bannalia.blogspot.hr

Insertion

- Insertion has complexity of $O(1)$:
  1. Hash the key to find the bucket
     - $O(1)$
  2. If the key already exists in the bucket, give up
     - $O(k)$, where k is the number of elements in the bucket
     - A good hash function and a good maximum load factor guarantee that k will be around 1
  3. Add an item to the front of the list
     - $O(1)$
  4. Update pointers in the bucket
     - $O(1)$

Deletion

- Deletion has complexity of $O(1)$:
  1. Hash the key to find the bucket
     - $O(1)$
  2. Find the element in the bucket
     - $O(k)$
  3. Remove it from the list
     - $O(1)$
  4. Update pointers in the bucket if necessary
     - $O(1)$

Class unordered_map

- The main class that implements a hash table in STL is `unordered_map`
- It is defined like this:

```
template <class Key,
          class T,
          class Hash = hash<Key>,
          class Pred = equal_to<Key>,
          class Alloc = allocator< pair<const Key,T> >
          > class unordered_map;
```

- Required parameters are Key and T
  - Key is the data type of the key
  - T is the data type of the value

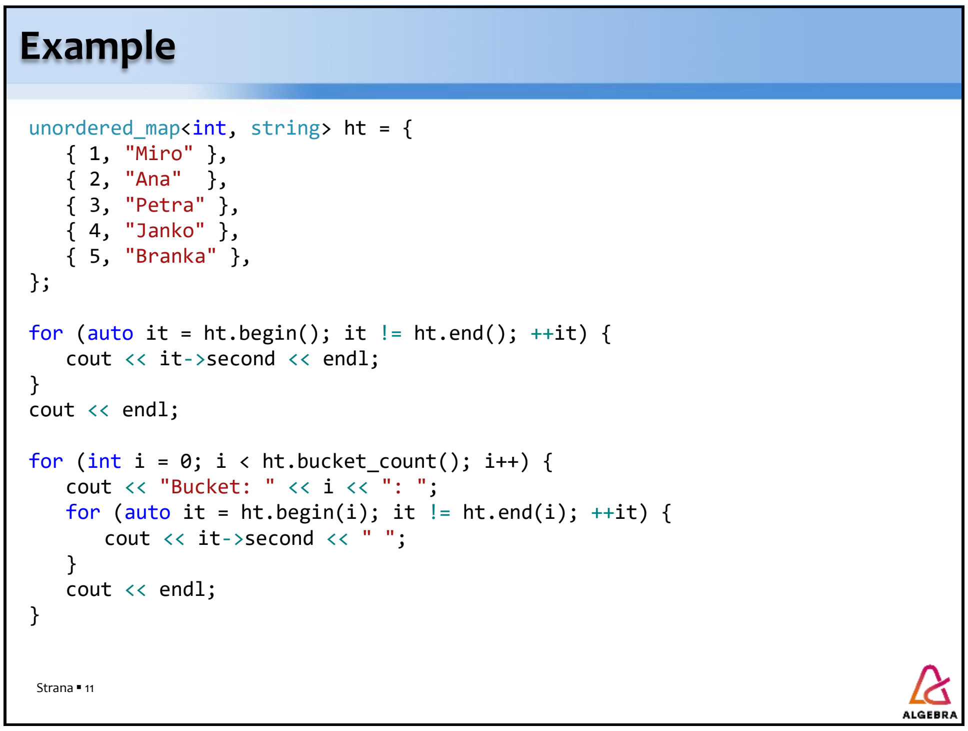
Creating hash table

- Basic ways are:
  - `unordered_map<int, string> one;`
    - Creates an empty hash table with key `int` and value `string`
  - `unordered_map<int, string> two(n);`
    - Create an empty hash table with key `int` and value `string`, but with minimum of *n* empty buckets

  -
    ```
    unordered_map<int, string> three = {
        { 1, "Miro" },
        { 2, "Ana"  }
    };
    ```
    - Creates a hash table with key `int` and value `string` and with two elements

Hash table iterators

- There are two possible procedures for iterating elements:
  - Iteration of elements in the bucket *b*
  - Iterate all elements in the hash table
- Methods we use in these procedures:
  - `ht.begin()` returns an iterator to a first element in hash table
  - `ht.begin(b)` returns an iterator to a first element in bucket b
