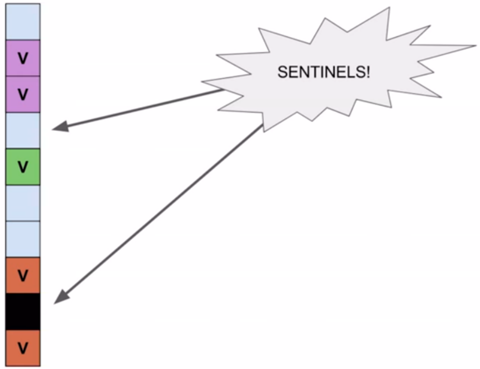
Hešovanie

kuko

5.4.2018

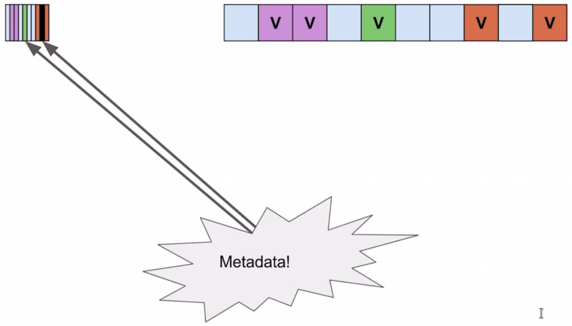
Vybrané partie z dátových štruktúr

dense_hash_set^

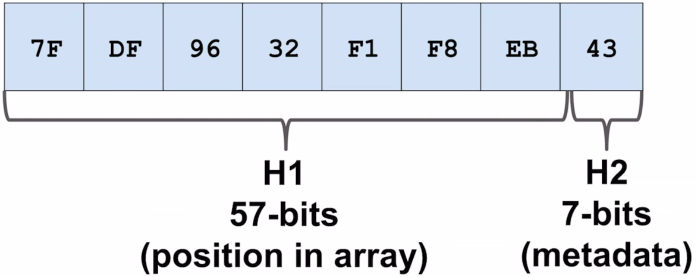


^34% true

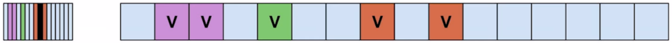
`flat_hash_set`?



?41% true

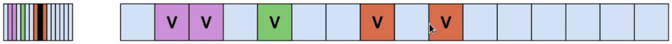

7F
DF
96
32
F1
F8
EB
43
H1
57-bits
(position in array)
H2
7-bits
(metadata)

`flat_hash_set`



`51% true`

flat_hash_set~

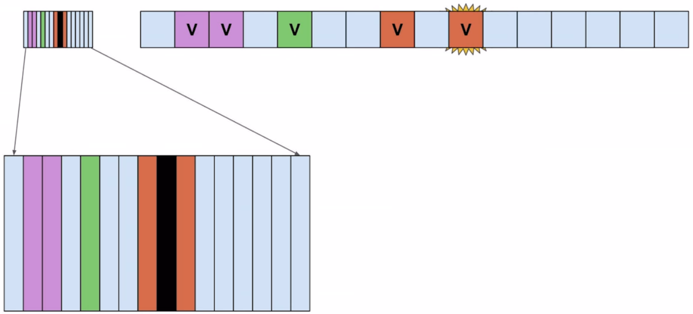


```
enum Ctrl : ctrl_t {
  kEmpty = -128,    // 0b10000000
  kDeleted = -2,    // 0b11111110
  kSentinel = -1,   // 0b11111111
  // kFull =           0b0xxxxxxx
};

size_t H1(size_t hash) { return hash >> 7; }
ctrl_t H2(size_t hash) { return hash & 0x7F; }
```

