13. C++ advanced (III): Functors and Lambda

Functors: Motivation

A simple output filter

```
template <typename T, typename Function>
void filter(const T& collection, Function f){
    for (const auto& x: collection)
        if (f(x)) std::cout << x << " ";
    std::cout << "\n";
}
```

Functors: Motivation

```
template <typename T, typename Function>
void filter(const T& collection, Function f);

template <typename T>
bool even(T x){
%
   return x % 2 == 0;
}

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
filter(a,even<int>); // output: 2,4,6,16
```

Functor: object with overloaded operator ()

```
class GreaterThan{
  int value; // state
  public:
  GreaterThan(int x):value{x}{}

  bool operator() (int par) const {
    return par > value;
  }
};
```

Functor is a callable object. Can be understood as a stateful function.

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;
filter(a,GreaterThan(value)); // 9,11,16,19
```

Functor: object with overloaded operator ()

```
template <typename T>
class GreaterThan{
    T value;
public:
    GreaterThan(T x):value{x}{}

    bool operator() (T par) const{
        return par > value;
    }
};
```

also works with a template, of course

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;
filter(a,GreaterThan<int>(value)); // 9,11,16,19
```

The same with a Lambda-Expression

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;

filter(a, [value](int x) {return x > value;} );
```

$$\boxed{\texttt{[value](int x) \{return x > value;\}}}$$

Sum of Elements – Old School

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int sum = 0;
for (auto x: a)
  sum += x;
std::cout << sum << "\n"; // 83
```

Sum of Elements – with Functor

```cpp
template <typename T>
struct Sum{
    T value = 0;

    void operator() (T par){ value += par; }
};

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
Sum<int> sum;
// for_each copies sum: we need to copy the result back
sum = std::for_each(a.begin(), a.end(), sum);
std::cout << sum.value << std::endl; // 83
```

Sum of Elements – with References[16]

```
template <typename T>
struct SumR{
    T& value;
    SumR (T& v):value{v} {}

    void operator() (T par){ value += par; }
};

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int s=0;
SumR<int> sum{s};
// cannot (and do not need to) assign to sum here
std::for_each(a.begin(), a.end(), sum);
std::cout << s << std::endl; // 83
```

[16]Of course this works, very similarly, using pointers

Sum of Elements – with Λ

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int s=0;

std::for_each(a.begin(), a.end(), [&s] (int x) {s += x;} );

std::cout << s << "\n";
```

Sorting, different

```
// pre: i >= 0
// post: returns sum of digits of i
int q(int i){
    int res =0;
    for(;i>0;i/=10)
        res += i % 10;
    return res;
}

std::vector<int> v {10,12,9,7,28,22,14};
std::sort (v.begin(), v.end(),
  [] (int i, int j) { return q(i) < q(j);}
);
```

Sorting, different

```
// pre: i >= 0
// post: returns sum of digits of i
int q(int i){
    int res =0;
    for(;i>0;i/=10)
        res += i % 10;
    return res;
}

std::vector<int> v {10,12,9,7,28,22,14};
std::sort (v.begin(), v.end(),
  [] (int i, int j) { return q(i) < q(j);}
);
```

Now $v =$

Sorting, different

```
// pre: i >= 0
// post: returns sum of digits of i
int q(int i){
    int res =0;
    for(;i>0;i/=10)
        res += i % 10;
    return res;
}

std::vector<int> v {10,12,9,7,28,22,14};
std::sort (v.begin(), v.end(),
  [] (int i, int j) { return q(i) < q(j);}
);
```

Now $v = 10, 12, 22, 14, 7, 9, 28$ (sorted by sum of digits)

Lambda-Expressions in Detail

$$\underbrace{\texttt{[value]}}_{\text{capture}}\ \underbrace{\texttt{(int x)}}_{\text{parameters}}\ \underbrace{\texttt{->bool}}_{\text{return type}}\ \underbrace{\texttt{\{return x > value;\}}}_{\text{statement}}$$

Closure

```
[value] (int x) ->bool {return x > value;}
```

- Lambda expressions evaluate to a temporary object – a closure
- The closure retains the execution context of the function, the captured objects.
- Lambda expressions can be implemented as functors.

Simple Lambda Expression

```
[]()->void {std::cout << "Hello World";}
```

Simple Lambda Expression

```
[]()->void {std::cout << "Hello World";}
```

call:

```
[]()->void {std::cout << "Hello World";}();
```

Minimal Lambda Expression

```
[]{}
```

- Return type can be inferred if ≤ 1 return statement.[17]

```
[]() {std::cout << "Hello World";}
```

- If no parameters and no explcit return type, then () can be omitted.

