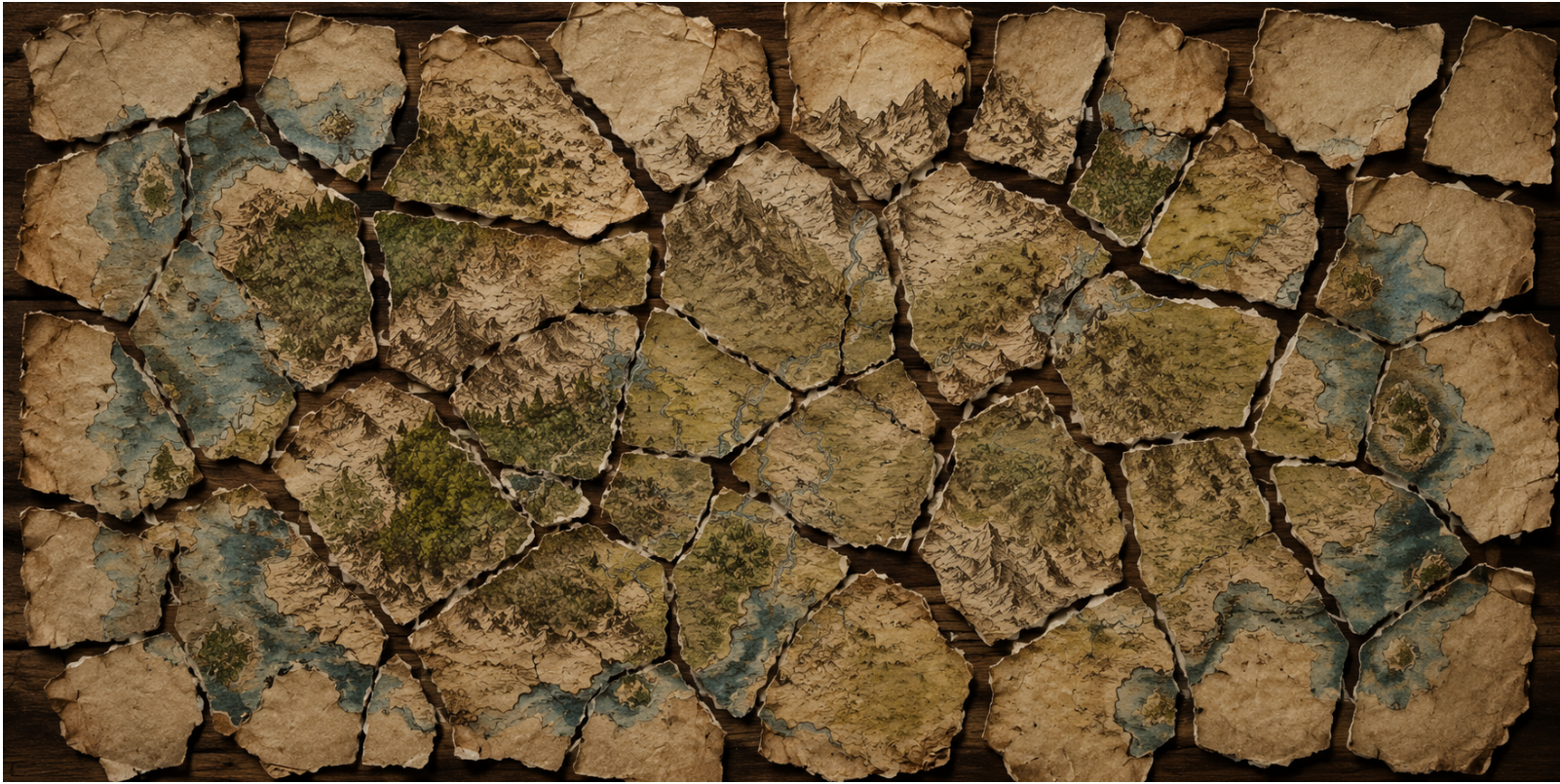
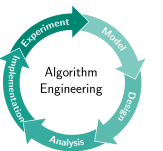




Hash Tables III

Stefan Walzer, Ragnar Groot Koerkamp, [Stefan Hermann](#) | compiled on 27. Juli 2026



Credits

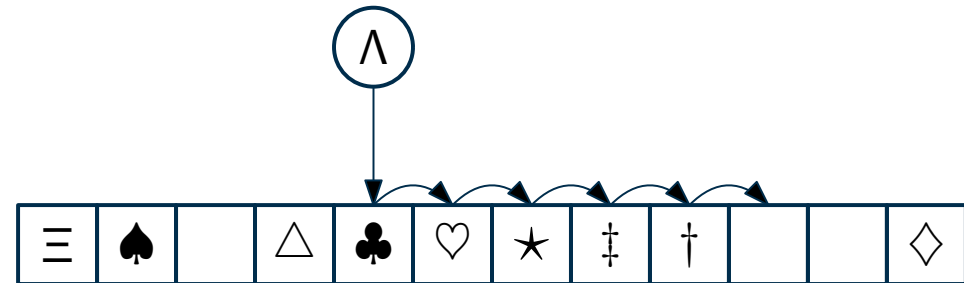
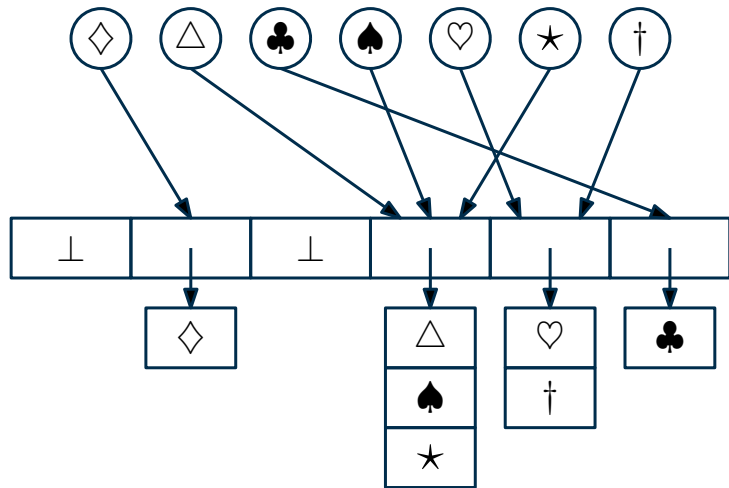
This lecture uses figures of and is based on two talks:

- 1 Joaquín M López talk about `boost::unordered_flat_map` [YT]
- 2 Matt Kulukundis talk about `absl::flat_hash_map` [YT]

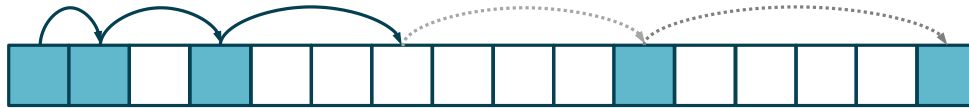
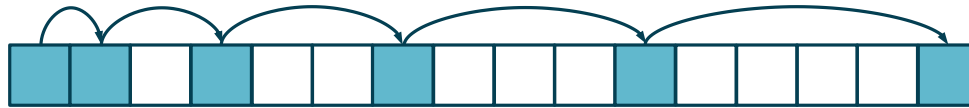
Taxonomy of Hash Tables

Taxonomy

- Closed addressing (e.g. chaining) std
- Hybrids (e.g. coalesced) Verstable
- Open addressing (e.g. probing, cuckoo) boost, ankerl, abseil



Probing: Tombstones vs Relocating



$$\left\{ \begin{array}{l} \text{APL}_{\text{successful}} = \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right) \\ \text{APL}_{\text{unsuccessful}} = \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right) \\ \text{Primary and secondary clustering} \end{array} \right.$$

APL better than linear probing
Secondary clustering

APL never decreases

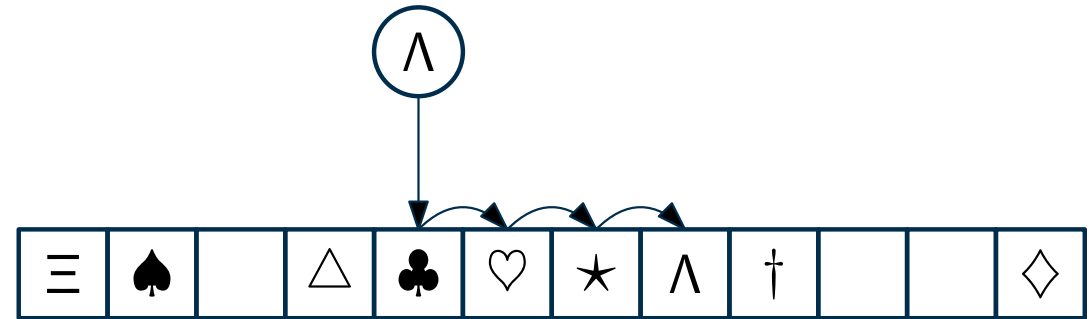
Several algorithms, increased complexity

* APL = Average Probing Length

Fingerprints

Idea

Store for each slot a small hash value of the key (e.g. 1 byte) called fingerprint.
Only if fingerprints match, make full comparison.

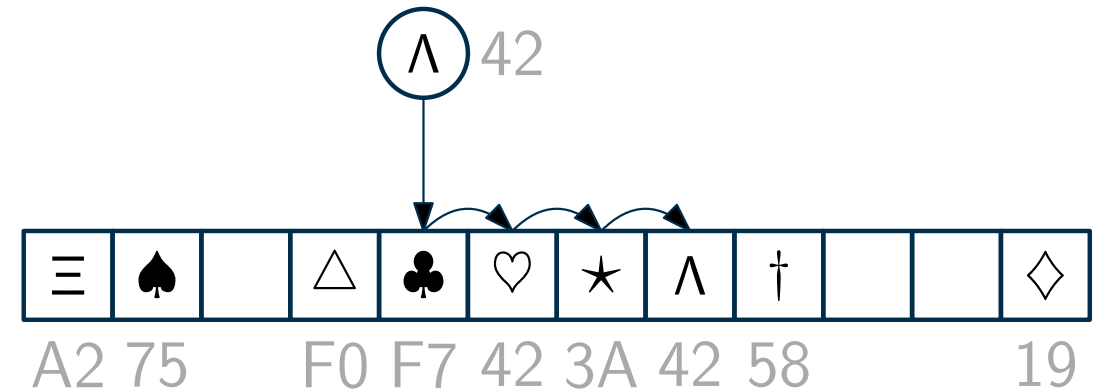


Fingerprints

Idea

Store for each slot a small hash value of the key (e.g. 1 byte) called fingerprint.

Only if fingerprints match, make full comparison.



Fingerprints

Idea

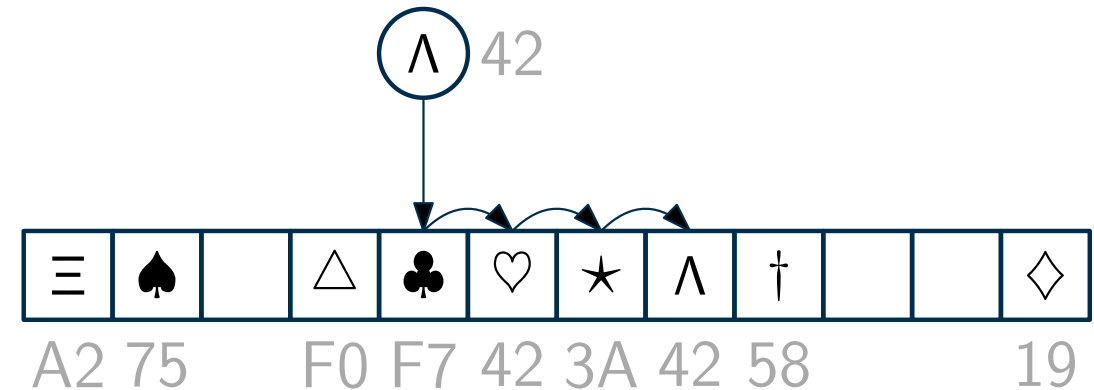
Store for each slot a small hash value of the key (e.g. 1 byte) called fingerprint.
Only if fingerprints match, make full comparison.