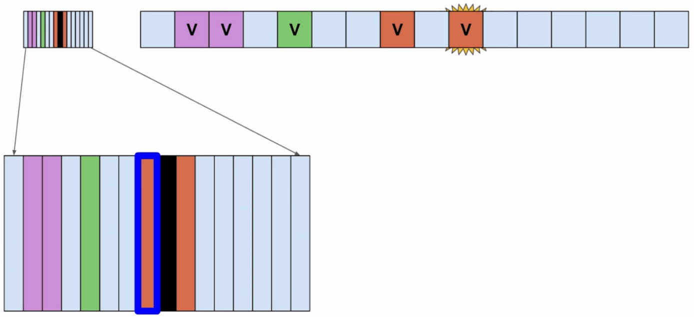
~61% true

`flat_hash_set`₦



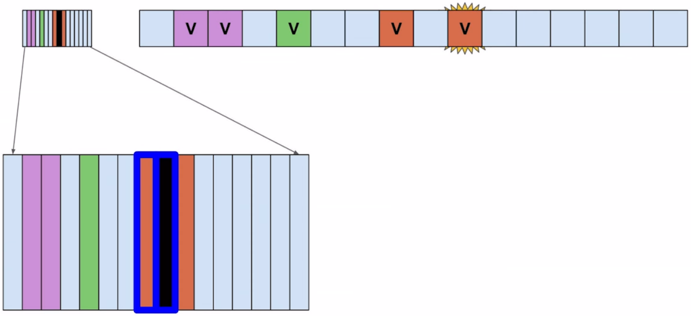
₦63% true

flat_hash_set□



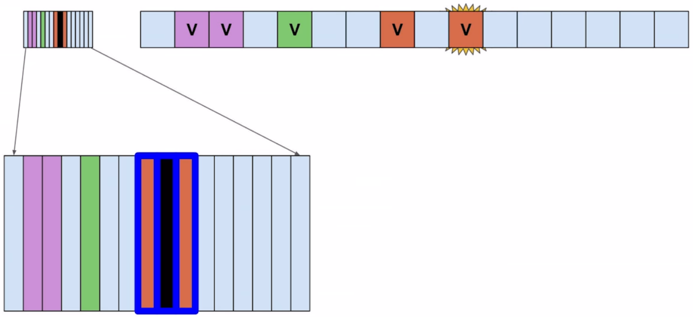
□63% true

flat_hash_set[f]



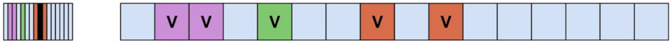
[f]63% true

flat_hash_set[F]



[F]63% true

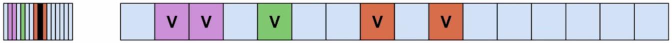
flat_hash_set^{M}



```
iterator find(const K& key, size_t hash) const {
  size_t pos = H1(hash) % size_;
  while (true) {
    if (H2(hash) == ctrl_[pos] && key == slots_[pos])
      return iterator_at(pos);
    if (ctrl_[pos] == kEmpty) return end();
    pos = (pos + 1) % size_;
  }
}
```

^{M}63% true

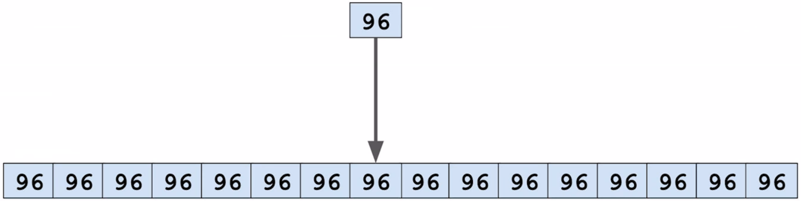
flat_hash_set[÷]



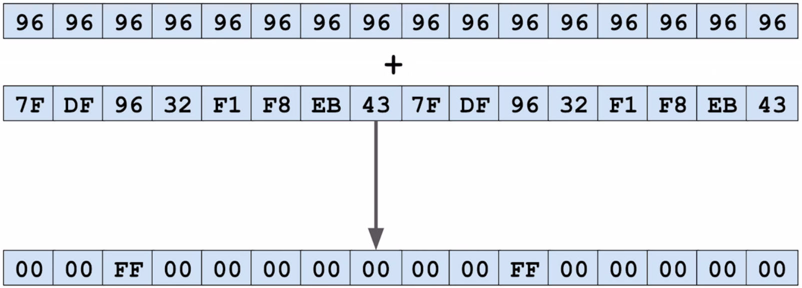
```
BitMask<uint32_t> Match(h2_t hash) const {
  auto match = _mm_set1_epi8(hash);
  return BitMask<uint32_t>(
      _mm_movemask_epi8(_mm_cmpeq_epi8(match, ctrl)));
}
```

