CS100
Introduction to Programming

Recitation 7

llk89

Problem 1
std::chrono

Get the time and measure times with std::chrono

- Header:

```
#include <chrono>
```

- Use

```
std::chrono::high_resolution_clock
```

- Figure out how to
  - Get time-points
  - Extract durations
  - Express durations in microseconds, and cast them to an int

Get the time and measure times with std::chrono

- Test the time/duration functionality
- You may use a dummy function to consume time

```
// long operation to time
int dummyFunction(int n) {
  if (n < 2) {
    return n;
  } else {
    return fib(n-1) + fib(n-2);
  }
}
```

Problem 2

Implement a Timer class

Implement a Timer class

- Lap timer to accumulate iteration times!
- Interface:

```
class Timer {
public:
  Timer( bool start = false );
  virtual ~Timer();

  void start();
  void stop( bool restart = false );
  void stop( size_t iterations,
      bool restart = false );
  void reset();

  double averageTime();
  std::list<double>::iterator begin();
  std::list<double>::iterator end();
  …
};
```

Implement a Timer class

- What private variable to choose?
- What type should the lap time container be?

- begin():
  - Iterator to first element in lap-time container
- end():
  - Iterator to last element in lap-time container

Test the Timer class

- Use **`dummyFunction()`**

Problem 3
Filling lists and vectors

Define a large object

```
class LargeObject {
public:
  LargeObject();
  virtual ~LargeObject();
private:
  int m_data[1000000];
};

LargeObject::LargeObject() {};
LargeObject::~LargeObject() {};
```

Measure filling times

- Measure times for putting elements into a list and a vector!
  - Fill 500 **`LargeObject`**s into a list and a vector
  - Measure the time of each iteration with a **`Timer`**

- Print all times into the console
- What can you observe?

Constant time & linear time

- For a container…
  - Constant time: an ops takes the same amount of time no matter what
    - e.g. array get/set. linked list push_back/pop
  - Linear time: an ops takes the time proportional to number of elements in the container
    - e.g. linked list random access

Part 4
std::unordered_map

Associative container

- Stores elements as as key-value pairs
- Similar to std::map, every element needs to have unique key

Search, insertion, and removal all have approximately **constant-time complexity**!
(remember that std::map has complexity of O(log(n))

```
name          employee
string        Employee

map<string, Employee *> employees;
```

Associative container

- Stores elements as as key-value pairs
- Similar to std::map, every element needs to have unique key

Search, insertion, and removal all have approximately **constant-time complexity**!
(remember that std::map has complexity of O(n)

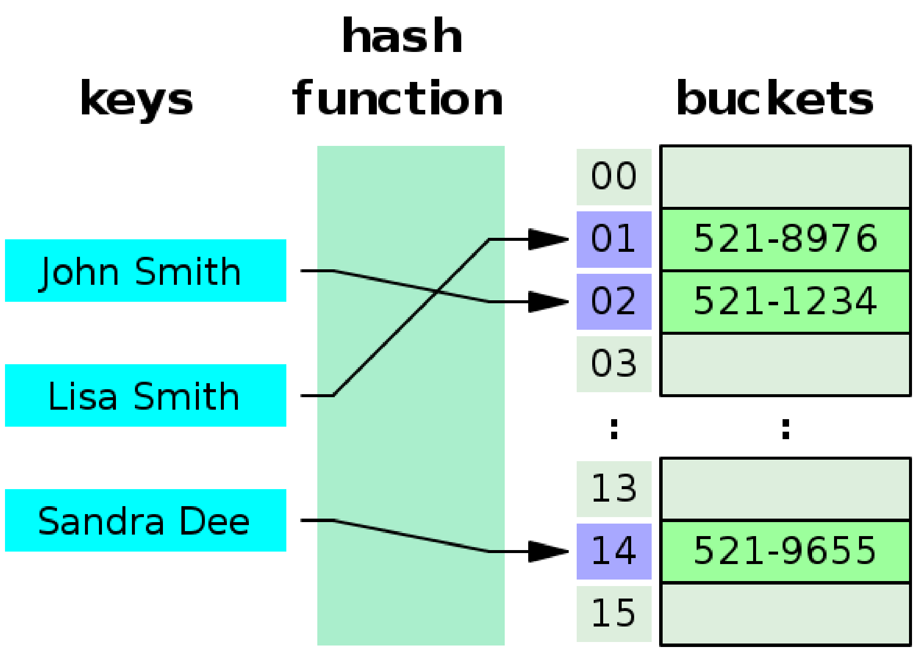
```
name         employee
string       Employee

unordered_map<string, Employee *> employees;
```

How does it work?