```
[]{std::cout << "Hello World";}
```

- `[...]` can never be omitted.

[17] Since C++14 also several returns provided that the same return type is deduced

Examples

```
[](int x, int y) {std::cout << x * y;} (4,5);
```

Output:

Examples

```
[](int x, int y) {std::cout << x * y;} (4,5);
```

Output: 20

Examples

```
int k = 8;
[](int& v) {v += v;} (k);
std::cout << k;
```

Output:

Examples

```cpp
int k = 8;
[](int& v) {v += v;} (k);
std::cout << k;
```

Output: 16

Examples

```
int k = 8;
[](int v) {v += v;} (k);
std::cout << k;
```

Output:

Examples

```
int k = 8;
[](int v) {v += v;} (k);
std::cout << k;
```

Output: 8

Capture – Lambdas

For Lambda-expressions the capture list determines the context accessible

Syntax:

- `[x]`: Access a copy of x (read-only)
- `[&x]`: Capture x by reference
- `[&x,y]`: Capture x by reference and y by value
- `[&]`: Default capture all objects by reference in the scope of the lambda expression
- `[=]`: Default capture all objects by value in the context of the Lambda-Expression

Capture – Lambdas

```
int elements=0;
int sum=0;
std::for_each(v.begin(), v.end(),
  [&] (int k) {sum += k; elements++;} // capture all by reference
)
```

Capture – Lambdas

```
template <typename T>
void sequence(vector<int> & v, T done){
  int i=0;
  while (!done()) v.push_back(i++);
}

vector<int> s;
sequence(s, [&] {return s.size() >= 5;} )
```

now v =

Capture – Lambdas

```
template <typename T>
void sequence(vector<int> & v, T done){
  int i=0;
  while (!done()) v.push_back(i++);
}

vector<int> s;
sequence(s, [&] {return s.size() >= 5;} )
```

now v = 0 1 2 3 4

Capture – Lambdas

```
template <typename T>
void sequence(vector<int> & v, T done){
  int i=0;
  while (!done()) v.push_back(i++);
}

vector<int> s;
sequence(s, [&] {return s.size() >= 5;} )
```

now v = 0 1 2 3 4

The capture list refers to the context of the lambda expression.

Capture – Lambdas

When is the value captured?

```
int v = 42;
auto func = [=] {std::cout << v << "\n"};
v = 7;
func();
```

Output:

Capture – Lambdas

When is the value captured?

```
int v = 42;
auto func = [=] {std::cout << v << "\n"};
v = 7;
func();
```

Output: 42

Values are assigned when the lambda-expression is created.

Capture – Lambdas

(Why) does this work?

```
class Limited{
  int limit = 10;
 public:
  // count entries smaller than limit
  int count(const std::vector<int>& a){
    int c = 0;
    std::for_each(a.begin(), a.end(),
        [=,&c] (int x) {if (x < limit) c++;}
    );
    return c;
  }
};
```

Capture – Lambdas

(Why) does this work?

```
class Limited{
  int limit = 10;
 public:
  // count entries smaller than limit
  int count(const std::vector<int>& a){
    int c = 0;
    std::for_each(a.begin(), a.end(),
        [=,&c] (int x) {if (x < limit) c++;}
    );
    return c;
  }
};
```

The `this` pointer is implicitly copied by value

Capture – Lambdas

```
struct mutant{
  int i = 0;
  void do(){ [=] {i=42;}();}
};

mutant m;
m.do();
std::cout << m.i;
```

Output:

Capture – Lambdas

```
struct mutant{
  int i = 0;
  void do(){ [=] {i=42;}();}
};

mutant m;
m.do();
std::cout << m.i;
```

Output: 42

The `this` *pointer* is implicitly copied by value

Lambda Expressions are Functors

```
[x, &y] () {y = x;}
```

can be implemented as

```
unnamed {x,y};
```

with

```
class unnamed {
  int x; int& y;
  unnamed (int x_, int& y_) : x (x_), y (y_) {}
  void operator () () {y = x;}
};
```

Lambda Expressions are Functors

```
[=] () {return x + y;}
```

can be implemented as

```
unnamed {x,y};
```

with

```
class unnamed {
  int x; int y;
  unnamed (int x_, int y_) : x (x_), y (y_) {}
  int operator () () const {return x + y;}
};
```

Polymorphic Function Wrapper `std::function`