Implementation

- 1 Of the 64-bit hash, use lower 8 for fingerprint

64 bit hash of Λ

55	88	71	FA	BD	1C	AB	42
----	----	----	----	----	----	----	----



Fingerprints

Idea

Store for each slot a small hash value of the key (e.g. 1 byte) called fingerprint.
Only if fingerprints match, make full comparison.

Implementation

- 1 Of the 64-bit hash, use lower 8 for fingerprint
- 2 Encode empty / tombstone into the fingerprint

☰	♠		△	♣	♥	★	Λ	†			◇
A2	75	00	F0	F7	42	3A	42	58	00	00	19

Fingerprints

Idea

Store for each slot a small hash value of the key (e.g. 1 byte) called fingerprint.
Only if fingerprints match, make full comparison.

Implementation

- 1 Of the 64-bit hash, use lower 8 for fingerprint
- 2 Encode empty / tombstone into the fingerprint

☰	♠			♣	♥	★	Λ	†			◇
A2	75	00	01	F7	42	3A	42	58	00	00	19

Fingerprints

Idea

Store for each slot a small hash value of the key (e.g. 1 byte) called fingerprint.
Only if fingerprints match, make full comparison.

Implementation

- 1 Of the 64-bit hash, use lower 8 for fingerprint
- 2 Encode empty / tombstone into the fingerprint
- 3 Store fingerprints separately (memory hierarchy)

☰	♠			♣	♥	★	Λ	†			♦
A2	75	00	01	F7	42	3A	42	58	00	00	19

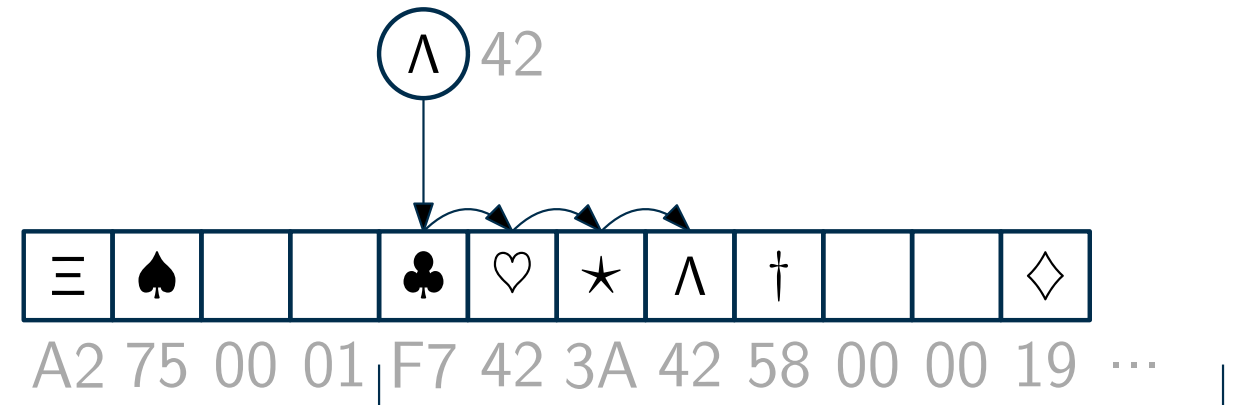
Fingerprints

Idea

Store for each slot a small hash value of the key (e.g. 1 byte) called fingerprint.
Only if fingerprints match, make full comparison.

Implementation

- 1 Of the 64-bit hash, use lower 8 for fingerprint
- 2 Encode empty / tombstone into the fingerprint
- 3 Store fingerprints separately (memory hierarchy)
- 4 Fingerprints + linear probing ♡ SIMD

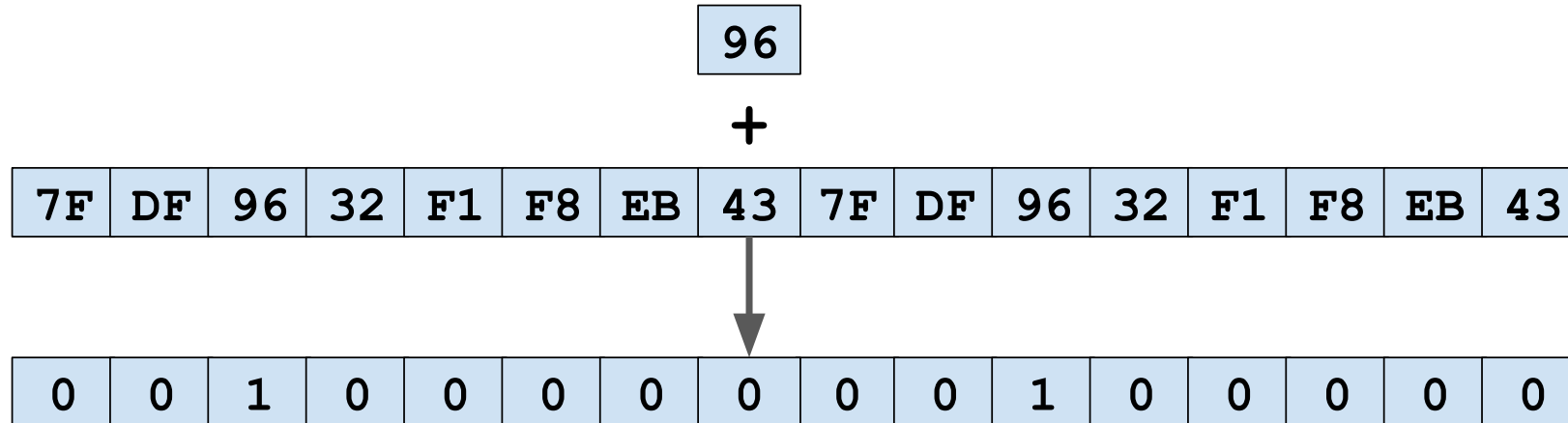


SIMD: check 16 fingerprints at once

Fingerprints + linear probing ♡ SIMD

SIMD fingerprint matching

```
uint32_t match(uint8_t fingerprint, __m128i vec1) {  
    __m128i vec2 = _mm_set1_epi8(fingerprint);  
    __m128i mask = _mm_cmpeq_epi8(vec1, vec2);  
    return _mm_movemask_epi8(mask);  
}
```

