Hash Table

CP::unordered_map

A map with faster find but has no ordering

Overview

- Binary Search Tree (and AVL) utilizes the fact that the data is sorted by the structure. This helps finding where we keep the data by narrowing down the searching area of the structure.
  - However, we still have to check, for about O(log n), position
  - We also get sorting of the data as a by-product. Traversing the tree (using in-order traversal) access the data in sorted order.
- Hash table aims to reduces this checking further down but give up the ordering of the data (and some more overhead cost in iteration of all data)
- Hash Table useห a Hash Function (h) to map a single data value to a unique position. An array T is created to store a data x by saying that the data x must be at position T[h(x)] only.
  - The function h depends on the type and range of value of the data to be stored
  - The function h maps any possible value of data to position in T. Ideally, h should be 1-to-1 and onto and the co-domain of h should be {0,...,n-1}. This is called a perfect hash function.
  - We will see that a perfect hash is hardly possible in practice, and we have to fix it somehow.

Perfect Hash Example: The Structure

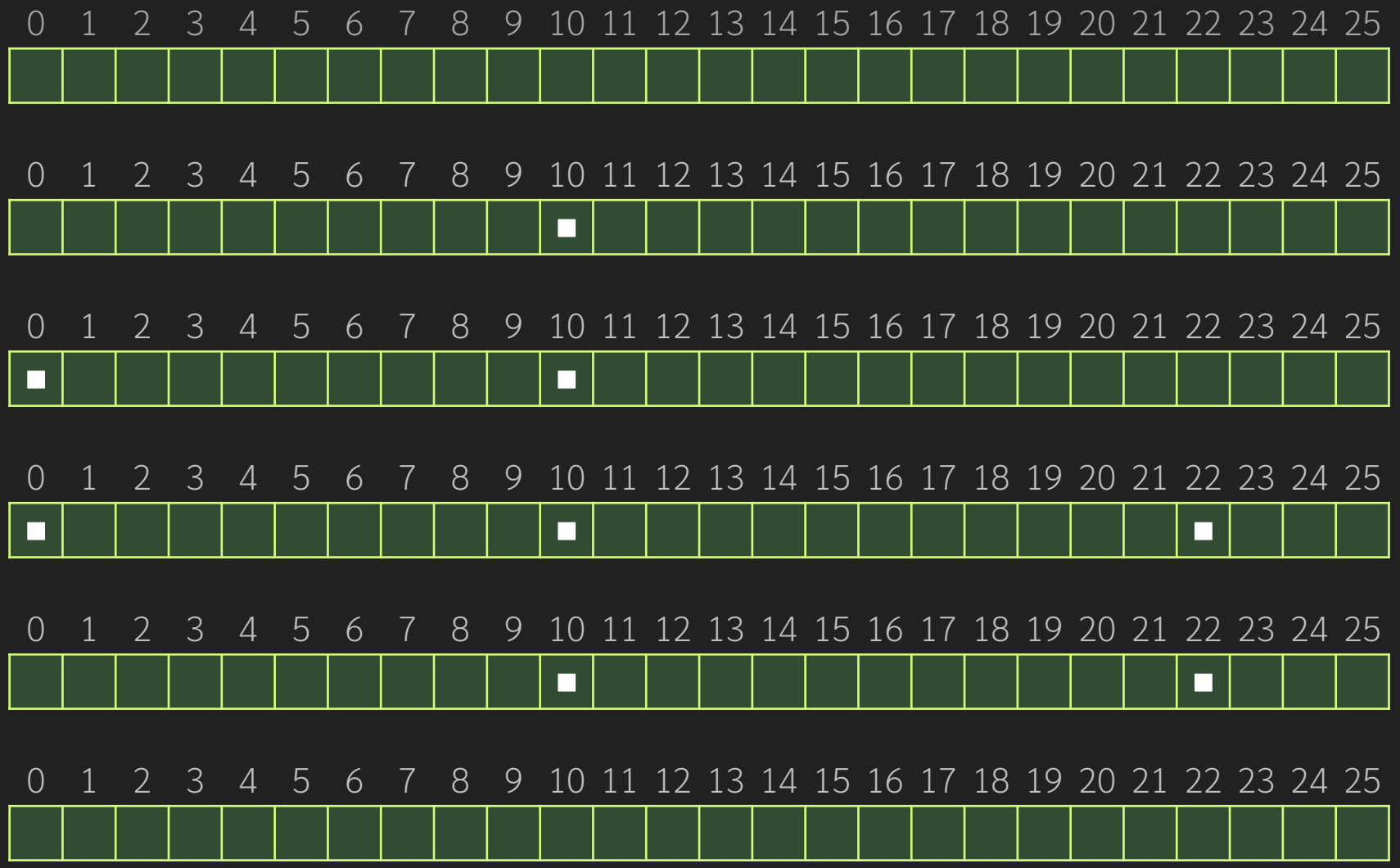
- Assume that the data we wish to store are only lower English character ‘a’ to ‘z’.
- We define a hash function h(c) that maps ‘a’ to 0, ‘b’ to 1, …, and ‘z’ to 25. The range of the function is 0 to 25 and we can clearly see that h(c) is 1-to-1 and onto
- Since h(c) has 26 possible value (0 to 25), the table T must be created with 26 slots (T[0] to T[25])
  - For now, consider T[i] to be a Boolean where T[h(x)] = true means we have x in our structure.

x	H(x)
a	0
b	1
c	2
d	3
e	4
f	5
g	6
h	7
i	8
j	9
k	10
l	11
m	12
n	13
o	14
p	15
q	16
r	17
s	18
t	19
u	20
v	21
w	22
x	23
y	24
z	25

The hash function

The Table T

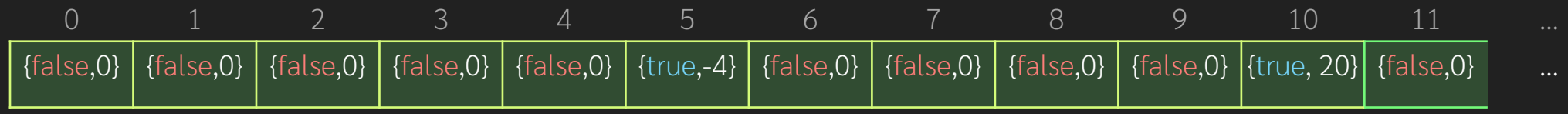
Perfect Hash Example: Operation



x	H(x)
a	0
b	1
c	2
d	3
e	4
f	5
g	6
h	7
i	8
j	9
k	10
l	11
m	12
n	13
o	14
p	15
q	16
r	17
s	18
t	19
u	20
v	21
w	22
x	23
y	24
z	25

Using Hash Table as a std::map

- As a map, we store a pair of key and its associated data
- For simplicity in describing the hash, we will consider using a hash as a std::set where we only store whether we have the key in the data structure.
  - To use a map, we will store both the key and its associated value
  - In that case, T[x] (where x is of type T1) stores a pair <bool,T2> where bool indicate whether we have the key X in our data structure and T2 stored the associated value.
- For example, let's have an `unordered_map<char,int> a` and let use set `a['k'] = 20` and `a['e'] = -4`



Basic Analysis of perfect hash

- Adding, Deleting and Finding can be done in constant time (plus the time to run the hash function)
  - For the previous example, the hash function is
  - This is as fast as possible

```
int h(char c) {
  return c - 'a';
}
```

- Drawback?
  - The size of the table is always the number of possible value of data to be stored regardless of how many actual data are being stored, it is 26 in the previous example
  - What is the size of the table T when we wish to store any string of lower character of length 10?
  - In many case, the number of possible value of data is prohibitively large.
  - There are approx. 600k English words in the dictionary
  - Comparing to the number of any string of length 10

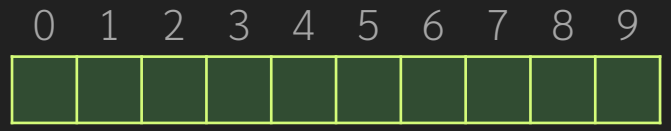
Perfect hash problem and solution

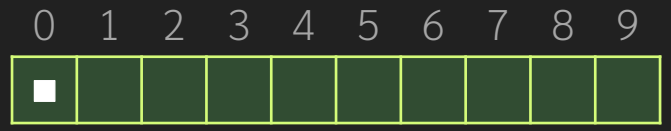
- Just don't use the perfect hash, h may not be 1-to-1 nor onto, to reduce the size of the co-domain of h, so that the size of T can be reduced
- Since h is now not 1-to-1, it is possible that two different keys x1 and x2 map to the same value under h, i.e., h(x1) == h(x2)
  - This is called a collision
- Example, consider a non-perfect hash function h(x) = `return (c - 'a') % 10;`

x	H(x)
a	0
b	1
c	2
d	3
e	4
f	5
g	6
h	7
i	8
j	9
k	0
l	1
m	2
n	3
o	4
p	5
q	6
r	7
s	8
t	9
u	0
v	1
w	2
x	3
y	4
z	5

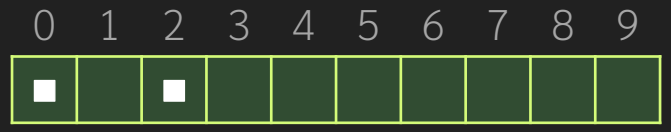
Collision Example

Since h(x) is 0..9, we only need T[0] .. T[9]



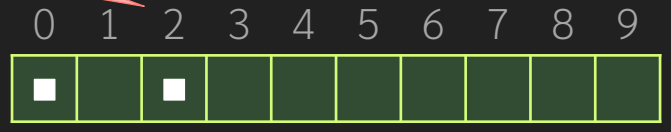


Add ‘k’

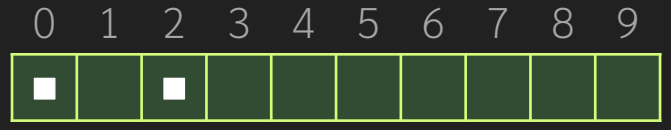


Add ‘w’

We don’t have ‘m’ but T[h(‘m’)] is marked TRUE



find ‘m’



add ‘a’

We can’t add ‘a’ because T[h(‘a’)] is marked TRUE by ‘k’

x	h(x)
a	0
b	1
c	2
d	3
e	4
f	5
g	6
h	7
i	8
j	9
k	0
l	1
m	2
n	3
o	4
p	5
q	6
r	7
s	8
t	9
u	0
v	1
w	2
x	3
y	4
z	5