```
#include <functional>

int k= 8;
std::function<int(int)> f;
f = [k](int i){ return i+k; };
std::cout << f(8); // 16
```

Kann verwendet werden, um Lambda-Expressions zu speichern.

Other Examples

`std::function<int(int,int)>`;

`std::function<void(double)>` ...

http://en.cppreference.com/w/cpp/utility/functional/function

14. Hashing

Hash Tables, Birthday Paradoxon, Hash functions, Perfect and Universal Hashing, Resolving Collisions with Chaining, Open Addressing, Probing [Ottman/Widmayer, Kap. 4.1-4.3.2, 4.3.4, Cormen et al, Kap. 11-11.4]

Motivation

Gloal: Table of all n students of this course

Requirement: fast access by name

Naive Ideas

Mapping Name $s = s_1 s_2 \dots s_{l_s}$ to key

$$k(s) = \sum_{i=1}^{l_s} s_i \cdot b^i$$

b large enough such that different names map to different keys.

Naive Ideas

Mapping Name $s = s_1 s_2 \dots s_{l_s}$ to key

$$k(s) = \sum_{i=1}^{l_s} s_i \cdot b^i$$

b large enough such that different names map to different keys.

Store each data set at its index in a huge array.

Naive Ideas

Mapping Name $s = s_1 s_2 \dots s_{l_s}$ to key

$$k(s) = \sum_{i=1}^{l_s} s_i \cdot b^i$$

b large enough such that different names map to different keys.

Store each data set at its index in a huge array.

Example with $b = 100$. Ascii-Values s_i.

Anna $\mapsto 71111065$

Jacqueline $\mapsto 102110609021813999774$

Naive Ideas

Mapping Name $s = s_1 s_2 \dots s_{l_s}$ to key

$$k(s) = \sum_{i=1}^{l_s} s_i \cdot b^i$$

b large enough such that different names map to different keys.

Store each data set at its index in a huge array.

Example with $b = 100$. Ascii-Values s_i.

Anna $\mapsto 71111065$
Jacqueline $\mapsto 102110609021813999774$

Unrealistic: requires too large arrays.

Better idea?

Allocation of an array of size m ($m > n$).

Better idea?

Allocation of an array of size m ($m > n$).

Mapping Name s to

$$k_m(s) = \left(\sum_{i=1}^{l_s} s_i \cdot b^i\right) \bmod m.$$

Better idea?

Allocation of an array of size m ($m > n$).

Mapping Name s to

$$k_m(s) = \left(\sum_{i=1}^{l_s} s_i \cdot b^i\right) \bmod m.$$

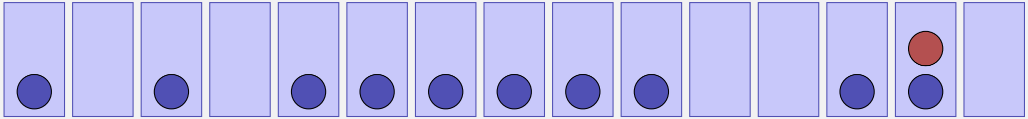
Different names can map to the same key (“Collision”). And then?

Estimation

Maybe collision do not really exist? We make an estimation ...

Abschätzung

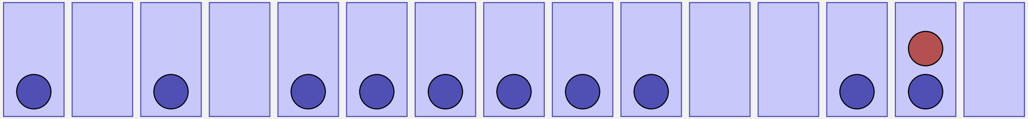
Assumption: m urns, n balls (wlog $n \leq m$).
n balls are put uniformly distributed into the urns



What is the collision probability?

Abschätzung

Assumption: m urns, n balls (wlog $n \leq m$).
n balls are put uniformly distributed into the urns



What is the collision probability?

Very similar question: with how many people (n) the probability that two of them share the same birthday ($m = 365$) is larger than 50%?

Estimation

$$\mathbb{P}(\text{no collision}) = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \ldots \cdot \frac{m-n+1}{m} = \frac{m!}{(m-n)! \cdot m^m}.$$

Estimation

$\mathbb{P}(\text{no collision}) = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \ldots \cdot \frac{m-n+1}{m} = \frac{m!}{(m-n)! \cdot m^m}$.

Let $a \ll m$. With $e^x = 1 + x + \frac{x^2}{2!} + \ldots$ approximate $1 - \frac{a}{m} \approx e^{-\frac{a}{m}}$.
This yields:

$$1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \ldots \cdot \left(1 - \frac{n-1}{m}\right) \approx e^{-\frac{1+\cdots+n-1}{m}} = e^{-\frac{n(n-1)}{2m}}.$$