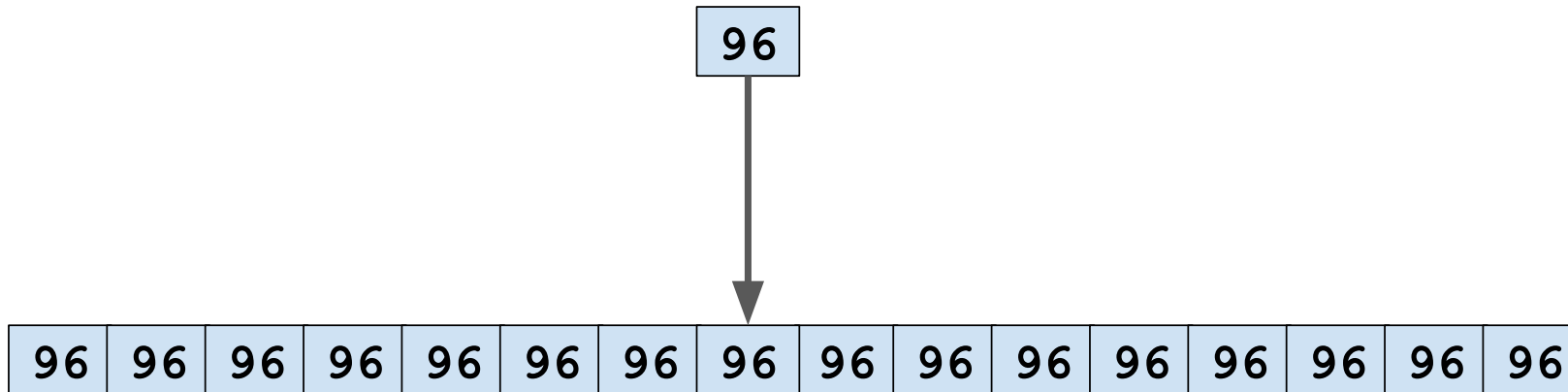


Fingerprints + linear probing ♡ SIMD

SIMD fingerprint matching

```
uint32_t match(uint8_t fingerprint, __m128i vec1) {  
    __m128i vec2 = _mm_set1_epi8(fingerprint);  
    __m128i mask = _mm_cmpeq_epi8(vec1, vec2);  
    return _mm_movemask_epi8(mask);  
}
```

`_mm_set1_epi8`

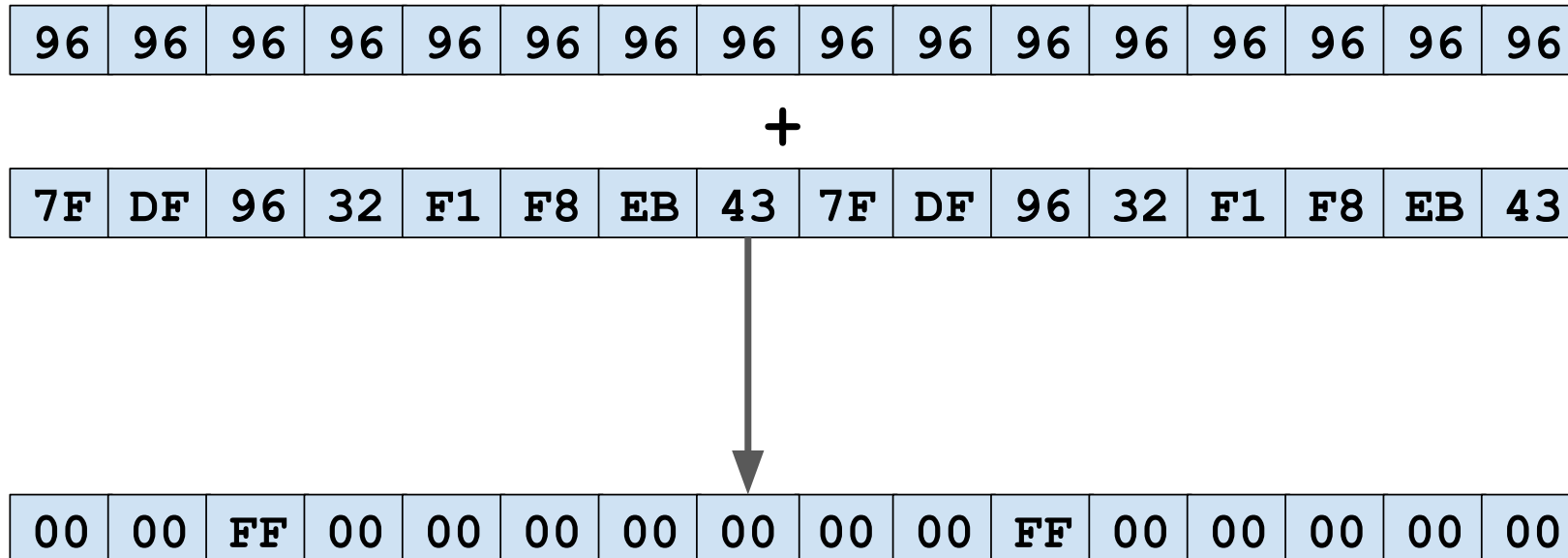


Fingerprints + linear probing ♡ SIMD

SIMD fingerprint matching

```
uint32_t match(uint8_t fingerprint, __m128i vec1) {  
    __m128i vec2 = _mm_set1_epi8(fingerprint);  
    __m128i mask = _mm_cmpeq_epi8(vec1, vec2);  
    return _mm_movemask_epi8(mask);  
}
```