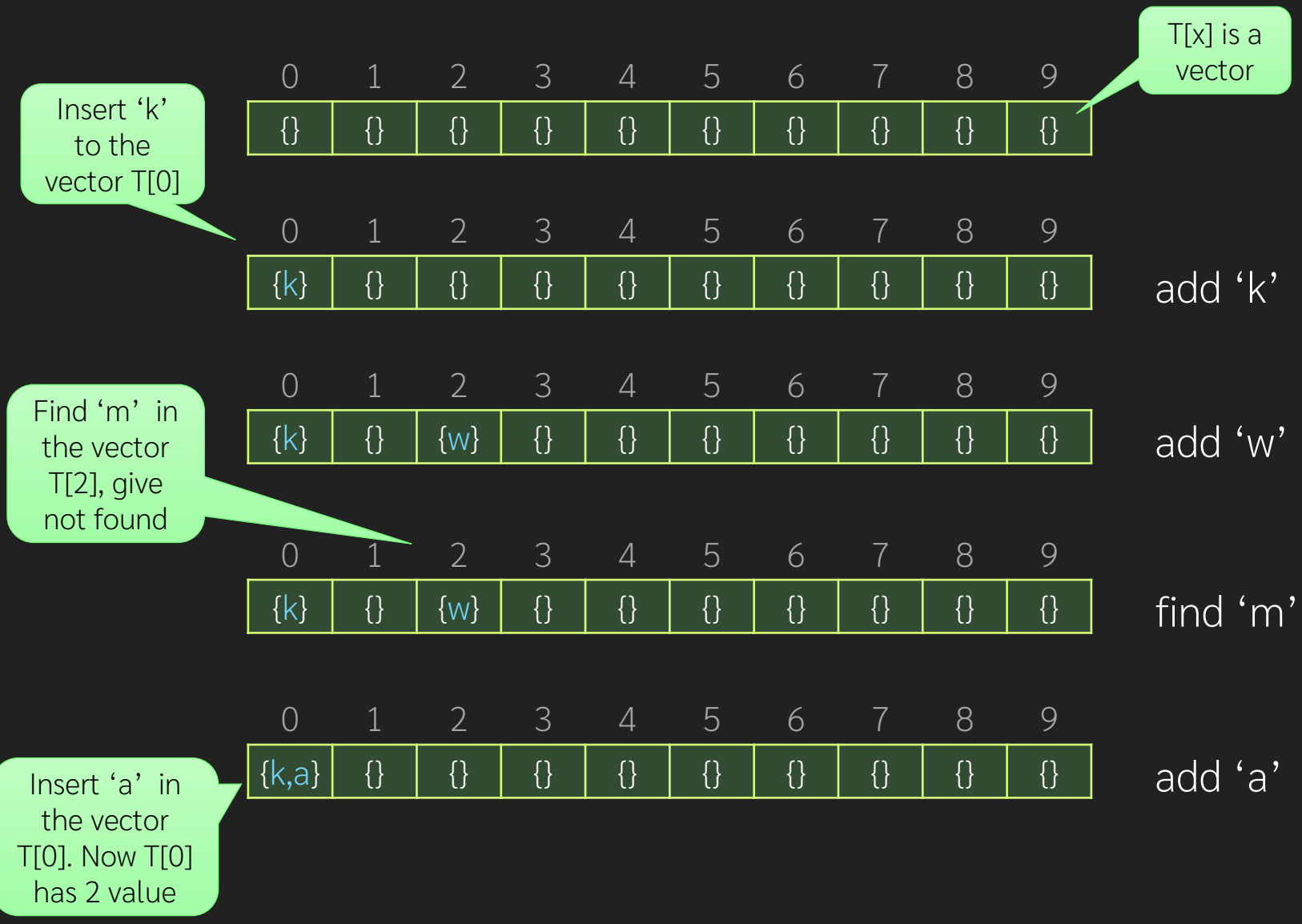
Outline

- How we fix collision problem?
  - Separate chaining technique
  - Open Addressing technique
- How we create a hash function in practice?
- Implementation

Separate Chaining

Collision Handling in Hash Table

Separate Chaining Overview



- Instead of having T[x] stores a true/false value to indicates whether we have x, we let T[x] stores another data structure that can hold multiple value of x
  - For example, T[x] can be a vector of type of x instead of a Boolean

Coding ver 0.1: overview

```
template <typename KeyT,
          typename HasherT = std::hash<KeyT>>
class hash_set_sp {
  protected:
    typedef std::vector<KeyT>      BucketT;

    std::vector<BucketT> mBuckets;
    size_t               mSize;
    HasherT              mHasher;

    size_t hash_to_bucket(const KeyT& key) {
      return mHasher(key) % mBuckets.size();
    }

  public:
    // default constructor
    hash_set_sp() :
      mBuckets( std::vector<BucketT>(97) ),
      mSize(0), mHasher(HasherT()) { }

    // basic functions
    bool empty() { return mSize == 0; }
    size_t size() { return mSize; }
    bool has(const KeyT &key) { … }
    bool insert(const KeyT &key) { … }
    bool erase(const KeyT &key) { … }
};
```

Need to mod by table's size

- Minimal version, no iterator
- The table T is `mBuckets`
  - Each slot is called a bucket and is a `std::vector<KeyT>`
- The size of the table is fixed arbitrarily at 97 (no special reason, just some random fixed number)
- C++ provides a kind of hash function as a class `std::hash` which can be used to convert a value of most basic types to a value in the range of `size_t`
- Our actual hash function is `hash_to_bucket` which convert `KeyT` to 0..96
- `empty` and `size` operate as usual

What is `std::hash`

- A class that converts basic types to a number
- Is used by default in C++ hash

```cpp
int main() {
  cout << "-- int --" << endl;
  std::hash<int> h1;
  for (int i = -5;i < 5;i++) {
    cout << i << ": " << h1(i) << endl;
  }

  cout << endl << "-- string --" << endl;
  std::hash<string> h2;
  cout << "a:       " << h2("a") << endl;
  cout << "aa:      " << h2("aa") << endl;
  cout << "aaa:     " << h2("aaa") << endl;
  cout << "somchai: " << h2("somchai") << endl;
  cout << "xyz:     " << h2("xyz") << endl;
  cout << "(empty): " << h2("") << endl;

  cout << endl << "-- float --" << endl;
  std::hash<float> h3;
  for (int i = -5;i < 5;i++) {
    cout << i << ": " << h3(-i) << endl;
  }
}
```

```
-- int --
-5: 18446744073709551611
-4: 18446744073709551612
-3: 18446744073709551613
-2: 18446744073709551614
-1: 18446744073709551615
0: 0
1: 1
2: 2
3: 3
4: 4

-- string --
a:       4993892634952068459
aa:      468926534229516570
aaa:     16343632196508755401
somchai: 17001615847932193163
xyz:     10512747046380093913
(empty): 6142509188972423790

-- float --
-5: 4236386057441319064
-4: 2393625642355595807
-3: 9509289860528194733
-2: 10014322907686782376
-1: 6045349164495482701
0: 0
1: 8126086649191387050
2: 11634374135363760669
3: 7062608420545188058
4: 10177811019412989047
```

Coding ver 0.1: basic operations

```
bool find(const KeyT &key) {
  size_t bucketIdx = hash_to_bucket(key);
  auto &b = mBuckets[bucketIdx];
  return std::find(b.begin(),b.end(),key) != b.end();
}

bool insert(const KeyT &key) {
  if (find(key)) {
    return false;
  } else {
    size_t bucketIdx = hash_to_bucket(key);
    mBuckets[bucketIdx].push_back(key);
    mSize++;
    return true;
  }
}

bool erase(const KeyT &key) {
  if (find(key)) {
    size_t bucketIdx = hash_to_bucket(key);
    auto &b = mBuckets[bucketIdx];
    auto it = std::find(b.begin(),b.end(),key);
    b.erase(it);
    mSize--;
    return true;
  } else {
     return false;
  }
}
```

- `find, insert, erase` are delegates to the vector
  - Notice that insert and erase use find
- `hash_to_bucket` tells us which vector we should operate on

Basic Analysis

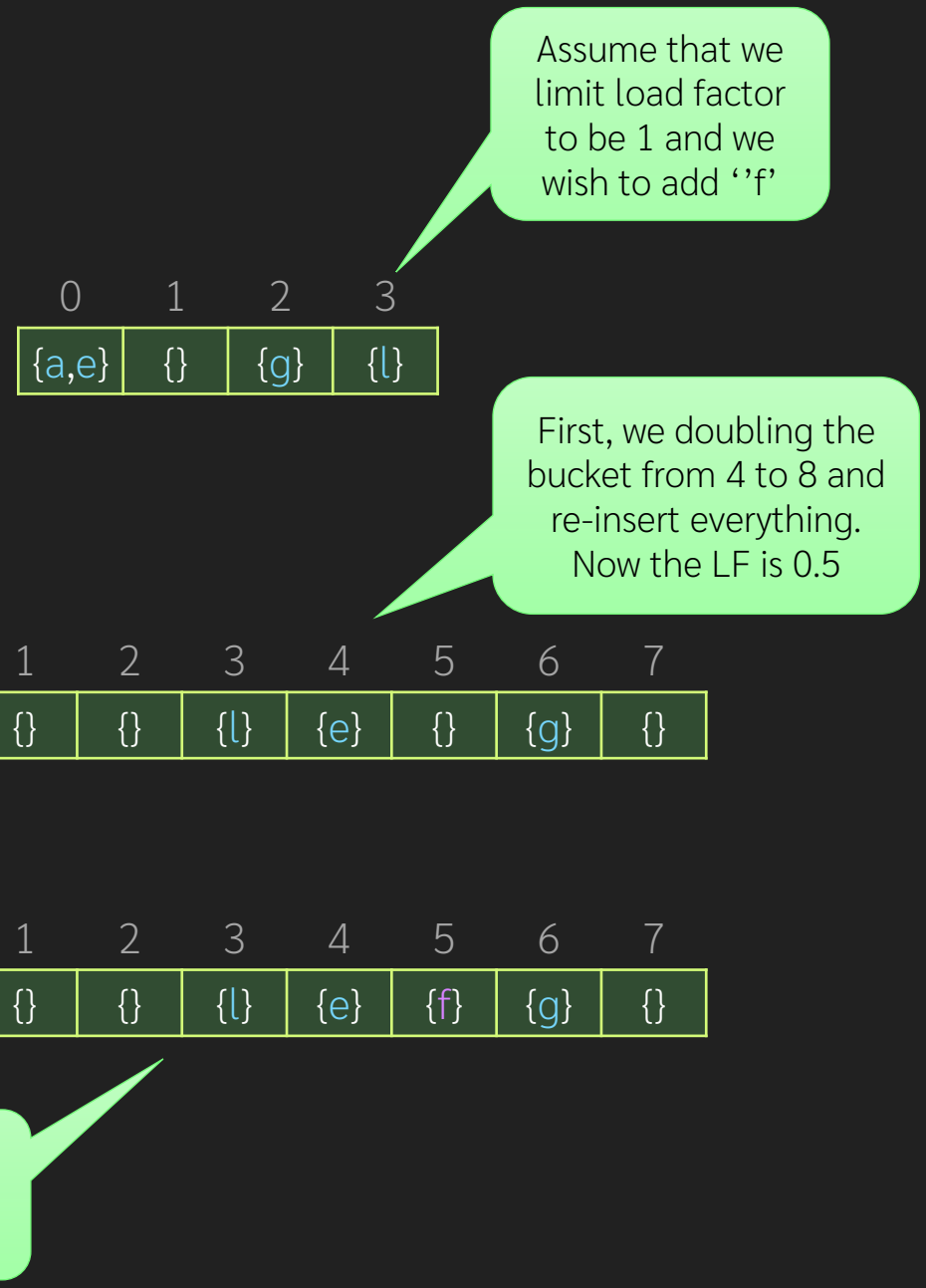
- Theoretically, since our hash function might map everything to the same bucket, it is possible that every data is stored in only one single bucket.
  - Find will takes O(N), which also make insert and erase also O(N)
  - This is the same as storing every data in just a simple vector
- In practice, assume that the data are distributed into each buckets equally.
  - Find will takes O(N / 97), which is the same as O(N) but 97 times faster in practice
  - If we instead use 1,000,000 buckets instead of 97 and assume that we never insert more than 1,000,000 data into our hash table, the practical run time is O(1)
- This is very crude analysis, but it gives main points of a separate chaining hash table
  - Speed depends on the number of data / the number of buckets and uniform distribution of the hash function
  - Memory Usage depends on the number of buckets