Estimation

$\mathbb{P}(\text{no collision}) = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \frac{m!}{(m-n)! \cdot m^m}$.

Let $a \ll m$. With $e^x = 1 + x + \frac{x^2}{2!} + \dots$ approximate $1 - \frac{a}{m} \approx e^{-\frac{a}{m}}$.
This yields:

$$1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \ldots \cdot \left(1 - \frac{n-1}{m}\right) \approx e^{-\frac{1+\dots+n-1}{m}} = e^{-\frac{n(n-1)}{2m}}.$$

Thus

$$\mathbb{P}(\text{Kollision}) = 1 - e^{-\frac{n(n-1)}{2m}}.$$

Estimation

$\mathbb{P}(\text{no collision}) = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \frac{m!}{(m-n)! \cdot m^m}$.

Let $a \ll m$. With $e^x = 1 + x + \frac{x^2}{2!} + \dots$ approximate $1 - \frac{a}{m} \approx e^{-\frac{a}{m}}$.
This yields:

$$1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \ldots \cdot \left(1 - \frac{n-1}{m}\right) \approx e^{-\frac{1+\dots+n-1}{m}} = e^{-\frac{n(n-1)}{2m}}.$$

Thus

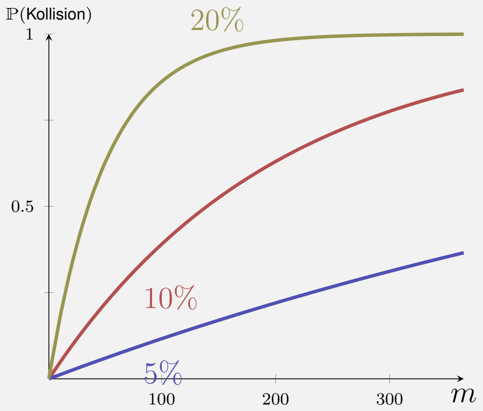
$$\mathbb{P}(\text{Kollision}) = 1 - e^{-\frac{n(n-1)}{2m}}.$$

Puzzle answer: with 23 people the probability for a birthday collision is 50.7%. Derived from the slightly more accurate Stirling formula.

With filling degree:

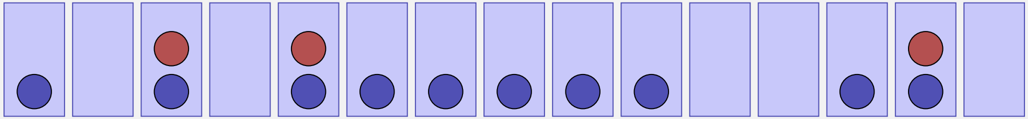
With filling degree $\alpha := n/m$ it holds that (simplified further)

$$\mathbb{P}(\text{collision}) \approx 1 - e^{-\alpha^2 \cdot \frac{m}{2}}.$$



Different Question

Assumption: m urns, n balls (wlog $n \leq m$).
n balls are put uniformly distributed into the urns



What is the expected number of collisions?

Expected Number Collisions

- $\mathbb{P}(\text{Kugel } B \text{ trifft Kugel } A_i) = 1/m$.
- $\mathbb{P}(\text{Kugel } B \text{ trifft Kugel } A_i \text{ nicht}) = 1 - 1/m$.
- $\mathbb{P}(n-1 \text{ Kugeln treffen } A_i \text{ nicht}) = (1 - 1/m)^{n-1}$.
- $\mathbb{P}(A_i \text{ getroffen}) = 1 - (1 - 1/m)^{n-1}$.
- Sei X_i Zufallsvariable mit $X_i = \mathbb{1}_{A_i \text{getroffen}}$
- $\mathbb{E}(\sum X_i) = \sum \mathbb{E}(X_i)$
- $\mathbb{E}(\text{Anzahl getroffene Kugeln}) = n(1 - (1 - 1/m)^n) \approx \frac{n^2}{2m}$.

Nomenclature

Hash funtion h: Mapping from the set of keys $\mathcal{K}$ to the index set $\{0, 1, \ldots, m-1\}$ of an array (*hash table*).

$$h : \mathcal{K} \rightarrow \{0, 1, \ldots, m-1\}.$$

Normally $|\mathcal{K}| \gg m$. There are $k_1, k_2 \in \mathcal{K}$ with $h(k_1) = h(k_2)$ (*collision*).

A hash function should map the set of keys as uniformly as possible to the hash table.

Examples of Good Hash Functions

- $h(k) = k \bmod m$, m prime
- $h(k) = \lfloor m(k \cdot r - \lfloor k \cdot r \rfloor) \rfloor$, r irrational, paritcularly good: $r = \frac{\sqrt{5}-1}{2}$.

Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$. Keys $12, 53, 15, 2, 19, 43$

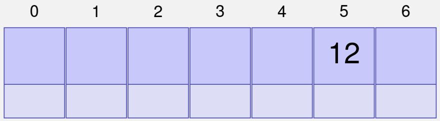
Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55

Chaining the Collisions

hash table



Colliding entries

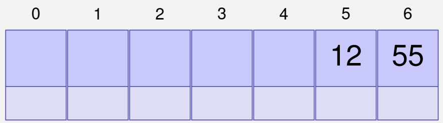
Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5

Chaining the Collisions

hash table



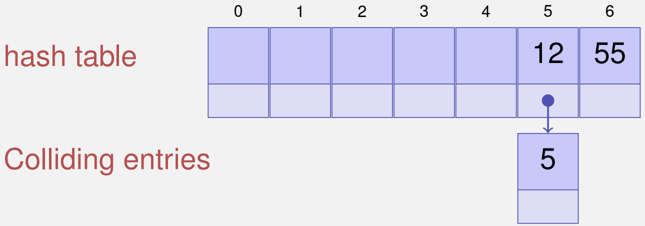
Colliding entries

Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5 , 15

Chaining the Collisions

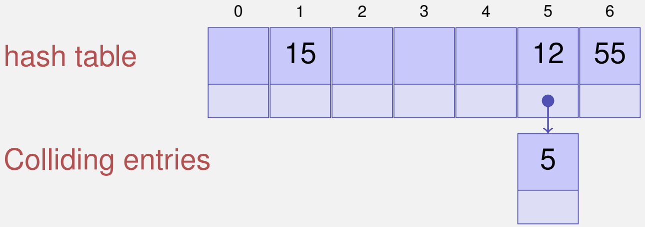


Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5 , 15 , 2