    - Bucket *b* is an integer in range [0, `bucket_count()`)
  - `ht.end()` returns an iterator to a last element in hash table
  - `ht.end(b)` returns an iterator to a last element in bucket *b*
    - Bucket *b* is an integer in range [0, `bucket_count()`)



Example

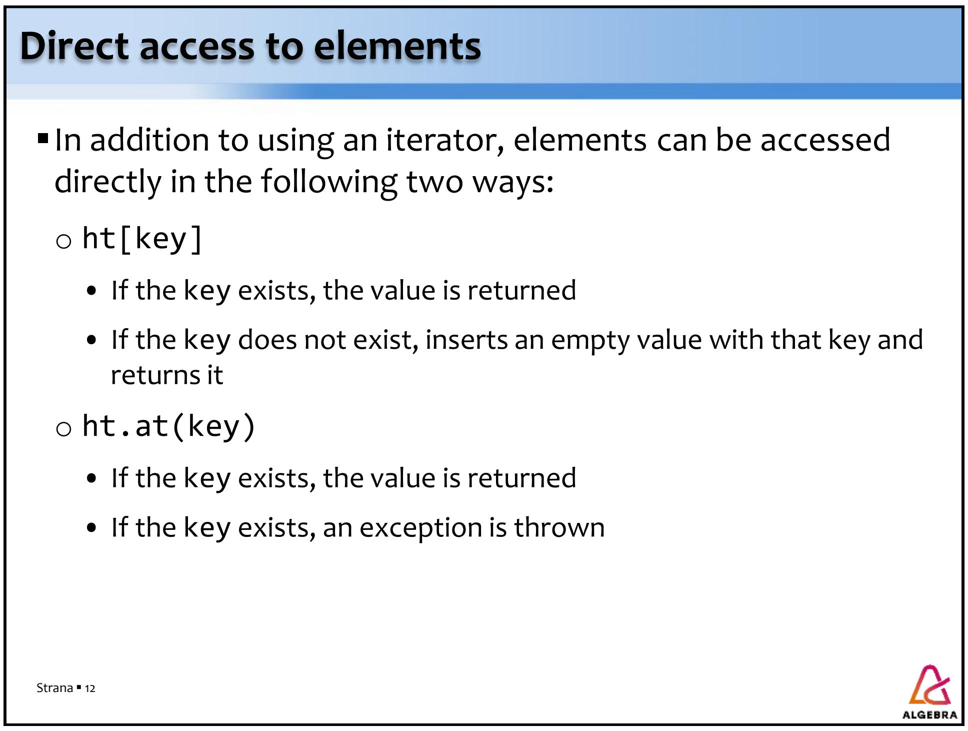
```
unordered_map<int, string> ht = {
   { 1, "Miro" },
   { 2, "Ana"  },
   { 3, "Petra" },
   { 4, "Janko" },
   { 5, "Branka" },
};

for (auto it = ht.begin(); it != ht.end(); ++it) {
   cout << it->second << endl;
}
cout << endl;

for (int i = 0; i < ht.bucket_count(); i++) {
   cout << "Bucket: " << i << ": ";
   for (auto it = ht.begin(i); it != ht.end(i); ++it) {
      cout << it->second << " ";
   }
   cout << endl;
}
```

Strana ▪ 11

ALGEBRA



Direct access to elements

- In addition to using an iterator, elements can be accessed directly in the following two ways:
  - `ht[key]`
    - If the key exists, the value is returned
    - If the key does not exist, inserts an empty value with that key and returns it
  - `ht.at(key)`
    - If the key exists, the value is returned
    - If the key exists, an exception is thrown

Strana ▪ 12

ALGEBRA

Example

```
unordered_map<int, string> ht({
   { 1, "Ana"   },
   { 2, "Juro"  },
   { 3, "Marko" }
});

cout << ht[1] << endl;
cout << ht[2] << endl;
cout << ht[3] << endl;
cout << ht[4] << endl;

cout << ht.size() << endl;
```

Insert into a hash table

- The most important ways to insert are as follows:
  - `ht.insert(pair<int, string>(99, "Ivana"));`
    - Inserts the pair and returns an object of type `pair<iterator, bool>`
      - `first` points to either a freshly inserted element or an equivalent element that already exists
      - `second` contains `true` (if the insertion was successful) or `false` (if the equivalent element already existed)
  - `ht.insert({ 99, "Ivana" });`
    - Same as before, but with prettier syntax
  - `ht.insert({{ 99, "Ivana" }, { 118, "Jurica" }});`
    - Inserts given pairs, but returns nothing

Example