Load Factor and Trade Off between Time and Space

- We define Load Factor (λ) as the number of data / the number of buckets
  - A hash table with 3 data and 5 buckets has a load factor of 3/5 = 0.6
- Load factor is the expected time it takes to find a key
- Small load factor means faster operation
  - To have small load factor we need large number of buckets which is wasteful of space when we store small number of data
- Large load factor means smaller memory usage
  - But it makes operations use longer time

Adaptive Bucket Size with Rehash

- We can maintain small load factor with small memory usage by using rehash which is similar to expansion of `mData` in vector
- Start with small number of buckets
  - So that it does not take too much space when we have small number of data
- When load factor goes above limiting value, we expand, usually by doubling the size of the `mBuckets`, and re-insert everything
  - This effectively reduce load factor by half
  - We have to re-insert because `hash_to_bucket` use the size of the table
- This make some insertions take very long time, but it happens infrequently
  - Using the same analysis as the expansion of vector



Code ver 0.2: load factor and rehash

```
float load_factor() { return mSize/(0.0+mBuckets.size()); }

float max_load_factor() { return mMaxLoadFactor; }

// setting the max load factor
void max_load_factor(float z) {
  mMaxLoadFactor = z;
  rehash(mBuckets.size());
}

// try to resize the mBuckets to n
void rehash(size_t n) {
  if (load_factor() < max_load_factor()
      && n <= mBuckets.size())
    return;
  n = std::max(n, mBuckets.size()*2);

  std::vector<KeyT> tmp;
  for (auto& b : mBuckets)
    for (auto &k : b)
      tmp.push_back(k);
  mBuckets = std::vector<KeyT>(n);
  for (auto& k : tmp) {
    insert(k);
  }
}
```

```
bool insert(const KeyT &key) {
  if (find(key)) {
    return false;
  } else {
    if (load_factor() > max_load_factor())
      rehash(2*bucket_count());
    size_t bucketIdx = hash_to_bucket(key);
    mBuckets[bucketIdx].push_back(key);
    mSize++;
    return true;
  }
}
```

Check load factor and rehash before insert

- More public function
  - Some for get/set load factor and its limit
- `mMaxLoadFactor` is a protected float
- `rehash` works like `resize` in vector
  - It expands at least a double in size
  - This take O(n) to read and another O(n) * insertion
- `insert` has to be modified as well

Analysis

- `insert` and `erase` depend on `find`
- `find` is O(n)
  - However, if we assume that the hash function distributes the data into each bucket uniformly, we expect that find is O(λ), or the load factor
  - If we maintain λ to be 1, we expect that, on average, each operation to takes a constant time, or O(1)
- Obviously, we have to do re-hashing. By doubling the size every time we rehash, we can use the same argument as the expand of vector to show that the cost of rehash is constant over a long run

What’s remain?

- Iterator
- Adjust the hash to actual `unordered_map` where both the key and associated data is stored
- We will skip all of these for now, moving on to another collision handling strategy first.

Open Addressing

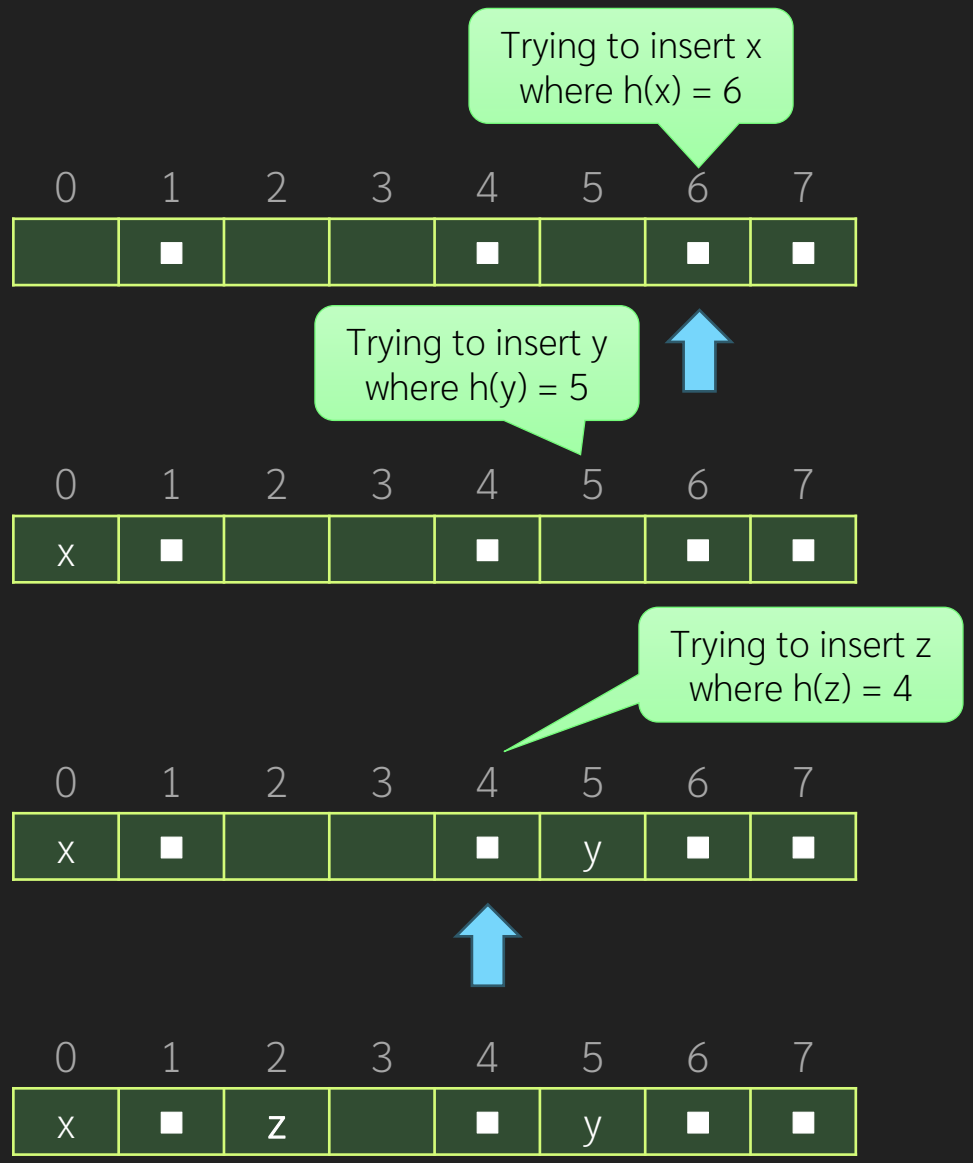
Another Collision Handling in Hash Table

X can be in different bucket other than h(x)

- Instead of allowing multiple data to be stored in the same location, open addressing only permit one single data to be stored in each position in the table
  - This will always make the load factor not be more than 1.0
- But it will give precise alternate positions to be used when collision occur
  - There are several strategy for calculating alternate positions: linear probing, quadratic probing and double hashing

Example: Linear Probing

- If T[h(x)] is occupied (because some other data y where h(y) == h(x) is already inserted into T[h(x)]
  - Check T[h(x) + 1] instead
  - If it is also occupied, check T[h(x) + 2] instead
  - Repeat (circularly) until we found empty bucket
- Obviously, if load factor less than 1.0, we can always find an empty bucket
  - Must rehash when load factor is 1.0
    - In practice, we rehash way sooner than that, e.g., rehash when LF is 0.5