- Internally, the elements are organized into buckets
- Which bucket an element is placed into depends on its key
  - From an arbitrary key-type, we derive a bucket-index
  - The bucket-index is called a hash
  - A hash-function is a function that maps an arbitrary input type to a defined type with defined range
- This allows fast access to individual elements, since once a hash is computed, it refers to the exact bucket the element is placed into

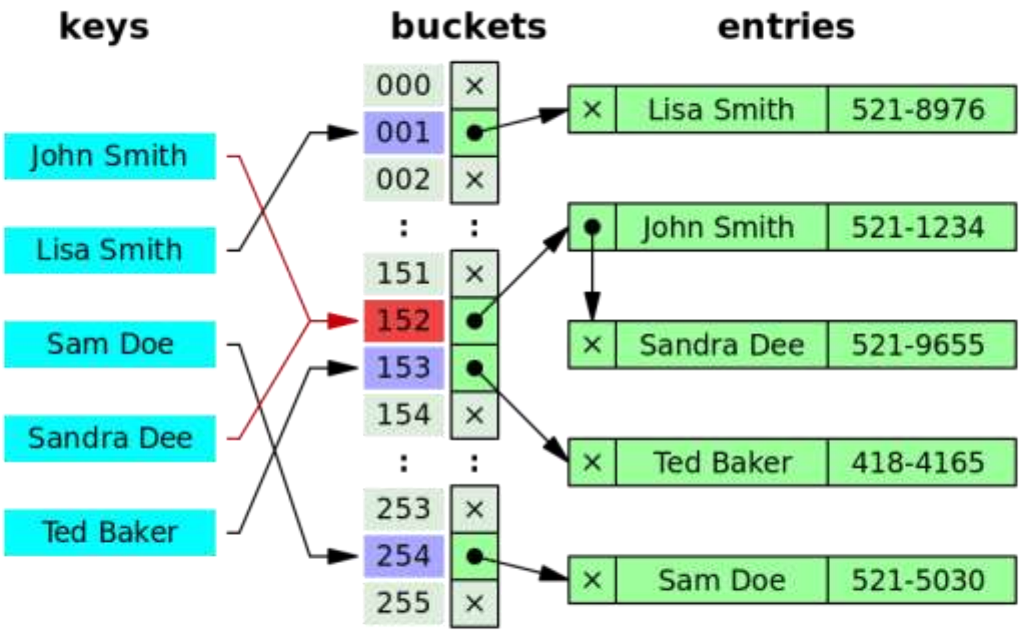
How does it work?

- Hash table
- https://en.wikipedia.org/wiki/Hash_table
- Ideally: Every key leads to an individual bucket



How does it work?

- Hash table
- https://en.wikipedia.org/wiki/Hash_table
- In practice: Limited number of bucket & extra collision resolving (e.g. chaining)



How does it work?

- Properties:

Algorithm	Average	Worst case
Space	$O(n)$[1]	$O(n)$
Search	$O(1)$	$O(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

- How does the space property come across?
  - Only a table with (initially NULL) bucket pointers is installed
  - This table has a constant size equal to the range of the hash
  - The actual buckets are growing linearly with the actual number of elements in the list

Exercise

- Create a main function in which you
  - Fill a std::map with key value pairs of the form

    ```
    std::map<int,std::string>
    ```

  - Notes:
    - The int can simply increase linearly for every element
    - The string can be same, dummy string everytime
- **Task 1:**
  - Add 10000 elements, then measure the time of adding 100 more elements

Exercise

- **Task 2:**
  - Do the same for an unordered map
  - What do you observe?

- **Task 3:**
  - Now increase the initial size of the container by a factor of 10.
  - What do you observe?

Part 5
LinkedMap

LinkedMap

- Behavior:
  - A generic container just like map.
  - Ordered based on insertion order instead of comparison
  - Approximate constant time insertion, deletion, look up and iteration
- Implementation:
  - Composition of a std::list and a std::unordered_map
  - Define an appropriate node structure

Implement your own template map container

- Requirements:
  - Reasonable iterators
  - Reasonable constructor and destructors
  - Able to be printed to a stream with operator <<
  - Required interface functions:

```
iterator begin();
iterator end();
bool empty();
int size();
void erase(const K& k);
void set(const K& k, const V& v);
V& get(const K& k);
void clear();
```

Test LinkedMap with this

- Instantiate new map and fill it with elements

```
#include "LinkedMap.hpp”
#include <iostream>
#include <algorithm>
#include <vector>

int main() {
  //instantiate one of my new map object and set two entry
  LinkedMap<int, double> linkedMap;
  linkedMap.set(1, 1.0);
  linkedMap.set(2, 4.0);
  linkedMap.set(0, 0.0);
  …
}
```

Test LinkedMap with this

- Print them out using the operator you defined
- Verify they are in the correct order

```
#include "LinkedMap.hpp"
#include <iostream>
#include <algorithm>
#include <vector>

int main() {
  …
  // print them out
  std::cout << linkedMap << std::endl;
  …
}
```