`_mm_cmpeq_epi8`

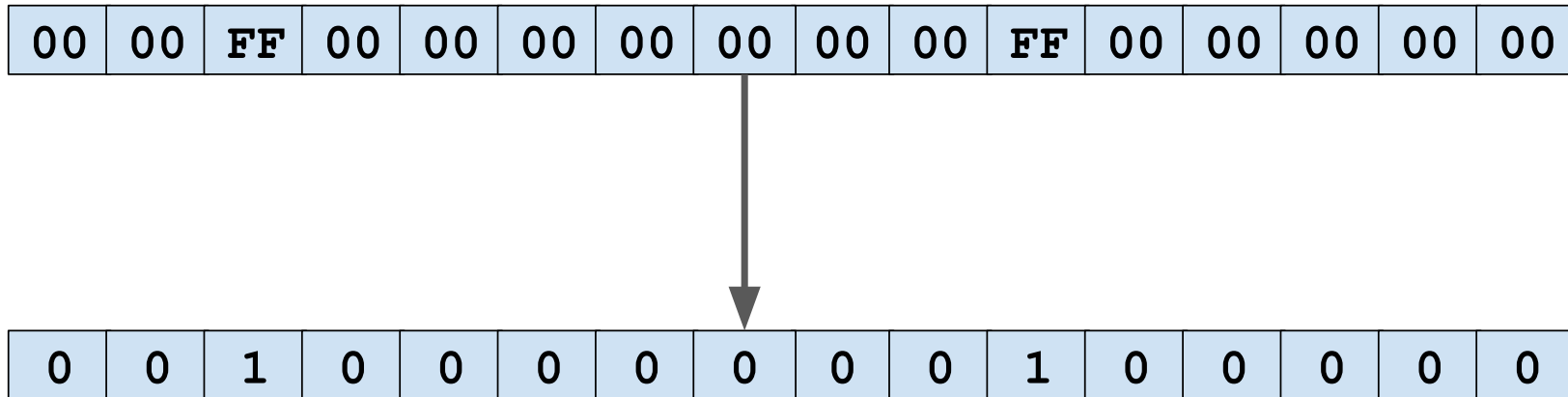


Fingerprints + linear probing ♡ SIMD

SIMD fingerprint matching

```
uint32_t match(uint8_t fingerprint, __m128i vec1) {  
    __m128i vec2 = _mm_set1_epi8(fingerprint);  
    __m128i mask = _mm_cmpeq_epi8(vec1, vec2);  
    return _mm_movemask_epi8(mask);  
}
```

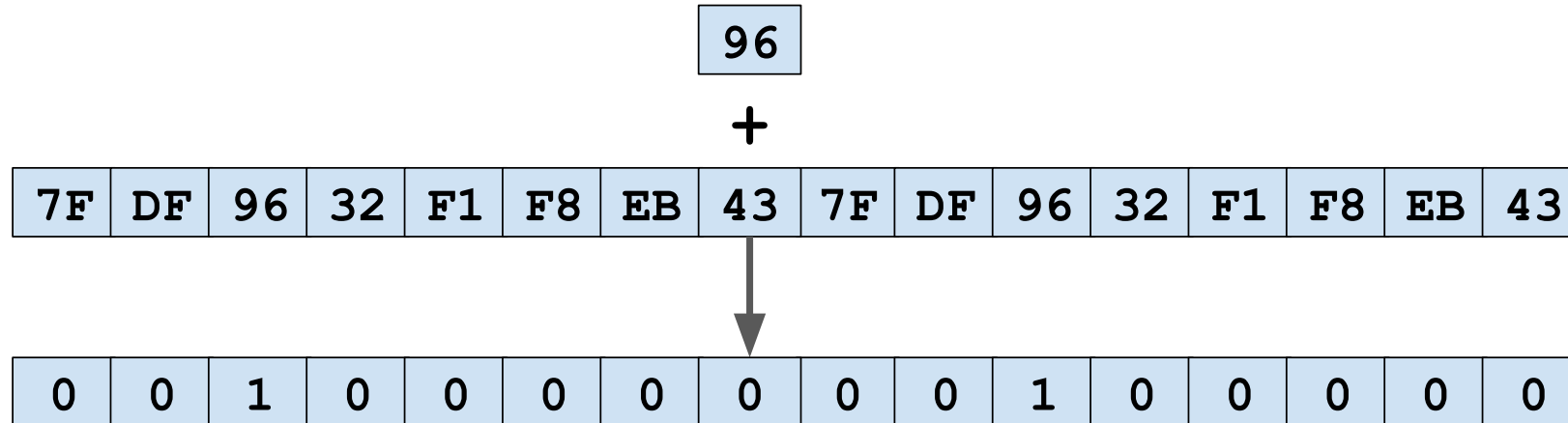
`_mm_movemask_epi8`



Fingerprints + linear probing ♡ SIMD

SIMD fingerprint matching

```
uint32_t match(uint8_t fingerprint, __m128i vec1) {  
    __m128i vec2 = _mm_set1_epi8(fingerprint);  
    __m128i mask = _mm_cmpeq_epi8(vec1, vec2);  
    return _mm_movemask_epi8(mask);  
}
```



Fingerprints + linear probing ♡ SIMD

SIMD fingerprint matching

```
uint32_t match(uint8_t fingerprint, __m128i vec1) {  
    __m128i vec2 = _mm_set1_epi8(fingerprint);  
    __m128i mask = _mm_cmpeq_epi8(vec1, vec2);  
    return _mm_movemask_epi8(mask);  
}
```

Find (simplification of boost::unordered_flat_map & absl::flat_hash_map)

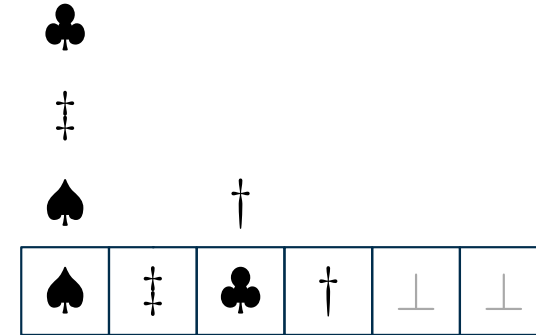
```
uint8_t fingerprint = get_fingerprint(hash);           // between 2 and 255  
size_t index = pos(hash);  
  
while (true) {  
    __m128i vec_fingerprints = _mm_loadu_si128(start + pos);  
    uint32_t mask = match(fingerprint, vec_fingerprints);  
    while (mask) {  
        size_t bit = __builtin_ctz(mask);              // find next candidate  
        size_t slot = (index + bit) & (capacity - 1);  
        if (keys[slot] == key) return &keys[slot];  
        mask &= mask - 1;                             // clear lowest set bit  
    }  
    if (match(0, vec_fingerprints)) return nullptr;    // No match found: if the group has an empty slot, key is absent  
    index = (index + 16) & (capacity - 1);              // capacity is power of two  
}
```

Robin Hood in Practice

Linear Probing Robin Hood

Keep elements ordered by start position.