[÷]67% true

_mm_set1_epi8

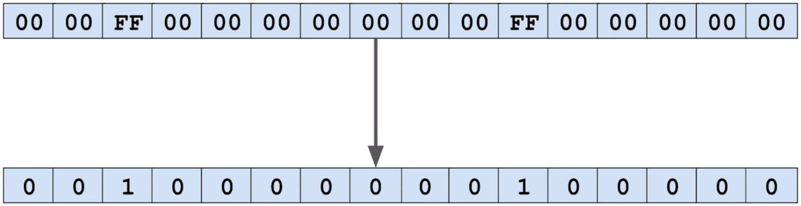


_mm_cmpeq_epi8

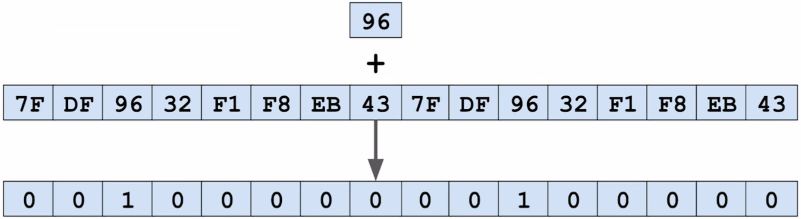


96 96 96 96 96 96 96 96 96 96 96 96 96 96 96 96
+
7F DF 96 32 F1 F8 EB 43 7F DF 96 32 F1 F8 EB 43
00 00 FF 00 00 00 00 00 00 00 FF 00 00 00 00 00

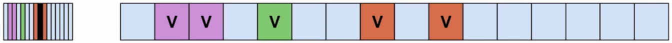
_mm_movemask_epi8



```
BitMask<uint32_t> Match(h2_t hash) const {
  auto match = _mm_set1_epi8(hash);
  return BitMask<uint32_t>(
      _mm_movemask_epi8(_mm_cmpeq_epi8(match, ctrl)));
}
```



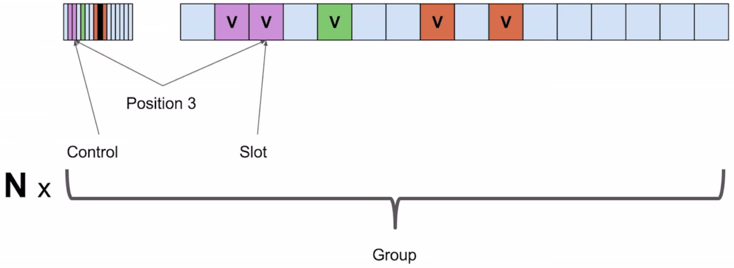
flat_hash_set÷



```
BitMask<uint32_t> Match(h2_t hash) const {
  auto match = _mm_set1_epi8(hash);
  return BitMask<uint32_t>(
      _mm_movemask_epi8(_mm_cmpeq_epi8(match, ctrl)));
}
```