```
unordered_map<int, string> ht;

auto it = ht.insert(pair<int, string>(99, "Ivana"));
cout << "Success: " << it.second << endl;

it = ht.insert(pair<int, string>(99, "Ivana"));
cout << "Success: " << it.second << endl;

cout << ht.size() << endl;

ht.insert({ { 99, "Marija" }, { 118, "Jurica" } });
cout << ht.size() << endl;

for (auto it = ht.begin(); it != ht.end(); ++it) {
    cout << it->first << " " << it->second << endl;
}
```

Delete from hash table

- Deletion can be done in the following ways:
  - `ht.erase(iterator)` deletes an element at a given position
    - Returns the iterator to the first next element after the deleted one
  - `ht.erase(key)` deletes the element with the given key
    - Returns the number of deleted items (0 or 1)
  - `ht.erase(begin, end)` deletes all elements in the range
    - Returns the iterator to the first following element after the last deleted one
  - `ht.clear()` deletes all elements

Example

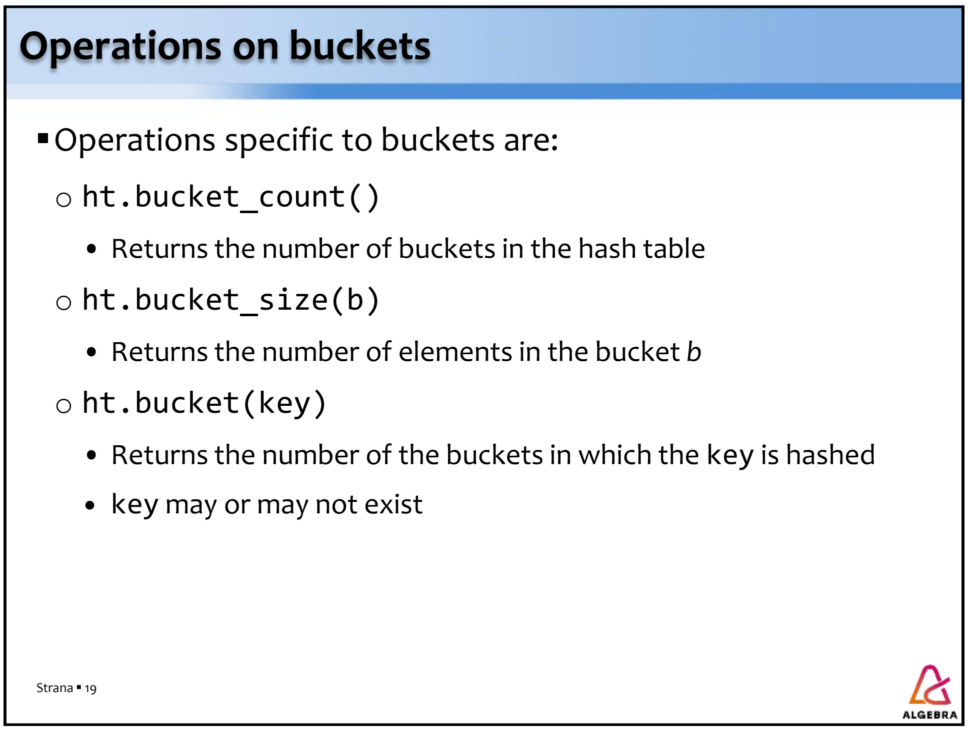
```
unordered_map<int, string> ht({
   { 1, "Ana" },
   { 2, "Juro" },
   { 3, "Marko" }
});

cout << ht.erase(2) << endl;
cout << ht.erase(2) << endl;