Advantage: early stopping on negative query

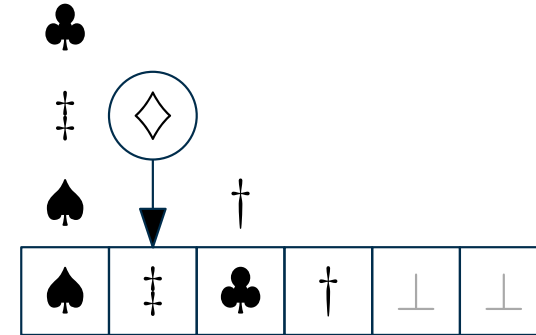


Robin Hood in Practice

Linear Probing Robin Hood

Keep elements ordered by start position.

Advantage: early stopping on negative query

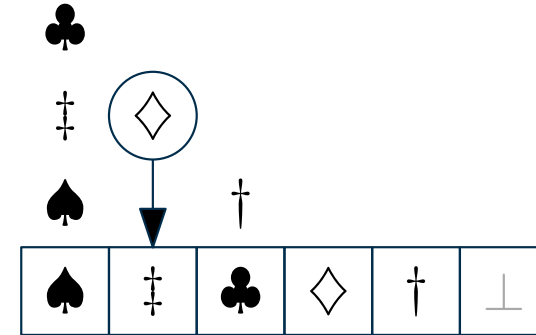


Robin Hood in Practice

Linear Probing Robin Hood

Keep elements ordered by start position.

Advantage: early stopping on negative query

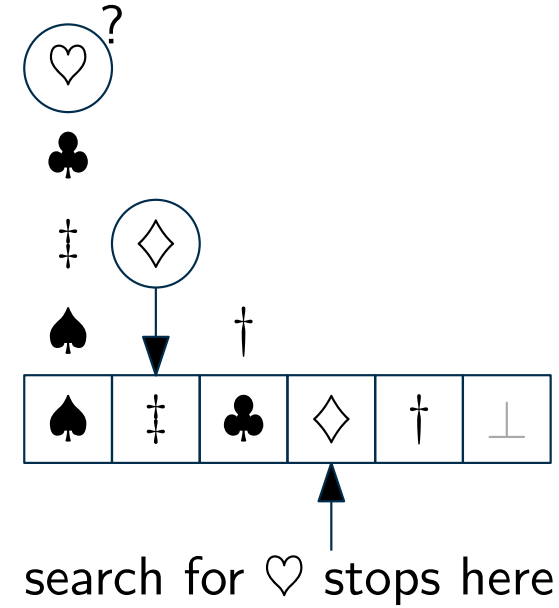


Robin Hood in Practice

Linear Probing Robin Hood

Keep elements ordered by start position.

Advantage: early stopping on negative query



Robin Hood in Practice

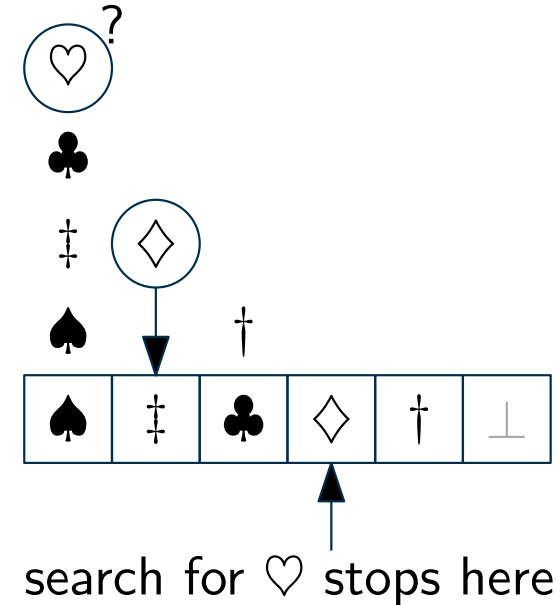
Linear Probing Robin Hood

Keep elements ordered by start position.

Advantage: early stopping on negative query

What we showed (Hashing I)

Expected time for	Negative Query	Avg. Pos. Query	Insertion
Linear Probing	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$
+ Robin Hood	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$



Robin Hood in Practice

Linear Probing Robin Hood

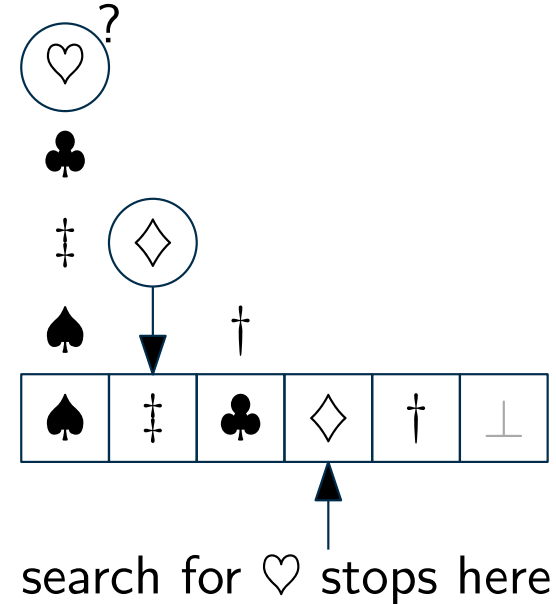
Keep elements ordered by start position.
Advantage: early stopping on negative query

What we showed (Hashing I)

Expected time for	Negative Query	Avg. Pos. Query	Insertion
Linear Probing	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$
+ Robin Hood	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$

Downside I

Many hash evaluations to evaluate probing distances.
Idea: Store probing distance explicitly



Robin Hood in Practice

Linear Probing Robin Hood

Keep elements ordered by start position.