Probing Function

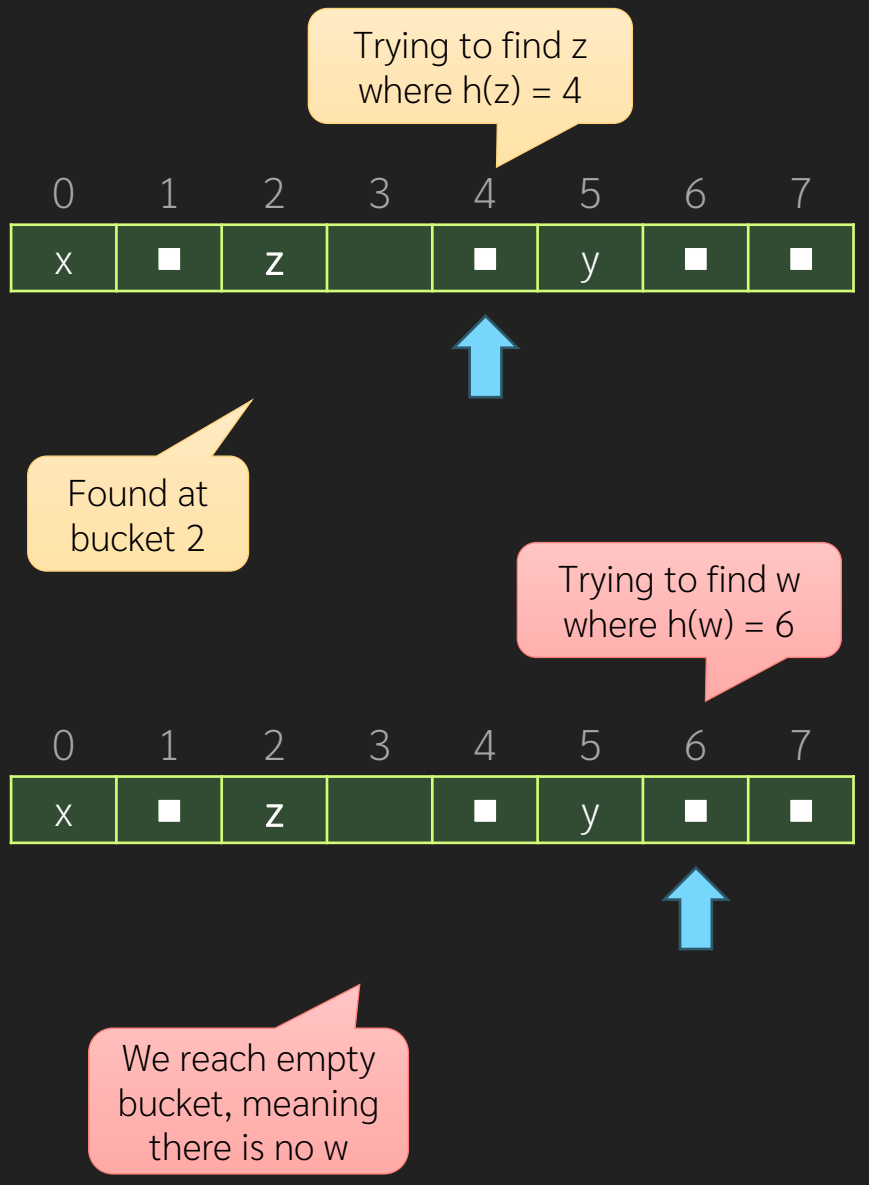
- Define `P(x, j)` as a position (index of bucket) to check for inserting x when we have collided j times
  - Define P(x,0) as h(x)
  - Open Addressing is

```
j = 0
while T[P(x,j)] is occupied
  j++
```

- For linear probing, `P(x, j) = (h(x) + j) % T.size()`

Find in Open Addressing

- To find x, Start at h(x) and using the probing function until
  - we reach a bucket that has x (which means that we have found it)
  - We reach an empty bucket (which means that there is no x)
    - Since probing function works exactly the same in both insert and find, if we reach an empty bucket, it must mean that we have not insert this data before.



Open Addressing v0.1 (LP)

- This code does not really work because there are some issue in erasing data
- Focus on linear probing first, we will fix erase soon

```
template <typename KeyT,
          typename HasherT = std::hash<KeyT>>
// Hash as Set using Open Addressing with Linear Probing
class hash_set_oa_lp {
  protected:
    typedef std::pair<bool,KeyT>      BucketT;

    std::vector<BucketT> mBuckets;
    size_t               mSize;
    HasherT              mHasher;
    float                mMaxLoadFactor;

    size_t hash_to_bucket(const KeyT &key) {
      return mHasher(key) % mBuckets.size();
    }

    size_t linear_probing(size_t start_pos, size_t col_count) {
      return start_pos + col_count;
    }

    size_t find_bucket(const KeyT &key) {
      size_t start_pos = hash_to_bucket(key);
      int col_count = 0;
      while (true) {
        size_t pos = linear_probing(start_pos,col_count++);
        if (!mBuckets[pos].first ||
             mBuckets[pos].second == key) return pos;
      }
    }
  public:
    //…
};
```

Each bucket now contains a pair where bool indicate whether this bucket has data

`linear_probing` is the probing function

`find_bucket` is used in find, insert, erase to probe for a free bucket or bucket with the data

Open Addressing v0.1 (LP)

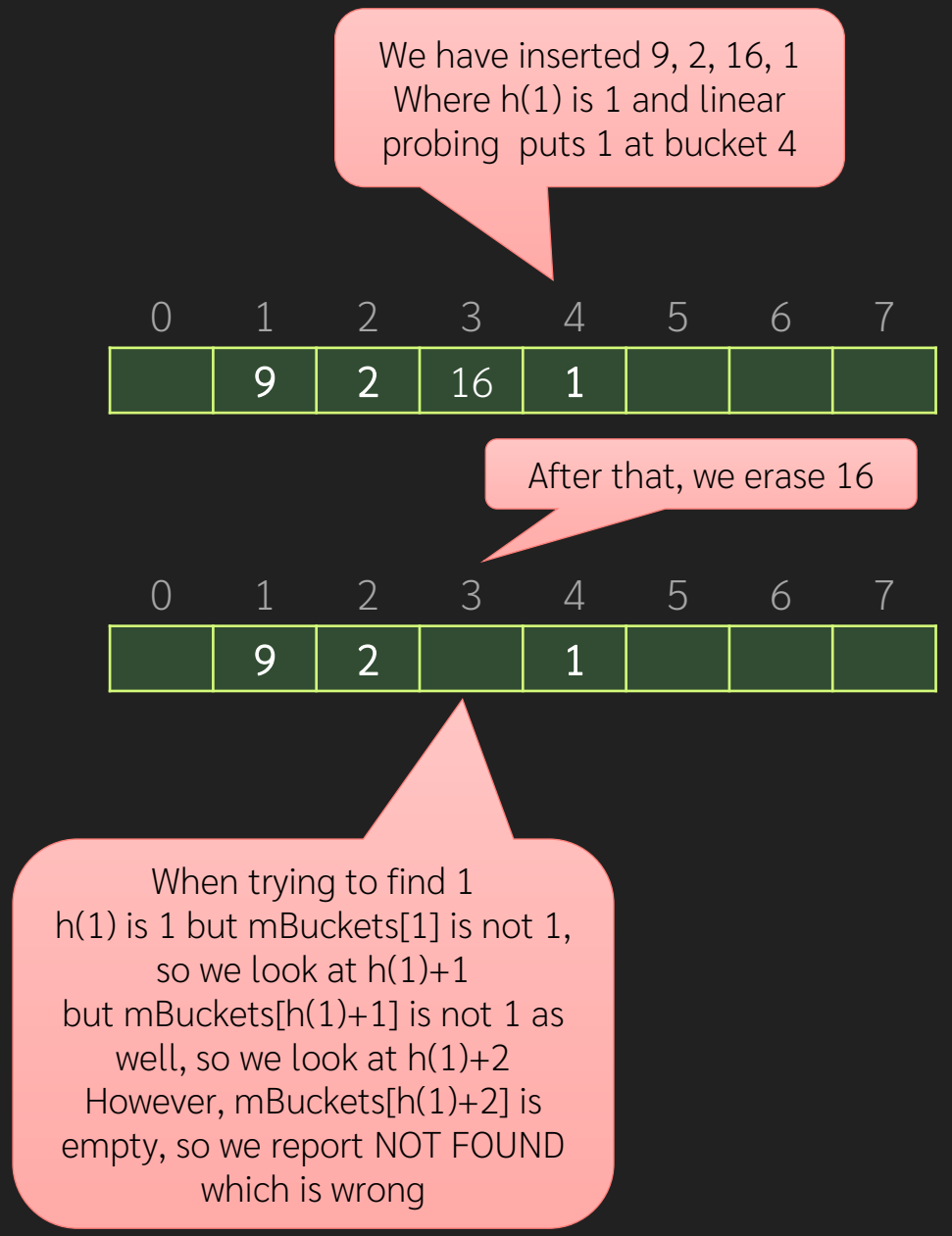
```
bool find(const KeyT &key) {
  size_t bucketIdx = find_bucket(key);
  return mBuckets[bucketIdx].first;
}

bool insert(const KeyT &key) {
  if ((mSize+1)/(0.0+mBuckets.size()) > max_load_factor())
    rehash(mSize * 2);
  size_t bucketIdx = find_bucket(key);
  if (mBuckets[bucketIdx].first) {
    //found existing key, do nothing
    return false;
  } else {
    mBuckets[bucketIdx]  = {true,key};
    mSize++;
    return true;
  }
}

bool erase(const KeyT &key) {
  size_t bucketIdx = find_bucket(key);
  if (mBuckets[bucketIdx].first) {
    //found existing key, erase
    mBuckets[bucketIdx].first = false;
    mSize--;
    return true;
  } else {
    return false;
  }
}
```

- Again, the erase function, while might seen correct, has issue
- `empty`, `size`, `load_factor`, `max_load_factor` and the constructor is the same as the separated chaining.

