CS32 Intro to Computer Science II

Baoxiong Jia & Muthu Palaniappan, DIS 1C Week 9
UCLA Spring 2021

About Us

- TA: Baoxiong Jia
  - Email: baoxiongjia@cs.ucla.edu
  - Office Hours: Tuesday 8:30-10:30am
  - Thursday 8:30-10:30am
  - Discussion 1C: Friday 12:00-13:50pm

- LA: Muthu Palaniappan
  - Email: muthupal@g.ucla.edu
  - Office Hours: Monday 10:30-11:30am
  - Wednesday 10:30-11:30am

Outline

- Hash tables

Hash tables

- Hashing
  - Distribute the entries (key and value pairs) across an array of buckets by using a hash function
- Hash function
  - A function carefully chosen to provide a uniform distribution of hash values to generate keys
  - The key is to avoid collisions of keys generated

Hashing

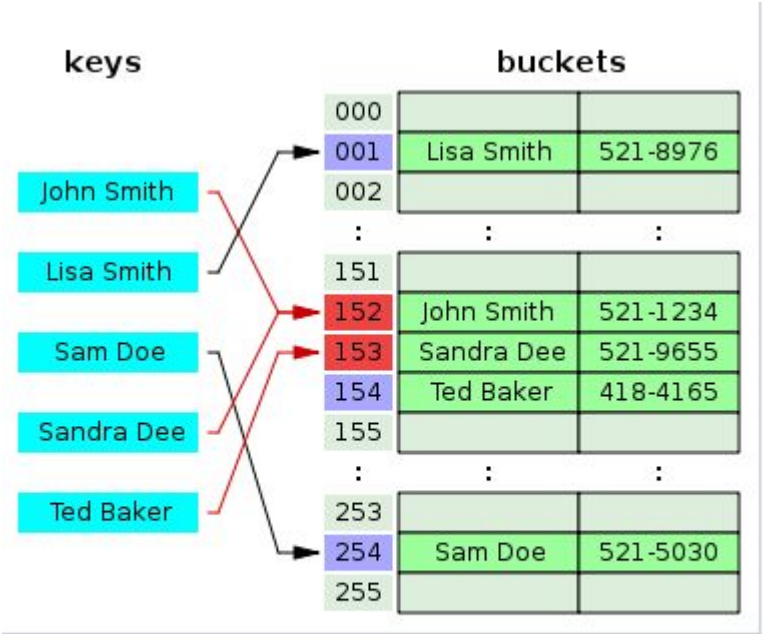
- Collision could occur even when the number of buckets is significantly larger than the number of entries
  - 10000 buckets to hash a dataset of strings, e.g. {Amy, Brendon, Cindy, David, ...}
  - A hash function f(x) based on capital letters where f('Amy') = 'a', f('Annie') = 'a', ...
- Load factor

$$\text{load factor} = \frac{n}{k},$$

where

- n is the number of entries occupied in the hash table.
- k is the number of buckets.

Resolving collision



Hash collision resolved by open addressing with linear probing (interval=1). Note that "Ted Baker" has a unique hash, but nevertheless collided with "Sandra Dee", that had previously collided with "John Smith".

To resolve collision, we typically use:

For a closed hash table

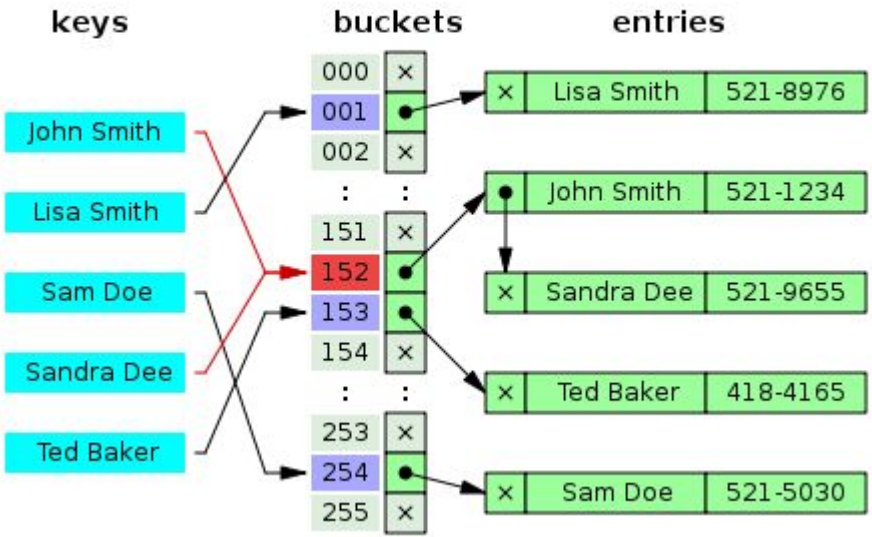
- Open addressing
  - Linear probing

Our closed hash table + linear probing works just fine, but it still has a few problems:

It's difficult to delete items

It has a cap on the number of items it can hold... That's a bummer.