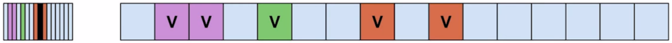
÷70% true

flat_hash_set₭



₭78% true

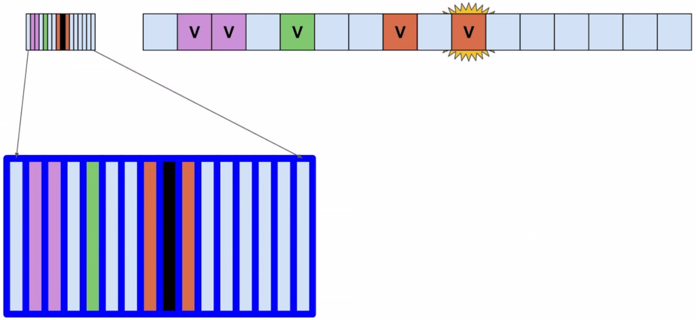
flat_hash_set¤



```
iterator find(const K& key, size_t hash) const {
  size_t group = H1(hash) % num_groups_;
  while (true) {
    Group g{ctrl_ + group * 16};
    for (int i : g.Match(H2(hash))) {
      if (key == slots_[group * 16 + i])
        return iterator_at(group * 16 + i);
    }
    if (g.MatchEmpty()) return end();
    group = (group + 1) % num_groups_;
  }
}
```

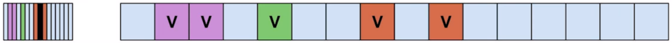
¤83% true

flat_hash_set□



□83% true

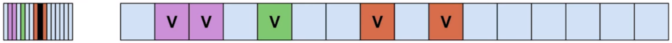
flat_hash_set₱



```
iterator find(const K& key, size_t hash) const {
  size_t group = H1(hash) % num_groups_;
  while (true) {
    Group g{ctrl_ + group * 16};
    for (int i : g.Match(H2(hash))) {
      if (key == slots_[group * 16 + i])
        return iterator_at(group * 16 + i);
    }
    if (g.MatchEmpty()) return end();
    group = (group + 1) % num_groups_;
  }
}
```

₱84% true

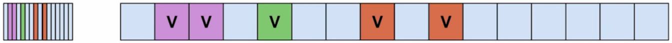
flat_hash_set[ℓ]



```
iterator find(const K& key, size_t hash) const {
  size_t group = H1(hash) % num_groups_;
  while (true) {
    Group g{ctrl_ + group * 16};
    for (int i : g.Match(H2(hash))) {
      if (key == slots_[group * 16 + i])
        return iterator_at(group * 16 + i);
    }
    if (g.MatchEmpty()) return end();
    group = (group + 1) % num_groups_;
  }
}
```

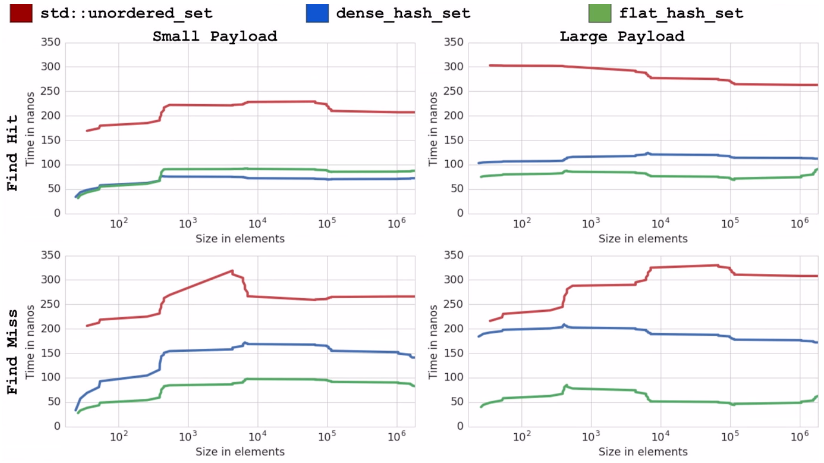
[ℓ]85% true

flat_hash_set[฿]



```
void erase(iterator it) {
  --size_;
  Group g{(it.ctrl_ - ctrl_) / 16 * 16 + ctrl_};
  *it.ctrl_ = g.MatchEmpty() ? kEmpty : kDeleted;
  it.slot_.~K()
}
```

[฿] 90% true


🟥 std::unordered_set
🟦 dense_hash_set
🟩 flat_hash_set
Small Payload
Large Payload
Find Hit
Find Miss
Time in nanos
Size in elements
0
50
100
150
200
250
300
350
10^2
10^3
10^4
10^5
10^6