Chaining the Collisions

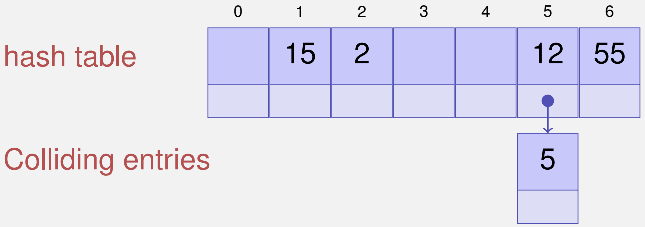


Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5 , 15 , 2 , 19

Chaining the Collisions

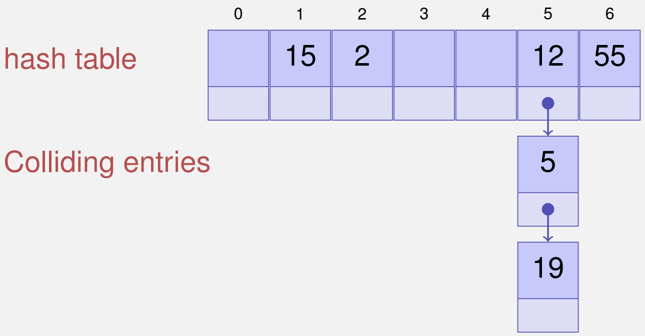


Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5 , 15 , 2 , 19 , 43

Chaining the Collisions



Resolving Collisions

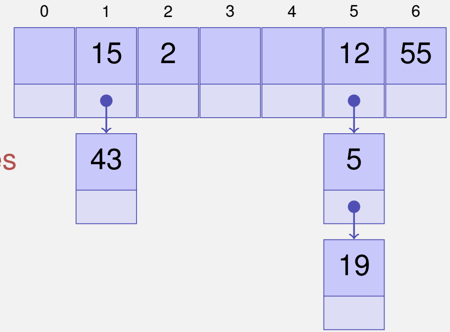
Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5 , 15 , 2 , 19 , 43

Chaining the Collisions

hash table

Colliding entries



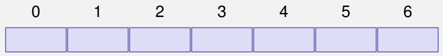
Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.

Keys 12

Direct Chaining of the Colliding entries

hash table



Colliding entries

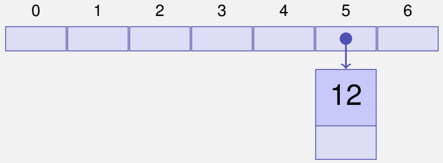
Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55

Direct Chaining of the Colliding entries

hash table



Colliding entries

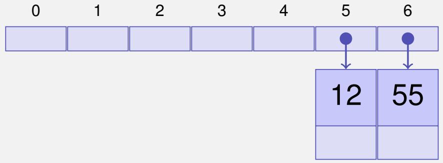
Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5

Direct Chaining of the Colliding entries

hash table



Colliding entries

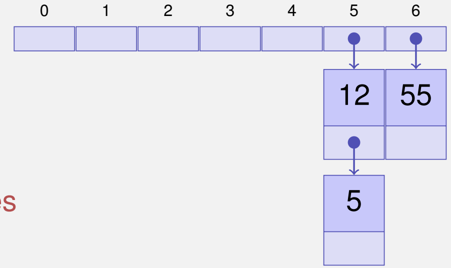
Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5 , 15

Direct Chaining of the Colliding entries

hash table



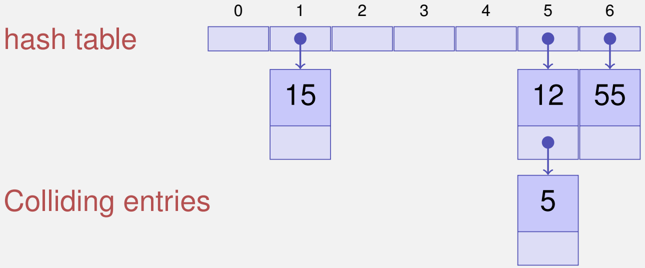
Colliding entries

Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5 , 15 , 2

Direct Chaining of the Colliding entries

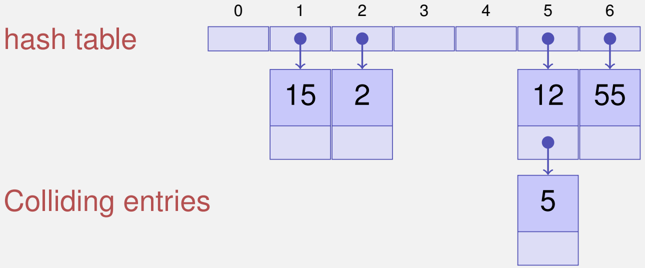


Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5 , 15 , 2 , 19

Direct Chaining of the Colliding entries

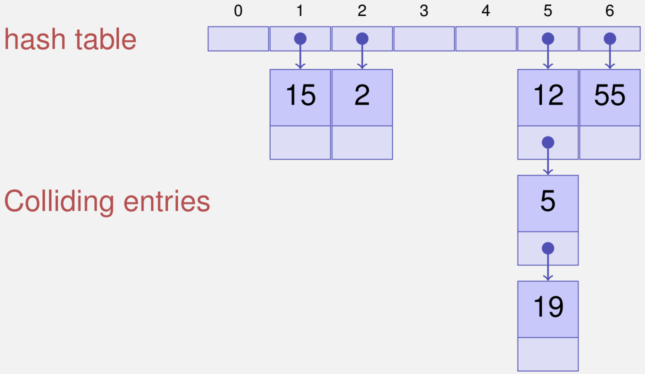


Resolving Collisions

Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5 , 15 , 2 , 19 , 43

Direct Chaining of the Colliding entries



Resolving Collisions

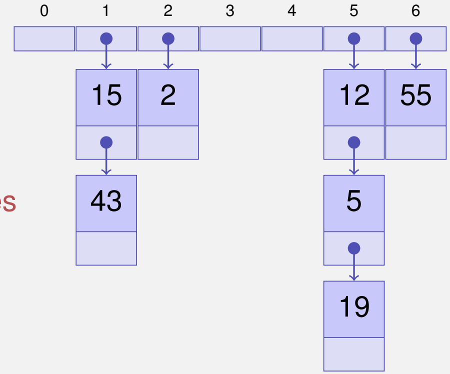
Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55 , 5 , 15 , 2 , 19 , 43

Direct Chaining of the Colliding entries

hash table

Colliding entries



Algorithm for Hashing with Chaining

- `contains`(k) Search in list from position $h(k)$ for k. Return true if found, otherwise false.
- `put`(k) Check if k is in list at position $h(k)$. If no, then append k to the end of the list. Otherwise error message.
- `get`(k) Check if k is in list at position $h(k)$. If yes, return the data associated to key k, otherwise error message.
- `remove`(k) Search the list at position $h(k)$ for k. If successful, remove the list element.

Analysis (directly chained list)

1. Unsuccesful search.

Analysis (directly chained list)

1. Unsuccesful search. The average list lenght is $\alpha = \frac{n}{m}$. The list has to be traversed completely.

Analysis (directly chained list)

1. Unsuccesful search. The average list lenght is $\alpha = \frac{n}{m}$. The list has to be traversed completely.
   $\Rightarrow$ Average number of entries considered

$$C'_n = \alpha.$$

Analysis (directly chained list)

1. Unsuccesful search. The average list lenght is $\alpha = \frac{n}{m}$. The list has to be traversed completely.
   $\Rightarrow$ Average number of entries considered
   $$C'_n = \alpha.$$

2. Successful search Consider the insertion history: key j sees an average list length of $(j-1)/m$.

Analysis (directly chained list)

1. Unsuccesful search. The average list lenght is $\alpha = \frac{n}{m}$. The list has to be traversed completely.