Resolving collision



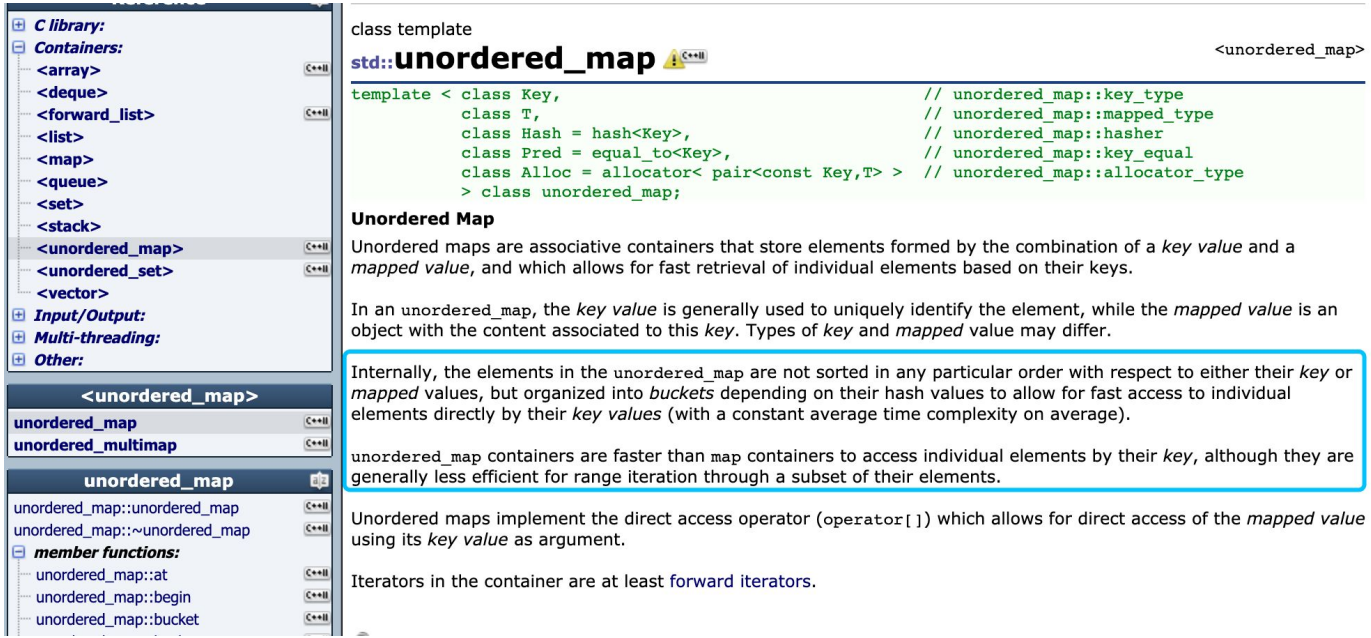
To resolve collision, we typically use:

For a open hash table

- Separate chaining
  - Linked list
  - Binary search trees

Hash table in C++

https://repl.it/@jiajerry/HashTable#main.cpp



- C library:
- Containers:
  - <array>
  - <deque>
  - <forward_list>
  - <list>
  - <map>
  - <queue>
  - <set>
  - <stack>
  - <unordered_map>
  - <unordered_set>
  - <vector>
- Input/Output:
- Multi-threading:
- Other:

<unordered_map>

- unordered_map
- unordered_multimap

unordered_map

- unordered_map::unordered_map
- unordered_map::~unordered_map
- member functions:
  - unordered_map::at
  - unordered_map::begin
  - unordered_map::bucket

class template

std::**unordered_map** <unordered_map>

```
template < class Key,                                    // unordered_map::key_type
           class T,                                      // unordered_map::mapped_type
           class Hash = hash<Key>,                       // unordered_map::hasher
           class Pred = equal_to<Key>,                   // unordered_map::key_equal
           class Alloc = allocator< pair<const Key,T> >  // unordered_map::allocator_type
           > class unordered_map;
```

Unordered Map

Unordered maps are associative containers that store elements formed by the combination of a *key value* and a *mapped value*, and which allows for fast retrieval of individual elements based on their keys.

In an `unordered_map`, the *key value* is generally used to uniquely identify the element, while the *mapped value* is an object with the content associated to this *key*. Types of *key* and *mapped* value may differ.

Internally, the elements in the `unordered_map` are not sorted in any particular order with respect to either their *key* or *mapped* values, but organized into *buckets* depending on their hash values to allow for fast access to individual elements directly by their *key values* (with a constant average time complexity on average).

`unordered_map` containers are faster than `map` containers to access individual elements by their *key*, although they are generally less efficient for range iteration through a subset of their elements.

Unordered maps implement the direct access operator (`operator[]`) which allows for direct access of the *mapped value* using its *key value* as argument.

Iterators in the container are at least forward iterators.