Advantage: early stopping on negative query

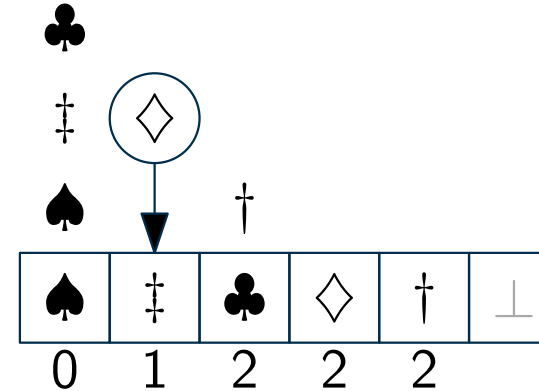
What we showed (Hashing I)

Expected time for	Negative Query	Avg. Pos. Query	Insertion
Linear Probing	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$
+ Robin Hood	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$

Downside I

Many hash evaluations to evaluate probing distances.

Idea: Store probing distance explicitly



Robin Hood in Practice

Linear Probing Robin Hood

Keep elements ordered by start position.

Advantage: early stopping on negative query

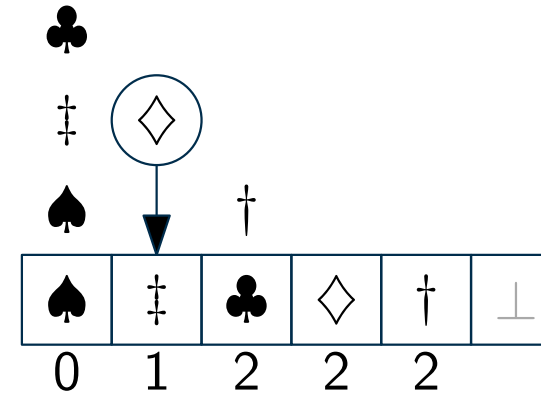
What we showed (Hashing I)

Expected time for	Negative Query	Avg. Pos. Query	Insertion
Linear Probing	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$
+ Robin Hood	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$

Downside I

Many hash evaluations to evaluate probing distances.

Idea: Store probing distance explicitly



Downside II

Insertions have to maintain the order:

Lots of moving around elements.

Idea: Store elements left justified and maintain a pointer (very fast iteration)

Robin Hood in Practice

Linear Probing Robin Hood

Keep elements ordered by start position.

Advantage: early stopping on negative query

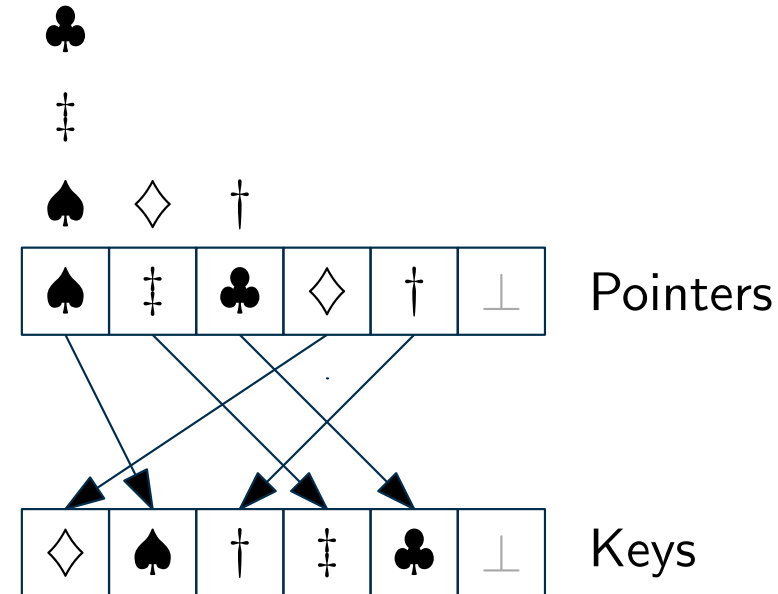
What we showed (Hashing I)

Expected time for	Negative Query	Avg. Pos. Query	Insertion
Linear Probing	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$
+ Robin Hood	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$

Downside I

Many hash evaluations to evaluate probing distances.

Idea: Store probing distance explicitly



Downside II

Insertions have to maintain the order:

Lots of moving around elements.

Idea: Store elements left justified and maintain a pointer (very fast iteration)

Robin Hood in Practice

Linear Probing Robin Hood

Keep elements ordered by start position.

Advantage: early stopping on negative query

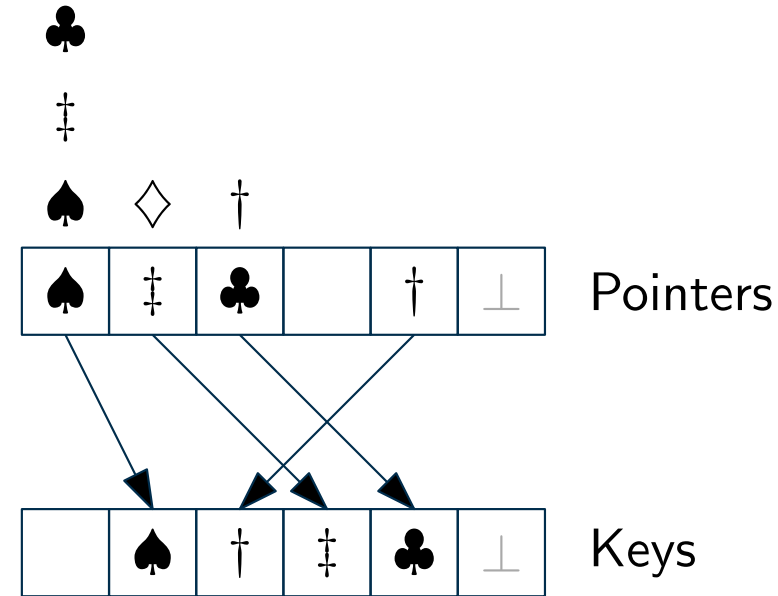
What we showed (Hashing I)

Expected time for	Negative Query	Avg. Pos. Query	Insertion
Linear Probing	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$
+ Robin Hood	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$

Downside I

Many hash evaluations to evaluate probing distances.