   $\Rightarrow$ Average number of entries considered
   $$C'_n = \alpha.$$

2. Successful search Consider the insertion history: key j sees an average list length of $(j-1)/m$.
   $\Rightarrow$ Average number of considered entries
   $$C_n = \frac{1}{n}\sum_{j=1}^{n}(1 + (j-1)/m)) \qquad .$$

Analysis (directly chained list)

1. Unsuccesful search. The average list lenght is $\alpha = \frac{n}{m}$. The list has to be traversed completely.
   $\Rightarrow$ Average number of entries considered
   $$C'_n = \alpha.$$

2. Successful search Consider the insertion history: key j sees an average list length of $(j-1)/m$.
   $\Rightarrow$ Average number of considered entries
   $$C_n = \frac{1}{n}\sum_{j=1}^{n}(1+(j-1)/m)) = 1 + \frac{1}{n}\frac{n(n-1)}{2m} \qquad .$$

Analysis (directly chained list)

1. Unsuccesful search. The average list lenght is $\alpha = \frac{n}{m}$. The list has to be traversed completely.
   $\Rightarrow$ Average number of entries considered

$$C'_n = \alpha.$$

2. Successful search Consider the insertion history: key j sees an average list length of $(j-1)/m$.
   $\Rightarrow$ Average number of considered entries

$$C_n = \frac{1}{n}\sum_{j=1}^{n}(1+(j-1)/m)) = 1 + \frac{1}{n}\frac{n(n-1)}{2m} \approx 1 + \frac{\alpha}{2}.$$

Advantages and Disadvantages

Advantages

- Possible to overcommit: $\alpha > 1$
- Easy to remove keys.

Disadvantages

- Memory consumption of the chains-

Open Addressing

Store the colliding entries directly in the hash table using a *probing function* $s(j, k)$ ($0 \leq j < m$, $k \in \mathcal{K}$)

Key table position along a *probing sequence*

$$S(k) := ((h(k) + s(0, k)) \bmod m, \ldots, (h(k) + s(m-1, k)) \bmod m)$$

Algorithms for open addressing

- `contains`(k) Traverse table entries according to $S(k)$. If k is found, return true. If the probing sequence is finished or an empty position is reached, return false.
- `put`(k) Search for k in the table according to $S(k)$. If k is not present, insert k at the first free position in the probing sequence. Otherwise error message.
- `get`(k) Traverse table entries according to $S(k)$. If k is found, return data associated to k. Otherwise error message.
- `remove`(k) Search k in the table according to $S(k)$. If k is found, replace it with a special `removed` key.

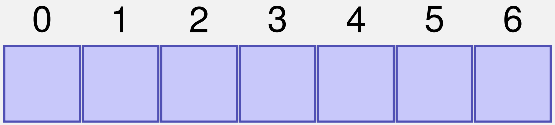
Linear Probing

$s(j,k) = j \Rightarrow$
$S(k) = (h(k) \bmod m, (h(k)+1) \bmod m, \ldots, (h(k)-1) \bmod m)$

Linear Probing

$s(j,k) = j \Rightarrow$
$S(k) = (h(k) \bmod m, (h(k)+1) \bmod m, \ldots, (h(k)-1) \bmod m)$

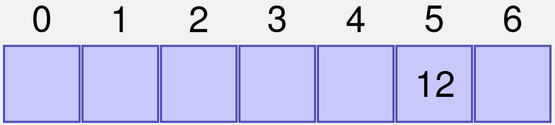
Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.
Key 12



Linear Probing

$s(j,k) = j \Rightarrow$
$S(k) = (h(k) \bmod m, (h(k)+1) \bmod m, \ldots, (h(k)-1) \bmod m)$

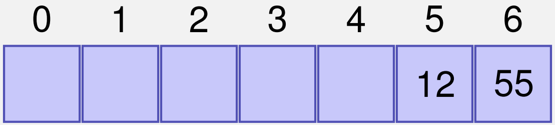
Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.
Key 12 , 55



Linear Probing

$s(j,k) = j \Rightarrow$
$S(k) = (h(k) \bmod m, (h(k)+1) \bmod m, \ldots, (h(k)-1) \bmod m)$

Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.
Key 12 , 55 , 5

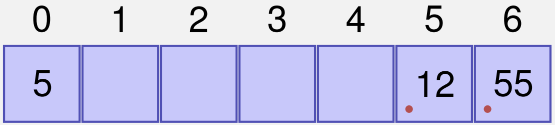


Linear Probing

$$s(j,k) = j \Rightarrow$$
$$S(k) = (h(k) \bmod m, (h(k)+1) \bmod m, \ldots, (h(k)-1) \bmod m)$$

Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.

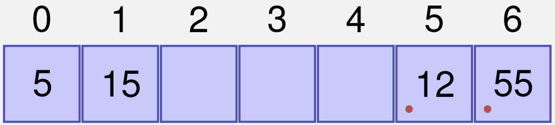
Key 12 , 55 , 5 , 15