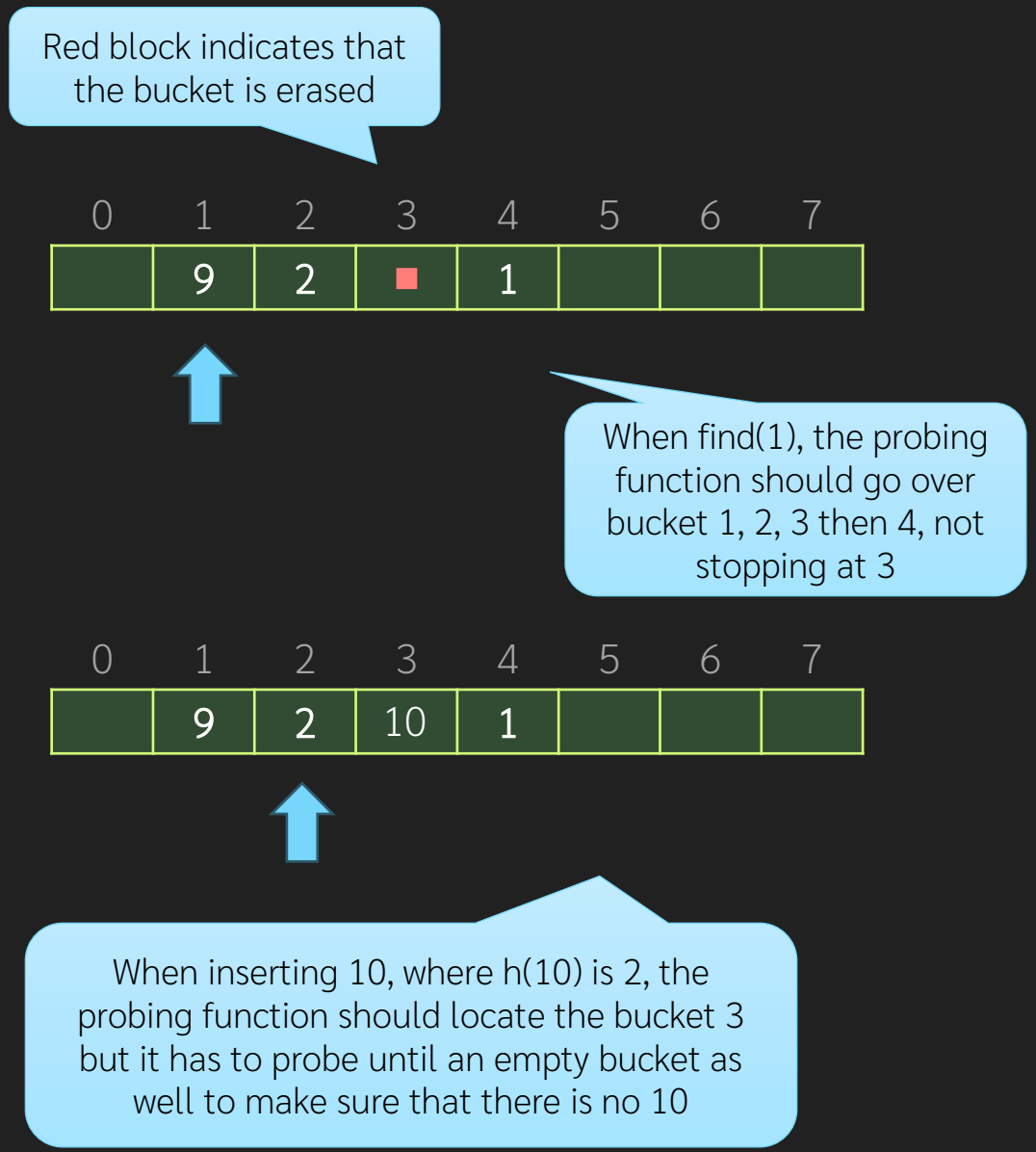
```
void rehash(size_t n) {
  if (load_factor() < max_load_factor() && n <= mBuckets.size()) return;
  n = std::max(n, mBuckets.size()*2);

  std::vector<KeyT> tmp;
  for (auto& b : mBuckets)
    if (b.first)
      tmp.push_back(b.second);
  mSize = 0;
  mBuckets = std::vector<BucketT>(n);
  for (auto& k : tmp) {
    insert(k);
  }
}
```

Erasing in Open Addressing

- Assume that we have add x and the probing function puts x at h(x) + j
- To find x later, every buckets between h(x) and h(x) + j must be occupied
  - Otherwise, the find function will report that we cannot find x
  - because we start probing from h(x) and go to h(x) + 1, h(x) + 2, ... until h(x) + j and if any of these buckets is empty, the find function will report not found



Fixing Erase in Open Addressing



- When deleting a data at a bucket, instead of marking the bucket as empty, we mark it as erased
  - Each bucket has 3 stated, EMPTY, ERASED, OCCUPIED
- The probing works just like before, however, when it the data is not found, it should return the first erased position (or the empty position if the probing does not find any erased position)
  - We can't stop when we find the erased position since we use Hash as a set (or map), which does not allow duplicate data
  - The probing must probe until it reaches an empty bucket or finds the searching data

Open Addressing v0.2 (LP)

```
enum state {EMPTY, ERASED, OCCUPIED};
typedef std::pair<state,KeyT>      BucketT;
```

```
size_t find_bucket(const KeyT &key) {
  size_t start_pos = hash_to_bucket(key);
  size_t first_erased = 0;
  bool found_erased = false;
  int col_count = 0;
  while (true) {
    size_t pos = linear_probing(start_pos,col_count++);
    switch (mBuckets[pos].first) {
      case ERASED:
        found_erased = true;
        first_erased = pos;
        break;
      case EMPTY:
        if (found_erased) return first_erased;
        return pos;
      case OCCUPIED:
        if (mBuckets[pos].second == key) return pos;
        break;
    }
  }
}
```

When detecting an ERASED bucket, just remember the location of the first one

When detecting an empty bucket, return the first erased bucket or the current bucket

- Fixed the erase problem
- Bucket is a pair of state and the data
- Find now handle all 3 states

Open Addressing v0.2 (LP)

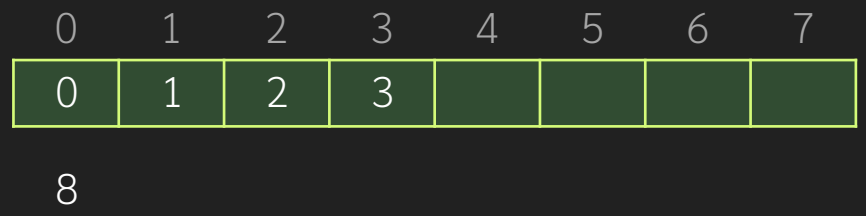
```cpp
bool find(const KeyT &key) {
  size_t bucketIdx = find_bucket(key);
  return mBuckets[bucketIdx].first == OCCUPIED;
}

bool insert(const KeyT &key) {
  if ((mSize+1)/(0.0+mBuckets.size()) > max_load_factor())
    rehash(mSize * 2);
  size_t bucketIdx = find_bucket(key);
  if (mBuckets[bucketIdx].first == OCCUPIED) {
    //found existing key, skip
    return false;
  } else {
    mBuckets[bucketIdx]  = {OCCUPIED,key};
    mSize++;
    return true;
  }
}

bool erase(const KeyT &key) {
  size_t bucketIdx = find_bucket(key);
  if (mBuckets[bucketIdx].first == OCCUPIED) {
    //found existing key, erase
    mBuckets[bucketIdx].first = ERASED;
    mSize--;
    return true;
  } else {
    return false;
  }
}
```

```cpp
void rehash(size_t n) {
  if (load_factor() < max_load_factor() && n <= mBuckets.size()) return;
  n = std::max(n, mBuckets.size()*2);

  std::vector<KeyT> tmp;
  for (auto& b : mBuckets)
    if (b.first == OCCUPIED)
      tmp.push_back(b.second);
  mSize = 0;
  mBuckets = std::vector<BucketT>(n,{EMPTY,KeyT()});
  for (auto& k : tmp) {
    insert(k);
  }
}
```

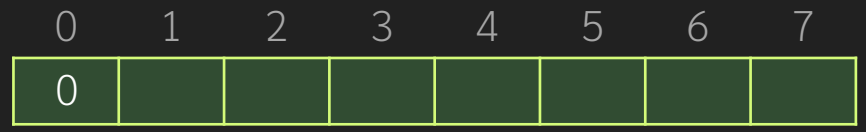
- `find`, `insert`, `erase` and `rehash` are now state-awared

Basic Analysis

- Because of open addressing, x might not be at bucket h(x) and we have to check some other buckets
  - In the worst case, we might have to check all buckets
- In Open Addressing, the order of insertion might affect which bucket is used
  - Consider inserting {0,1,2,3,8} and {0,8,1,2,3} into a hash table of size 8



8

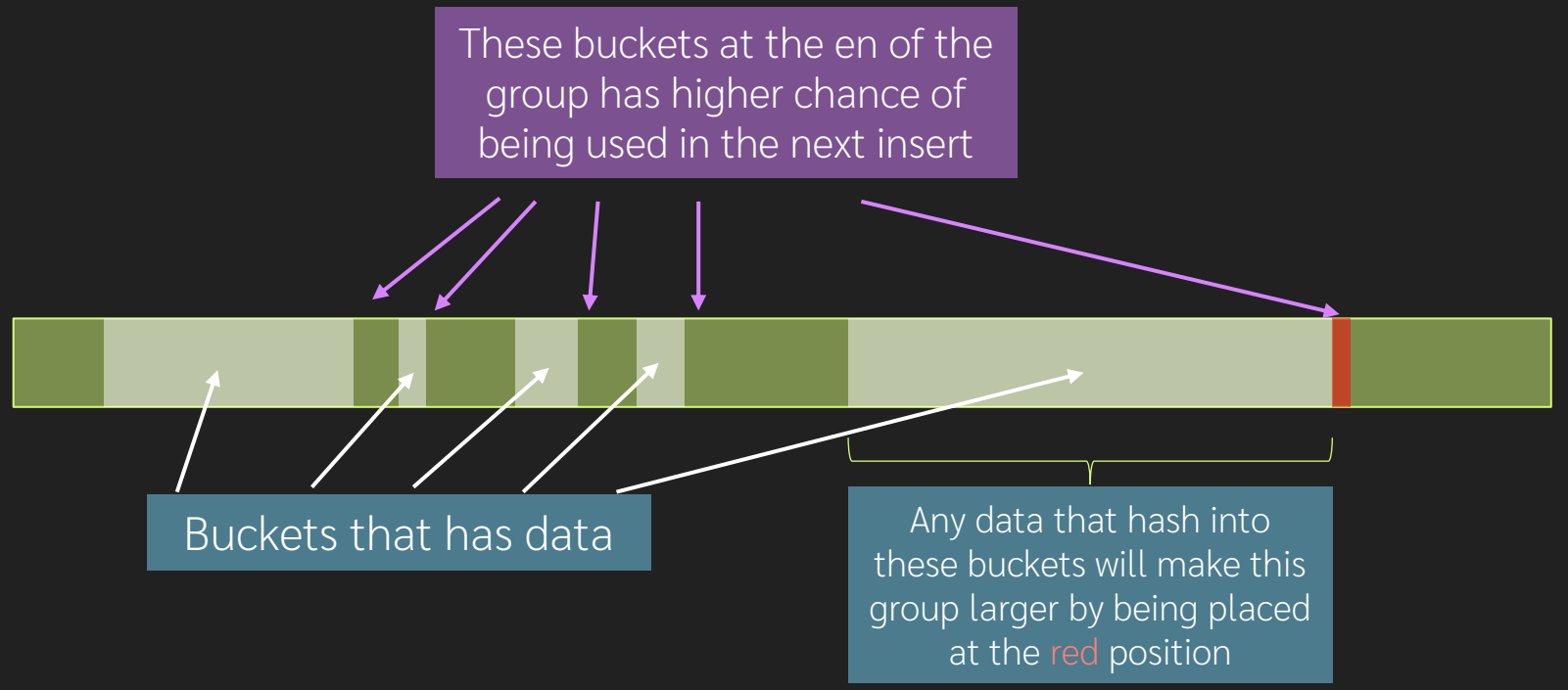


8 1 2 3

Primary Clustering Problem

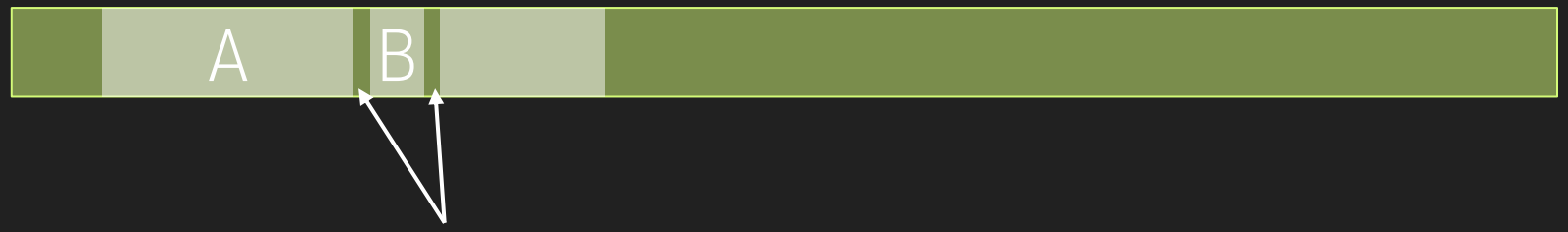
- Data with different hash value might end up getting linear probed into a large contiguous group. That group has very high chance to have another data landing in the group making the group bigger
  - In short, bigger group is more likely to get bigger