for (auto it = ht.begin(); it != ht.end(); ++it) {
   cout << it->first << " " << it->second << endl;
}
```

Other important methods

- `ht.find(key)`
  - Returns the iterator to the element with the key
  - Returns `ht.end()` if it doesn't exist
- `ht.size()`
  - Returns the number of elements in the hash table
- `ht.empty()`
  - Returns if the hash table is empty

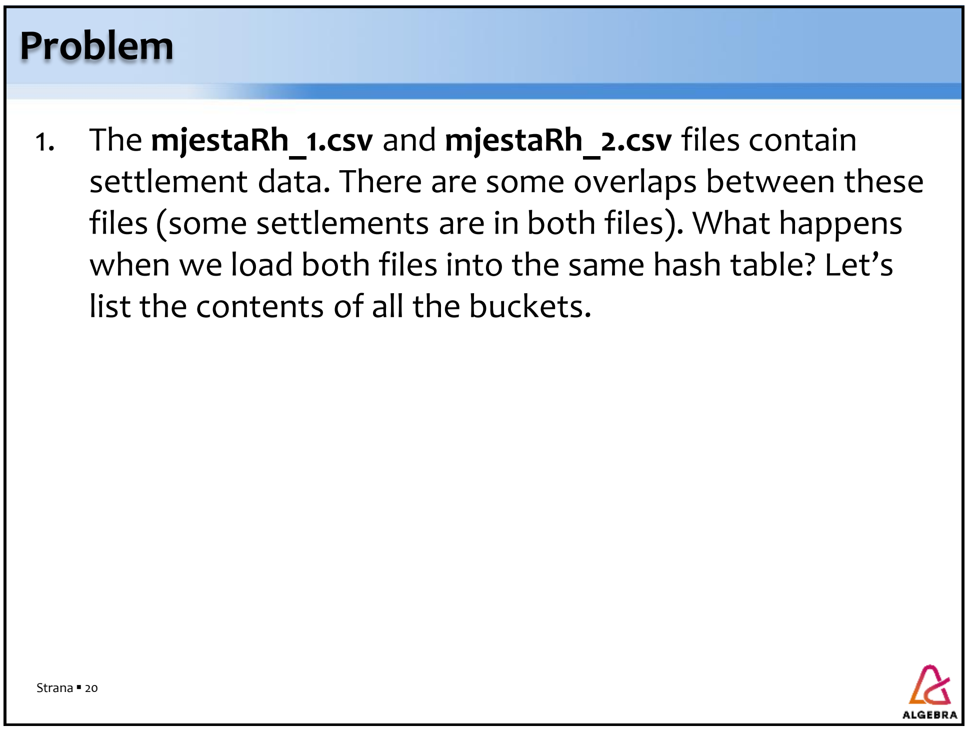


Operations on buckets

- Operations specific to buckets are:
  - `ht.bucket_count()`
    - Returns the number of buckets in the hash table
  - `ht.bucket_size(b)`
    - Returns the number of elements in the bucket *b*
  - `ht.bucket(key)`
    - Returns the number of the buckets in which the key is hashed
    - key may or may not exist

Strana ▪ 19

ALGEBRA



Problem

1. The **mjestaRh_1.csv** and **mjestaRh_2.csv** files contain settlement data. There are some overlaps between these files (some settlements are in both files). What happens when we load both files into the same hash table? Let's list the contents of all the buckets.

Strana ▪ 20

ALGEBRA

Solution

```
void load(ifstream& dat, unordered_map<string, string>& settlements)
{
   string temp;
   getline(dat, temp);

   string key;
   string val;
   while (true) {
        if (!getline(dat, key, ';')) {
            return;
        }

        getline(dat, val);

        settlements.insert({ key, val });
   }
}
```

Solution

```
unordered_map<string, string> settlements;

ifstream dat1("mjestaRh_1.csv");
ifstream dat2("mjestaRh_2.csv");
if (!dat1 || !dat2) {
    cout << "Error opening files" << endl;
    return 1;
}

load(dat1, settlements);
load(dat2, settlements);

for (unsigned i = 0; i < settlements.bucket_count(); i++) {
    cout << "Bucket " << i << ": ";
    for (auto it = settlements.begin(i); it != settlements.end(i);
++it) {
        cout << it->first << "-" << it->second << " ";
    }
    cout << endl;
}
```

Hashing rules (1/2)

- `load_factor()`
  - Returns the current load factor
  - The load factor is the ratio of the number of elements to the number of buckets
- `max_load_factor()`
  - Returns the maximum load factor (default 1.0)
- When the current factor exceeds the maximum factor, a rehash operation occurs:
  - The number of buckets in the hash table is growing
  - The hash function is changed to take into account the new number of buckets
  - The existing elements are rearranged in buckets

Example