Linear Probing

$s(j,k) = j \Rightarrow$
$S(k) = (h(k) \bmod m, (h(k)+1) \bmod m, \ldots, (h(k)-1) \bmod m)$

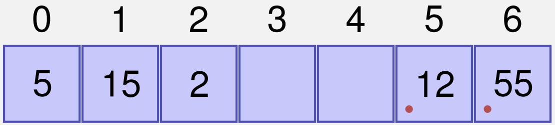
Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.
Key 12 , 55 , 5 , 15 , 2



Linear Probing

$s(j,k) = j \Rightarrow$
$S(k) = (h(k) \bmod m, (h(k)+1) \bmod m, \ldots, (h(k)-1) \bmod m)$

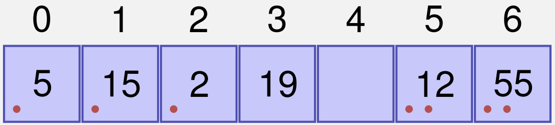
Example $m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod m$.
Key 12 , 55 , 5 , 15 , 2 , 19



Linear Probing

$s(j,k) = j \Rightarrow$
$S(k) = (h(k) \bmod m, (h(k)+1) \bmod m, \dots, (h(k)-1) \bmod m)$

Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.
Key 12 , 55 , 5 , 15 , 2 , 19



Analysis linear probing (without proof)

Analysis linear probing (without proof)

1. Unsuccessful search. Average number of considered entries

$$C'_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Analysis linear probing (without proof)

1. Unsuccessful search. Average number of considered entries

$$C'_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

2. Successful search. Average number of considered entries

$$C_n \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right).$$

Discussion

Discussion

Example $\alpha = 0.95$

The unsuccessful search consideres 200 table entries on average!

Discussion

Example $\alpha = 0.95$

The unsuccessful search consideres 200 table entries on average!

? Disadvantage of the method?

Discussion

Example $\alpha = 0.95$

The unsuccessful search consideres 200 table entries on average!

? Disadvantage of the method?

! *Primary clustering:* similar hash addresses have similar probing sequences $\Rightarrow$ long contiguous areas of used entries.

Quadratic Probing

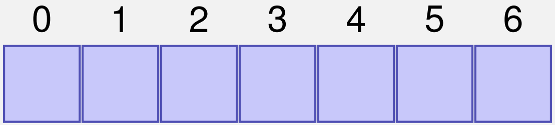
$$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$$

Quadratic Probing

$$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$$

Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.

Keys 12

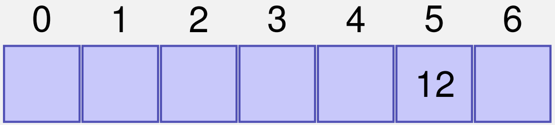


Quadratic Probing

$$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$$

Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.

Keys 12 , 55

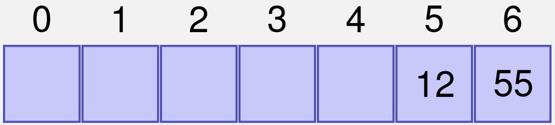


Quadratic Probing

$$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$$

Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.

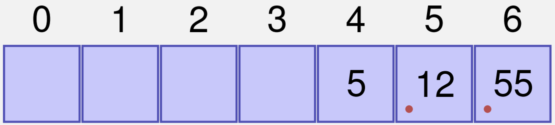
Keys 12 , 55 , 5



Quadratic Probing

$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$
$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$

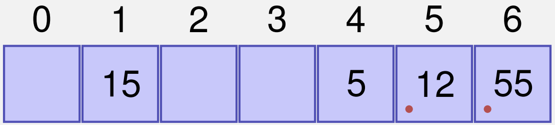
Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.
Keys 12 , 55 , 5 , 15



Quadratic Probing

$$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$$

Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \mod m$.
Keys 12 , 55 , 5 , 15 , 2



Quadratic Probing

$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$
$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$

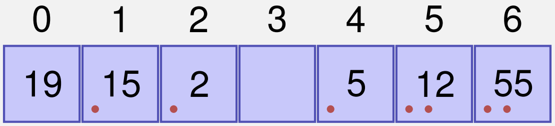
Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \mod m$.
Keys 12 , 55 , 5 , 15 , 2 , 19

0	1	2	3	4	5	6
	15	2		5	● 12	● 55

Quadratic Probing

$$s(j,k) = \lceil j/2 \rceil^2 (-1)^{j+1}$$
$$S(k) = (h(k), h(k)+1, h(k)-1, h(k)+4, h(k)-4, \dots) \mod m$$

Example $m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod m$.
Keys 12 , 55 , 5 , 15 , 2 , 19