Also known as cookie monster problem because of its runaway nature



Another Cause for Primary Clustering

- Joining of multiple smaller cluster



If some data are inserted here (which is very likely because of the cluster A and B), it will join these group together, result in a very large clustering

This situation is very likely

Quadratic Probing: Fixing Primary Clustering

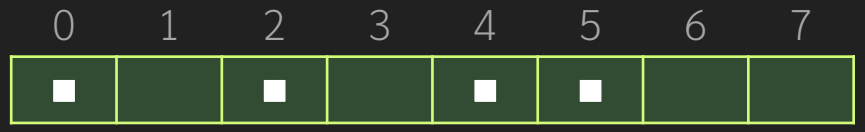
- Make probing jump over larger and larger instead of keep looking to the next empty slot
- By using Quadratic Probing
  - The formular is $P(x, j) = (h(x) + j^2) \% T.size()$

Notice the quadratic of J

- Original position is h(x)
  - First probe at h(x) + 1
  - Second probe at h(x) + 4
  - Third probe at h(x) + 9

Collision count	Linear Probing	Quadratic Probing
0	h(x)	h(x)
1	h(x) + 1	h(x) + 1
2	h(x) + 2	h(x) + 4
3	h(x) + 3	h(x) + 9
4	h(x) + 4	h(x) + 16
5	h(x) + 5	h(x) + 25

Does Quadratic Probing always find an empty bucket?



Trying to insert x where h(x) = 4

Collision count	$h(x) + j^2$	Buckets
0	4	4
1	4+1 = 5	5
2	4+4 = 8	0
3	4+9 = 13	5
4	4+16 = 20	4
5	4+25 = 29	5
6	4+36 = 40	0
7	4+49 = 53	5
8	4+64 = 68	4
9	4+81 = 85	5
10	4+100 = 104	0

- The answer is "No", QP might not be able to find an empty bucket
- Need some precondition to guarantee an empty bucket

Sufficient Condition for finding free bucket

- Prove that QP always find a free bucket
- Show that for 0 <= a < b <= m/2, P(x, a) != P(x, b), when m is prime
  - This means that, for a table of size m, the first m/2 probing of QP will check unique buckets. If load factor is at most 0.5, these m/2 probing will always find a free bucket
- Proof by contradiction, assume that P(x, a) == P(x, b)

- Table size must be a prime number
- Load factor must be at most 0.5

These are all congruent in mod m

$$h(x) + a^2 \equiv h(x) + b^2$$
$$a^2 \equiv b^2$$
$$a^2 - b^2 \equiv 0$$
$$(a - b)(a + b) \equiv 0$$

(a-b)(a+b) can't be 0 mod m because m is prime

(a-b) can't be 0 because a < b, by assumption

(a+b) is less than m hence (a+b) can't be 0 mod m

QP Summary

- Help reduce primary clustering problem
- Require that the table size is prime
- Require that load factor is maintained to not exceed 0.5
  - This is good because high load factor means less efficient find
  - 0.5 means that at, at worst, we allocate approximately 4 times of the required space
  - For example, number of buckets is 103 and we use 51 bucket, when adding one more item, the table is resized to 211 (the next prime after 103*2) holding 52 items.

Open Addressing v0.3 (QP)

- Minor adjustment from LP
- `find_bucket` use `quadratic_probing` instead
- constructor now set `mMaxLoadFactor` to 0.5 and the starting size is 11
- `rehash` use the next prime after doubling the size
- `PRIMES` is a precomputed vector of prime numbers

```
size_t quadratic_probing(size_t start_pos, size_t col_count) {
  return start_pos + col_count;
}

size_t find_bucket(const KeyT &key) {
  size_t start_pos = hash_to_bucket(key);
  size_t first_erased = 0;
  bool found_erased = false;
  int col_count = 0;
  while (true) {
    size_t pos = quadratic_probing(start_pos,col_count++);
    switch (mBuckets[pos].first) {
      case ERASED:
      //...
```

```
hash_set_oa_qp() :
  mBuckets( std::vector<BucketT>(11,{EMPTY,KeyT()}) ),
  mSize(0),
  mHasher(HasherT()),
  mMaxLoadFactor(0.5) { }
```

```
void rehash(size_t n) {
  if (load_factor() < max_load_factor() && n <= mBuckets.size()) return;
  n = std::max(n, mBuckets.size()*2);
  n = *std::lower_bound(PRIMES.begin(),PRIMES.end(),n);
  //...
```

Open Addressing v0.3 (QP)

```
class hash_set_oa_qp {
  protected:
    static const std::vector<unsigned long> PRIMES;
```

```
template <typename KeyT,
          typename HasherT>
const std::vector<unsigned long> hash_set_oa_qp<KeyT,HasherT>::PRIMES
= {
  2ul, 3ul, 5ul, 7ul, 11ul, 13ul, 17ul, 19ul, 23ul, 29ul, 31ul,
  37ul, 41ul, 43ul, 47ul, 53ul, 59ul, 61ul, 67ul, 71ul, 73ul, 79ul,
  83ul, 89ul, 97ul, 103ul, 109ul, 113ul, 127ul, 137ul, 139ul, 149ul,
  157ul, 167ul, 179ul, 193ul, 199ul, 211ul, 227ul, 241ul, 257ul,
  277ul, 293ul, 313ul, 337ul, 359ul, 383ul, 409ul, 439ul, 467ul,
  503ul, 541ul, 577ul, 619ul, 661ul, 709ul, 761ul, 823ul, 887ul,
  953ul, 1031ul, 1109ul, 1193ul, 1289ul, 1381ul, 1493ul, 1613ul,
  1741ul, 1879ul, 2029ul, 2179ul, 2357ul, 2549ul, 2753ul, 2971ul,
  3209ul, 3469ul, 3739ul, 4027ul, 4349ul, 4703ul, 5087ul, 5503ul,
  5953ul, 6427ul, 6949ul, 7517ul, 8123ul, 8783ul, 9497ul, 10273ul,
```

- `PRIMES` is precomputed and each entry grows by approx. 10%
- This is not the exhaustive list of primes
- In hash function section, we will see that having a table size as prime is good. This rehash to the prime after the double size should also be used in the Separate Chaining strategy as well.

Double Hashing

- While QP help reduce effect of the primary clustering problem, QP still has another problem called the secondary clustering problem
  - Data that has the same hash value still collide in the same sequence
- Fixed by making data jumps differently based on x
  - We need second hash function g(x)
- `P(x, j) = (h(x) + j*g(x)) % T.size()`
- Requirement for g(x)
  - `g(x) % m != 0` so that it does not keep probing the same spot as h(x)
  - `GCD(g(x), m) = 1` so that it probes every buckets
    - Can be done by making m a prime

Example

- One good example of g(x) is `g(x) = R - (x % R)` where R is a prime number less than M
- Example, adding x and y where h(x) == h(y)
- Assume that M is 7 and R is 5 and h(x) = g
  - g(x) = 5
  - g(y) = 2

Collision count	P(x, j)	P(y, j)
0	h(x)	h(y) == h(x)
1	h(x) + 5	h(x) + 2
2	h(x) + 10	h(x) + 4
3	h(x) + 15	h(x) + 6
4	h(x) + 20	h(x) + 8
5	h(x) + 25	h(x) + 10

Summary

- Linear Probing (LP)
  - `P(x, j) = (h(x) + j) % T.size()`
- Quadratic Probing (QP)
  - `P(x, j) = (h(x) + j²) % T.size()`
- Double Hashing (DH)
  - `P(x, j) = (h(x) + j*g(x)) % T.size()`
- Requirement
  - For QP and DH, `T.size()` should be prime
  - For QP, load factor must not exceed 0.5

Open Addressing Analysis

- As for the separate chaining, insert and erase depends on find and find is O(n), theoretically.
- Also just like the separate chaining, if we maintain the load factor to be small, the actual run time is near constant

Comparing Separate Chaining vs Open Addressing

- Both depends on the load factor
- For Separate Chaining (SP), since there is another data structure (std::vector) holding the data in each bucket, there is small overhead in accessing this data structure
- For Open Addressing (OA), if load factor approach 1, the number of probe increase significantly
- When load factor is small, OA tends to be faster while SP works better with higher load factor
- For both strategy, if we maintain the load factor to be small (i.e., < 1 in SP and < 0.5 in OA) the find operation takes O(1) on average

Expected Number of probing

(assume that we have uniform hash)

	Successful Find	Unsuccessful Find
Separate Chaining	$1 + \lambda$	$1 + \lambda$
Open Addressing	$\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$	$\frac{1}{1-\lambda}$

Hash Function

The goal of a hash function

- To convert a data into the index of a bucket
  - Must also consider when the hashing data is of variable length such as strings or vectors
  - This is usually done in two steps
  - First is to convert a data into one integer value
  - Next, that value is convert to mod M where M is the size of the hash table
- Hash Function must be deterministic, if x = y then h(x) must always be equal to h(y)
- Hash Function should be fast (easy to compute by CPU)
- Hash Function should be uniformly distributed
  - At least, it should span over all possible index
- Hash Function should have avalanche effect, a small change in x result in large change in h(x)