```
unordered_map<int, string> ht({
    { 1, "Ana" },
    { 2, "Juro" },
    { 3, "Marko" }
});

for (int i = 1; i <= 100; i++) {
    ht.insert({ i, "dummy" });
    cout << "n=" << ht.size();
    cout << ", m=" << ht.bucket_count();
    cout << ", a = " << ht.load_factor();
    cout << "/" << ht.max_load_factor() << endl;
}
```

Hashing rules (2/2)

- `rehash(n)`
  - Sets the number of buckets to at least *n*:
    - If *n* is greater than the current number of buckets, rehash follows
    - If *n* is smaller, it probably won't do anything

Example

```
void print(unordered_map<int, string>& ht) {
    for (unsigned i = 0; i < ht.bucket_count(); i++) {
        cout << "Bucket " << i << ": ";
        for (auto it = ht.begin(i); it != ht.end(i); ++it) {
            cout << it->first << "-" << it->second << " ";
        }
        cout << endl;
    }
    cout << "---" << endl;
}
int main() {
    unordered_map<int, string> ht({{1,"Ana"},{2,"Juro"},{3,"Iva"}});
    print(ht);
    ht.rehash(8);
    print(ht);
    ht.rehash(5);
    print(ht);
    ht.rehash(100);
    print(ht);
    return 0;
}
```

HASH TABLE VARIATIONS

Introduction

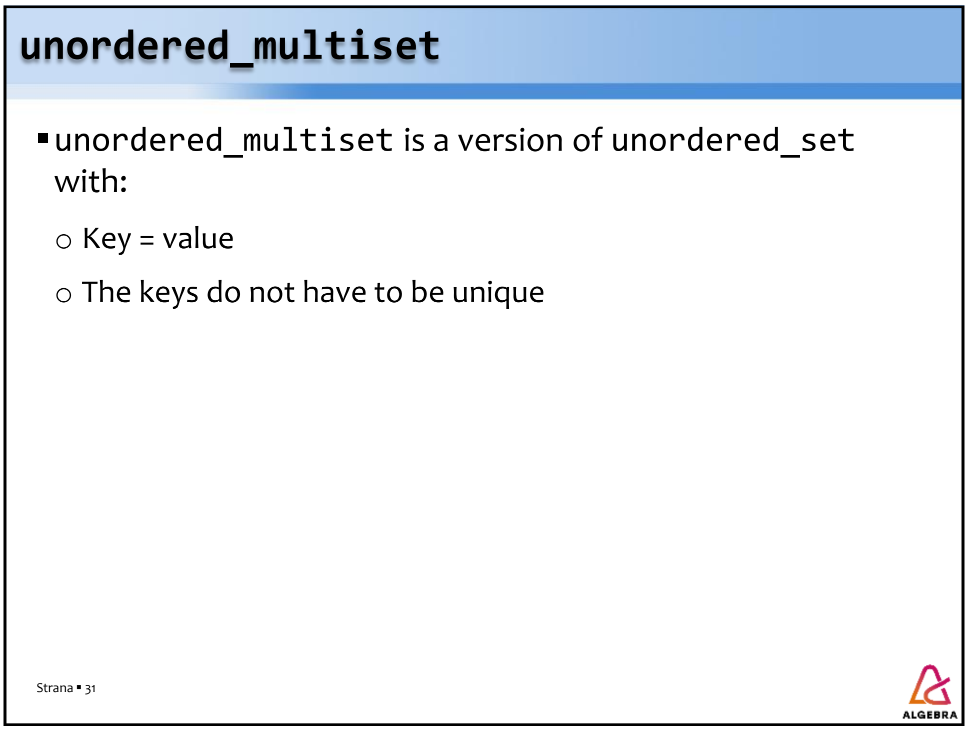
- Hash tables are implemented in the following STL classes:
  - `unordered_map`
  - `unordered_multimap`
  - `unordered_set`
  - `unordered_multiset`

unordered_multimap

- unordered_multimap is a version of unordered_map where the keys do not have to be unique
- The usage interface is similar, with the main difference:
  - No operator[]
  - No method at()

unordered_set

- unordered_set is a version of unordered_map with:
  - Key = value
  - The keys must be unique



`unordered_multiset`

- `unordered_multiset` is a version of `unordered_set` with:
  - Key = value
  - The keys do not have to be unique