Idea: Store probing distance explicitly



Downside II

Insertions have to maintain the order:

Lots of moving around elements.

Idea: Store elements left justified and maintain a pointer (very fast iteration)

Robin Hood in Practice

Linear Probing Robin Hood

Keep elements ordered by start position.

Advantage: early stopping on negative query

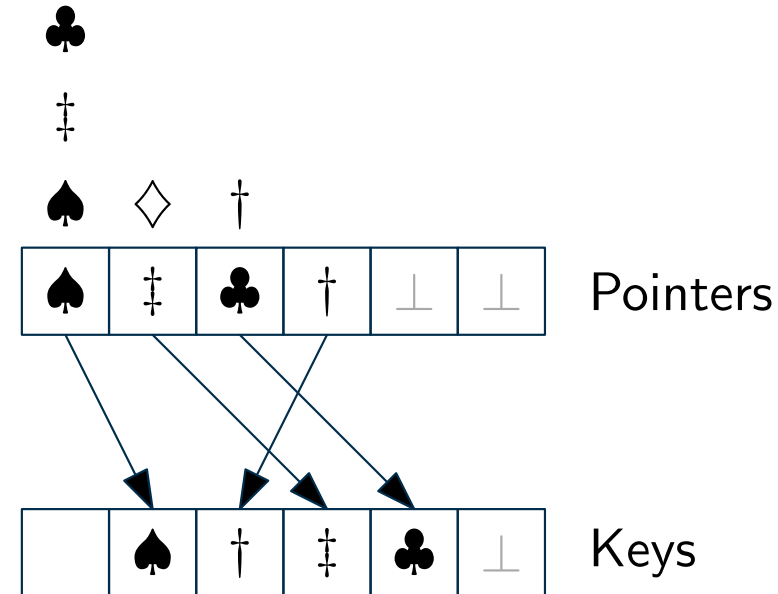
What we showed (Hashing I)

Expected time for	Negative Query	Avg. Pos. Query	Insertion
Linear Probing	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$
+ Robin Hood	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$

Downside I

Many hash evaluations to evaluate probing distances.

Idea: Store probing distance explicitly



Downside II

Insertions have to maintain the order:

Lots of moving around elements.

Idea: Store elements left justified and maintain a pointer (very fast iteration)

Robin Hood in Practice

Linear Probing Robin Hood

Keep elements ordered by start position.

Advantage: early stopping on negative query

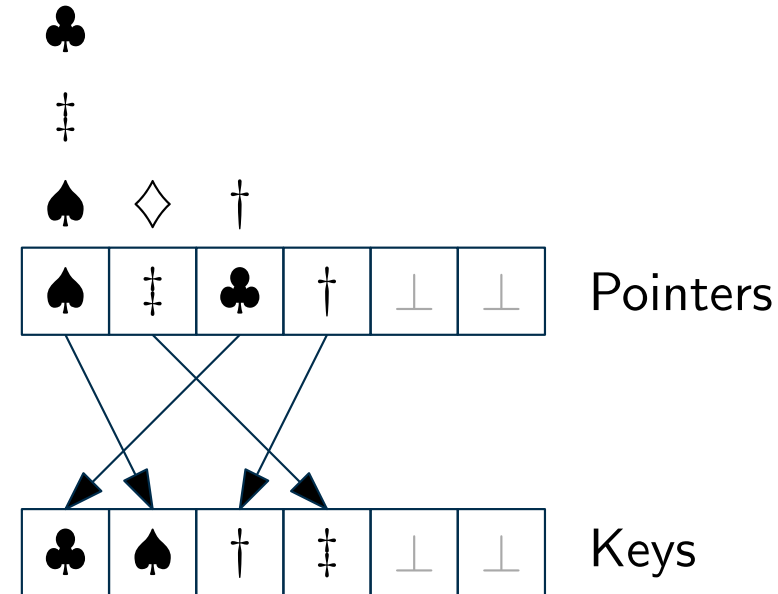
What we showed (Hashing I)

Expected time for	Negative Query	Avg. Pos. Query	Insertion
Linear Probing	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$
+ Robin Hood	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{(1-\alpha)^2}\right)$

Downside I

Many hash evaluations to evaluate probing distances.

Idea: Store probing distance explicitly



Downside II

Insertions have to maintain the order:

Lots of moving around elements.

Idea: Store elements left justified and maintain a pointer (very fast iteration)

ankerl::unordered_map

Structure

```
struct Bucket {  
    uint32_t distance_and_fingerprint; // 24 bit distance + 8 bit fingerprint  
    uint32_t value_idx; // pointer to data  
}  
vector<Bucket> buckets;  
vector<Key> keys; // left justified
```

ankerl::unordered_map

Structure

```
struct Bucket {  
    uint32_t distance_and_fingerprint; // 24 bit distance + 8 bit fingerprint  
    uint32_t value_idx; // pointer to data  
}  
vector<Bucket> buckets;  
vector<Key> keys; // left justified
```

Find

```
uint32_t dist_and_fingerprint = get_fingerprint(hash);  
size_t index = pos(hash);  
Bucket* bucket = &buckets[index];  
  
while (true) {  
    if (dist_and_fingerprint == bucket->dist_and_fingerprint) {  
        if (key == keys[bucket->value_idx]) {  
            return &keys[bucket->value_idx];  
        }  
    } else if (dist_and_fingerprint > bucket->dist_and_fingerprint) {  
        return nullptr;  
    }  
    dist_and_fingerprint = dist_inc(dist_and_fingerprint);  
    index = pos_inc(index);  
    bucket = &buckets[index];  
}
```

Possible Exam Questions

- What are the two ways of dealing with deletions? (tombstone, relocation) What are the advantages and disadvantages?
- What are advantages of fingerprints?
 - Fewer full key comparisons
 - Fingerprints are smaller (memory hierarchy)
 - 16 fingerprint comparisons at once using SIMD
 - encoding empty / tombstone information
- What are the downsides to Robin Hood hashing? What can we do about it? How do the operations work with those modifications?