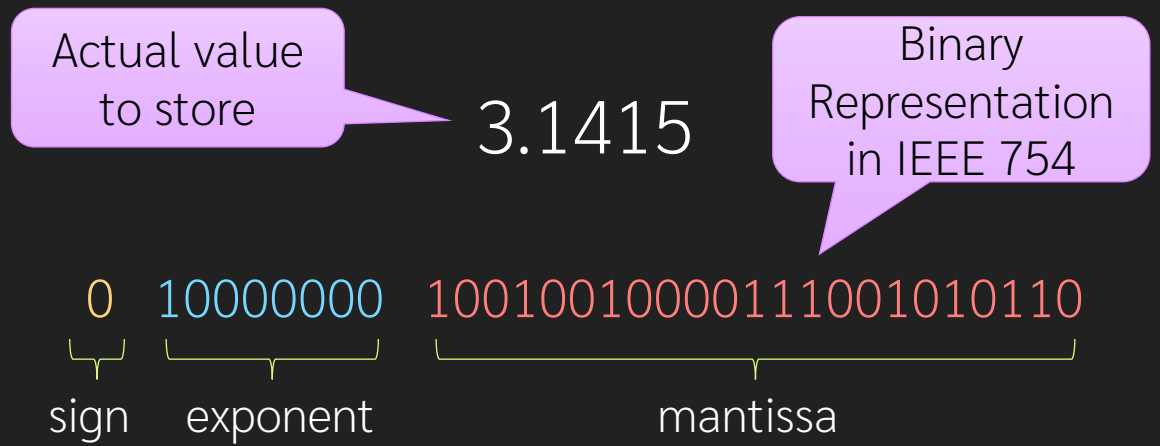
Bad Hash Function

- A constant function h(x) = c, this maps everything to a single bucket
- A hash that maps to a small subset of possible bucket, e.g., h(x) = x mod 100.
  - This function has only {0..99} as possible bucket index
- A hash that maps knowingly multiple data to the same bucket
  - For example, let x be an integer vector of size 3, a hash function h(x) = x[0] + x[1] + x[2] is bad because h({a,b,c}) = h({a,c,b}) = h({b,a,c}) = …
  - Another example, if x is a 10 digits student ID of the faculty of engineering, using the last two digits in computing a hash is bad because the last two digits of all engineering students is “21”

Trivial Hash Function

- For an integer data (such as int, char, long in C++), h(x) = x is a good hash function
  - It is uniformly distributed if we assume that x is uniformly distributed
    - This might not be true in practice but if we don’t have any information on the distribution of the input, we generally assume that x is uniform.
  - GNU C++ std::hasher use Trivial Hash Function for all integers type, bit-wisely converting negative number to positive number

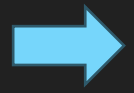
Trivial Hash for non-integer



- For non-integer data type of fixed length (such as float, double) it is possible to just re-interpret the data bit-wise into an integer and use the trivial hash
  - For example, 3.1415, when stored as a float, its binary representation can be read as integer as 1078529622
- The key idea is that any type of data is essentially stored as a sequence of bits
- For float and double, this is not recommended because rounding error nature of the float that might make two very close data maps to the same (or very close) bucket
  - It is recommended to treat float or double as a sequence of bytes and use variable length hash function instead

Hash Function for Variable Length Data

- One simple way is to treat the variable length as an array of byte and read it as a base-256 value.
  - This gives a very large number, but we will take its rightmost 32bit or 64bit
- Not really recommended because computing large radix is costly
  - The key idea of using the byte value as integer is still OK

"Somchai" 

S	o	m	c	h	a	i
83	111	109	99	104	97	105

$$83*256^6+111*256^5+109*256^4+$$
$$99*256^3+104*256^2+97*256^1+105$$

Hash Function for byte-array

- Several algorithm are proposed as a hash function to convert an array of byte into a single integer
- FNV Hash function
  - Proposed by Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo
  - Current version is FNV-1a and were used in older version of GNU C++
  - It simply XOR result of each by by BASIS where each byte is also multiply by FNV_PRIME

```
size_t Fnv_hashes(const void* ptr, size_t len)  {
  size_t hash = BASIS;
  const char* cptr = static_cast<const char*>(ptr);
  for (; len; --len) {
    hash ^= static_cast<size_t>(*cptr++);
    hash *= static_cast<size_t>(FNV_PRIME);
  }
  return hash;
}
```

BASIS is 2166136261UL
FNV_PRIME is 109951162811ULL

Hash Function for byte-array

- In GNU C++, the algorithm to convert byte-array into a hash value is Murmur Hash Function
- Similar to FNV, it perform some bit-wise operation (XOR, ROL, MUL) on each bit with some magic number, starting with a seed
  - Murmur works with group of 4 bytes
- Both FNV and Murmur try to scramble data to get avalanche effect

```
size_t hash = 0xc70f6907UL;
while (has_more_4_bytes_to_hash()) {
  k = take_4_bytes_from_data();
  //k is a group of 4 bytes
  k *= 0xcc9e2d51;
  k = (k << 15) | (k >> 17);
  k *= 0x1b873593;
  hash ^= k
  hash = (h << 13) | (h >> 19);
  hash = h * 5 + 0xe6546b64;
}

// more must be done when
// there are less than 4 bytes remain
```

Cryptographic Hash Function

- There is a similar function called cryptographic hash function (CHF) which essentially the same as a hash function, but the output is a string of fixed length
  - The CHF is very hard to find an inverse, i.e., computing x from h(x) is hard
  - The main usage is to check integrity of a data, to check that x == y, we simply check h(x) == h(y)
    - Consider checking a large file downloaded over an insecure connection. Instead of comparing the entire content of a file, we check if the value of CHF of the download file match the CHF of the original file.
  - Some well-known CHFs are MD5, SHA1, SHA256
- CHF, while seems to be uniformly distributed and pattern-less, it takes noticeable time to compute

CHF	“a”	“b”	“c”	“ab”
MD5	0cc175b9c0f1b6a831c399e269772661	92eb5ffee6ae2fec3ad71c777531578f	4a8a08f09d37b73795649038408b5f33	187ef4436122d1cc2f40dc2b92f0eba0
SHA1	86f7e437faa5a7fce15d1ddcb9eaeaea377667b8	e9d71f5ee7c92d6dc9e92ffdad17b8bd49418f98	84a516841ba77a5b4648de2cd0dfcb30ea46dbb4	da23614e02469a0d7c7bd1bdab5c9c474b1904dc
SHA256	ca978112ca1bbdcafac231b39a23dc4da786eff8147c4e72b9807785afee48bb	3e23e8160039594a33894f6564e1b1348bbd7a0088d42c4acb73eeaed59c009d	2e7d2c03a9507ae265ecf5b5356885a53393a2029d241394997265a1a25aefc6	fb8e20fc2e4c3f248c60c39bd652f3c1347298bb977b8b4d5903b85055620603

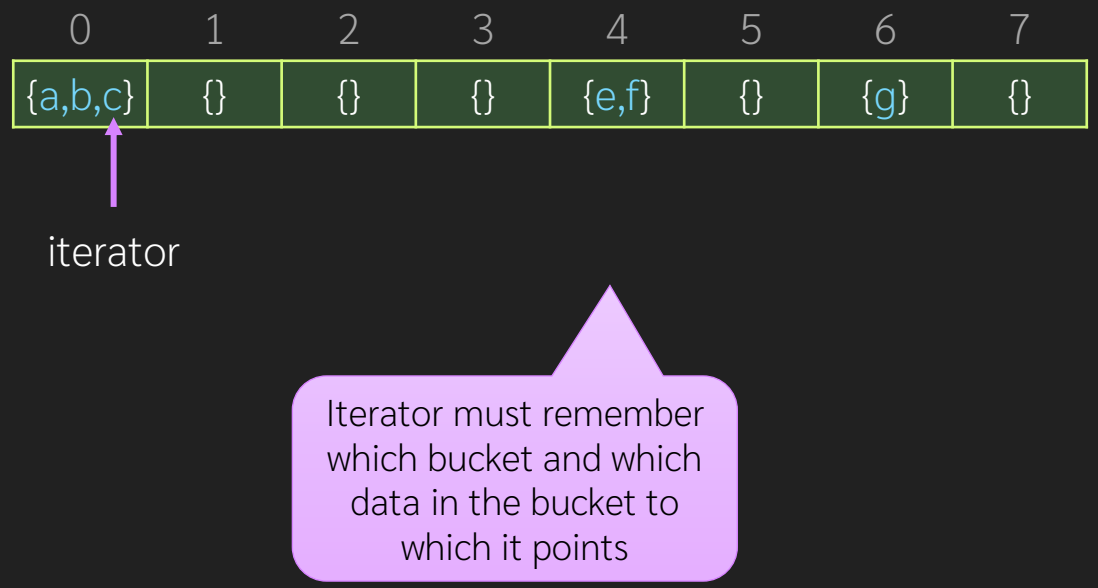
Iterator

Overview

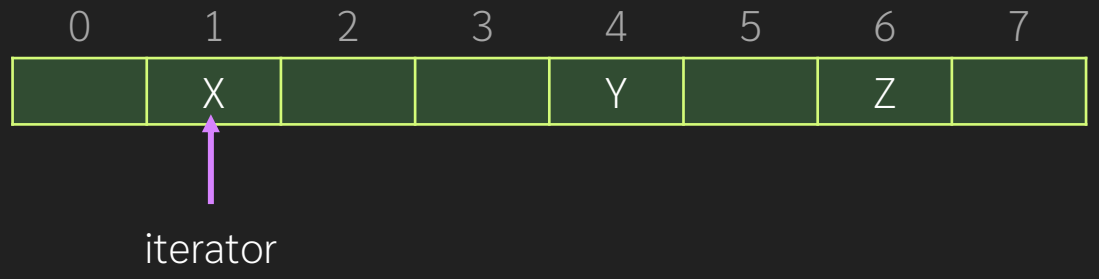
- Iterator of a hash table is very language-dependent
  - For separate chaining, this is because each bucket has another container (vector) and our iterator has to remember the state of both the current bucket and the current position in the vector
  - For Open Addressing, the iterator has to take into account the state of each bucket
- The detail implementation of the iterator is beyond the scope of this class, however, the detail is provided here for completeness. The student should focus on the key idea and performance of the iterator over the actual implementation in C++

Separate Chaining Iterator

- We need to iterate over all values in each bucket
- When reaching the last element in the bucket, we need to go to the next bucket that has data
- This make each next iteration take possible long time
  - Its O(M) where M is the table size
- The iterator is invalidated after insert and rehash
  - Because the underlying vector might be relocated



Open Addressing Iterator



- We use the iterator of the vector, but we have to skip over ERASED and EMPTY bucket
- Similarly to the Separate Chaining, the operation ++ takes O(M)
- It is also invalidated after insert or rehash

Separate Chaining Iterator Class

- `mCurValueItr` points to the current element in each bucket
- `mCurBucketItr` points to the current bucket whose element is pointed by `mCurValueItr`
- `mEndBucketItr` points to the end of mBucket. It is used to detect if our iterator reach the end
- `to_next_data` check if our iterator points to the end of a vector inside a bucket, if that is the case, it moves the iterator to the first element of a non-empty bucket
  - It is used after operator++
  - It takes O(M)

```
class hashtable_iterator  {
  friend class hash_set_sp;

  protected:
    ValueIterator  mCurValueItr;
    BucketIterator mCurBucketItr;
    BucketIterator mEndBucketItr;

    void to_next_data() {
      while (mCurBucketItr != mEndBucketItr &&
              mCurValueItr == mCurBucketItr->end()) {
        mCurBucketItr++;
        if (mCurBucketItr == mEndBucketItr) break;
        mCurValueItr = mCurBucketItr->begin();
      }
    }
```

Separate Chaining Iterator Class

```
public:

hashtable_iterator(ValueIterator element,
                   BucketIterator bucket,
                   BucketIterator endBucket) :
  mCurValueItr(element), mCurBucketItr(bucket),
  mEndBucketItr(endBucket) {
    to_next_data();
}

hashtable_iterator() { }

hashtable_iterator& operator++() {
  mCurValueItr++;
  to_next_data();
  return (*this);
}

hashtable_iterator operator++(int) {
  hashtable_iterator tmp(*this);
  operator++();
  return tmp;
}

reference operator*() {  return *mCurValueItr; }
pointer operator->()  {  return &(*mCurValueItr);  }
bool operator!=(const hashtable_iterator &other) {
  return mCurValueItr != other.mCurValueItr;
}
bool operator==(const hashtable_iterator &other) {
  return mCurValueItr == other.mCurValueItr;
}
```

- The key operation is `to_next_data` which is used in `operator++`
- The other functions simply works directly on the data pointed by `mCurValueItr`
- Below are iterator functions in the hash table class

```
    iterator begin() {
      return iterator(mBuckets.begin()->begin(),
                      mBuckets.begin(),
                      mBuckets.end());
    }

    iterator end() {
      return iterator(mBuckets[mBuckets.size()-1].end(),
                      mBuckets.end(),
                      mBuckets.end());
    }
```

Adjustment to find and insert

```
iterator find(const KeyT &key) {
  size_t bucketIdx = hash_to_bucket(key);
  auto &b = mBuckets[bucketIdx];
  auto it = find_in_bucket(b,key);
  if (it != b.end()) {
    return iterator(it,
                    mBuckets.begin() + bucketIdx,
                    mBuckets.end());
  } else
    return end();
}

std::pair<iterator,bool> insert(const KeyT &key) {
  auto it = find(key);
  if (it != end()) {
    return {it,false};
  } else {
    if (load_factor() > max_load_factor()) {
      rehash(2*mBuckets.size());
    }
    size_t bucketIdx = hash_to_bucket(key);
    auto &b = mBuckets[bucketIdx];
    b.push_back(key);
    mSize++;
    it = iterator(b.end() - 1,
                  mBuckets.begin()+bucketIdx,
                  mBuckets.end());
    return {it,true};
  }
}
```

- Iterator is returned for `find` and `insert`
- See how the iterator is created

Open Addressing Iterator Class

- `mCurBucketItr` points to the current bucket
- `mEndBucketItr` points to the end of mBucket. It is used to detect if our iterator reach the end
- `to_next_data` check if our iterator points to a bucket without data, if that is the case, it moves the iterator to the next occupied bucket (or until it reaches the end)
    - Similar to that of Separate Chaining, it is used after operator++ and takes O(M)

```
class hashtable_iterator {
  protected:
    BucketIterator mCurBucketItr;
    BucketIterator mEndBucketItr;

    void to_next_data() {
      while (mCurBucketItr != mEndBucketItr &&
             mCurBucketItr->first != OCCUPIED) {
        mCurBucketItr++;
      }
    }
```

Open Addressing Iterator Class

```
public:

hashtable_iterator(BucketIterator bucket,
                   BucketIterator endBucket) :
  mCurBucketItr(bucket), mEndBucketItr(endBucket) {
    to_next_data();
}

hashtable_iterator() { }

hashtable_iterator& operator++() {
  mCurBucketItr++;
  to_next_data();
  return (*this);
}
hashtable_iterator operator++(int) {
  hashtable_iterator tmp(*this);
  operator++();
  return tmp;
}

reference operator*() { return mCurBucketItr->second; }
pointer operator->() {  return &(*mCurBucketItr).second; }

bool operator!=(const hashtable_iterator &other) {
  return (this->mCurBucketItr != other.mCurBucketItr);
}

bool operator==(const hashtable_iterator &other) {
    return !((*this) != other);
}
```

- Almost the same as that of Separate Chaining except the constructor
- Below are iterator functions in the hash table class

```
    iterator begin() {
      return iterator(mBuckets.begin(),
                      mBuckets.end());
    }

    iterator end() {
      return iterator(mBuckets.end(),
                      mBuckets.end());
    }
```

Adjustment to find and insert

- Similar to that of Separate Chaining

```
iterator find(const KeyT &key) {
  size_t bucketIdx = find_bucket(key);
  if (mBuckets[bucketIdx].first == OCCUPIED) {
    return iterator(mBuckets.begin() + bucketIdx,mBuckets.end());
  } else {
    return end();
  }
}

std::pair<iterator,bool> insert(const KeyT &key) {
  if ((mSize+1)/(0.0+mBuckets.size()) > max_load_factor())
    rehash(mSize * 2);
  size_t bucketIdx = find_bucket(key);
  if (mBuckets[bucketIdx].first == OCCUPIED) {
    //found existing key, skip
    return {iterator(mBuckets.begin() +
bucketIdx,mBuckets.end()),false};
  } else {
    mBuckets[bucketIdx] = {OCCUPIED,key};
    mSize++;
    return {iterator(mBuckets.begin() +
bucketIdx,mBuckets.end()),true};
  }
}
```

Using Hash as a Map

Overview

- Previously, we focus on using hash as a set
- Like the Binary Search Tree (and AVL), the hash table can be adjusted to a map by storing a pair of KeyT and MappedT instead of just the KeyT
- We also need operator[]
- All of these can be done in the similar approach as in CP::map_bst
- For completeness, we also introduce EqualT as another template type which is the type used to check for equality of keys

More Template Declaration

```
template <typename KeyT,
          typename MappedT,
          typename HasherT = std::hash<KeyT>,
          typename EqualT = std::equal_to<KeyT>>
class hash_map_sp {
  protected:
    static const std::vector<unsigned long> PRIMES;

    typedef std::pair<KeyT,MappedT>                ValueT;
    typedef std::vector<ValueT>                    BucketT;
    typedef typename BucketT::iterator             ValueIterator;
    typedef typename std::vector<BucketT>::iterator BucketIterator;
```

```
template <typename KeyT,
          typename MappedT,
          typename HasherT = std::hash<KeyT>,
          typename EqualT = std::equal_to<KeyT>>
class hash_map_oa_qp {
  protected:
    static const std::vector<unsigned long> PRIMES;

    enum state {EMPTY, ERASED, OCCUPIED};
    typedef std::pair<KeyT,MappedT>                ValueT;
    typedef std::pair<state,ValueT>                BucketT;
    typedef typename
std::vector<BucketT>::iterator  BucketIterator;
```

- `MappedT` is the mapped datatype
- `EqualT` is the equality comparator class, has a default to `std::equal_to` which simply use the operator ==
- For Separate Chaining, each element in the vector in each bucket is a pair of KeyT and MappedT
- For Open Addressing, each bucket is pair<state,pair<KeyT,MappedT>>

operator[]

- Utilize the function find and insert.
- The same code can be used for both the separate chaining and open addressing

```
MappedT& operator[](const KeyT& key) {
  auto it = find(key);
  if (it == end()) {
    it = insert({key,MappedT()}).first;
  }
  return it->second;
}
```

mEqual

- To check if a given key is the same as the data in the hash table, we use mEqual instead of using operator ==
- This is similar to using mLess instead of operator< in Binary Heap or Binary Search Tree, AVL Tree

```
ValueIterator find_in_bucket(BucketT& bucket, const KeyT& key) {
   for (ValueIterator it = bucket.begin(); it != bucket.end(); it++) {
     if (mEqual(it->first, key)) return it;
   }
   return bucket.end();
}
```

```
size_t find_bucket(const KeyT &key) {
  size_t start_pos = hash_to_bucket(key);
  size_t first_erased = 0;
  bool found_erased = false;
  int col_count = 0;
  while (true) {
    size_t pos = quadratic_probing(start_pos,col_count++);
    switch (mBuckets[pos].first) {
      case ERASED:
        found_erased = true;
        first_erased = pos;
        break;
      case EMPTY:
        if (found_erased) return first_erased;
        return pos;
      case OCCUPIED:
        if (mEqual(mBuckets[pos].second.first,key)) return pos;
        break;
    }
  }
}
```

Insert

```
std::pair<iterator,bool> insert(const ValueT &value) {
  if ((mSize+1)/(0.0+mBuckets.size()) > max_load_factor())
    rehash(mSize * 2);
  size_t bucketIdx = find_bucket(value.first);
  if (mBuckets[bucketIdx].first == OCCUPIED) {
    //found existing key, skip
    return {iterator(mBuckets.begin() + bucketIdx,mBuckets.end()),false};
  } else {
    mBuckets[bucketIdx] = {OCCUPIED,value};
    mSize++;
    return {iterator(mBuckets.begin() + bucketIdx,mBuckets.end()),true};
  }
}
```

```
std::pair<iterator,bool> insert(const ValueT &value) {
  auto it = find(value.first);
  if (it != end()) {
    return {it,false};
  } else {
    if (load_factor() > max_load_factor()) {
      rehash(2*mBuckets.size());
    }
    size_t bucketIdx = hash_to_bucket(value.first);
    auto &b = mBuckets[bucketIdx];
    b.push_back(value);
    mSize++;
    it = iterator(b.end() - 1,mBuckets.begin()+bucketIdx,mBuckets.end());
    return {it,true};
  }
}
```

- Insert now takes pair<KeyT,MappedT>
- Passing the KeyT to the bucket finding

Summary

- Hash Table is fast if we maintain load factor to be small
  - The speed depends on uniformity of the hash function
- Iterator is a little slow because it might need to go over empty bucket
- There are two different collision handling strategies, Separate Chaining and Open Addressing.
  - SP is more permissive in the load factor range while OA limits it to be not exceeding 1