Analysis Quadratic Probing (without Proof)

Analysis Quadratic Probing (without Proof)

1. Unsuccessful search. Average number of entries considered

$$C'_n \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$$

Analysis Quadratic Probing (without Proof)

1. Unsuccessful search. Average number of entries considered

$$C'_n \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$$

2. Successful search. Average number of entries considered

$$C_n \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}.$$

Discussion

Example $\alpha = 0.95$

Unsuccessfuly search considers 22 entries on average

Discussion

Example $\alpha = 0.95$

Unsuccessfuly search considers 22 entries on average

? Problems of this method?

Discussion

Example $\alpha = 0.95$

Unsuccessfuly search considers 22 entries on average

? Problems of this method?

! *Secondary clustering:* Synonyms k and k' (with $h(k) = h(k')$) travers the same probing sequence.

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m-1)h'(k)) \mod m$

Double Hashing

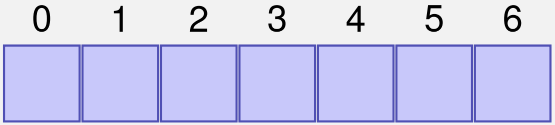
Two hash functions $h(k)$ and $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$S(k) = (h(k), h(k)+h'(k), h(k)+2h'(k), \dots, h(k)+(m-1)h'(k)) \mod m$

Example:

$m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.

Keys 12



Double Hashing

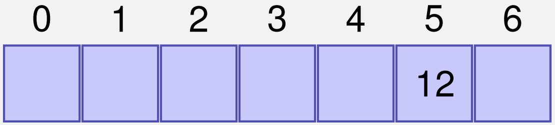
Two hash functions $h(k)$ and $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$$S(k) = (h(k), h(k)+h'(k), h(k)+2h'(k), \ldots, h(k)+(m-1)h'(k)) \mod m$$

Example:

$m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.

Keys 12 , 55



Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m-1)h'(k)) \mod m$$

Example:

$m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.

Keys 12 , 55 , 5

0	1	2	3	4	5	6
					12	55

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$$S(k) = (h(k), h(k)+h'(k), h(k)+2h'(k), \ldots, h(k)+(m-1)h'(k)) \mod m$$

Example:

$m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.

Keys 12 , 55 , 5 , 15

0	1	2	3	4	5	6
5					12	55

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$$S(k) = (h(k), h(k)+h'(k), h(k)+2h'(k), \ldots, h(k)+(m-1)h'(k)) \mod m$$

Example:

$m = 7$, $\mathcal{K} = \{0, \ldots, 500\}$, $h(k) = k \mod 7$, $h'(k) = 1 + k \mod 5$.

Keys 12 , 55 , 5 , 15 , 2

0	1	2	3	4	5	6
5	15				● 12	● 55

Double Hashing

Two hash functions $h(k)$ and $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m-1)h'(k)) \mod m$$

Example:

$m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \bmod 7$, $h'(k) = 1 + k \bmod 5$.

Keys 12 , 55 , 5 , 15 , 2 , 19

0	1	2	3	4	5	6
5	15	2			● 12	● 55

Double Hashing

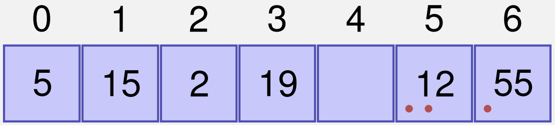
Two hash functions $h(k)$ and $h'(k)$. $s(j,k) = j \cdot h'(k)$.

$$S(k) = (h(k), h(k) + h'(k), h(k) + 2h'(k), \dots, h(k) + (m-1)h'(k)) \mod m$$

Example:

$m = 7$, $\mathcal{K} = \{0, \dots, 500\}$, $h(k) = k \mod 7$, $h'(k) = 1 + k \mod 5$.

Keys 12 , 55 , 5 , 15 , 2 , 19



Double Hashing

- Probing sequence must permute all hash addresses. Thus $h'(k) \neq 0$ and $h'(k)$ may not divide m, for example guaranteed with m prime.
- h' should be independent of h (avoiding secondary clustering)

Independence:

$$\mathbb{P}\left((h(k) = h(k')) \;\wedge\; (h'(k) = h'(k'))\right) = \mathbb{P}\left(h(k) = h(k')\right) \cdot \mathbb{P}\left(h'(k) = h'(k')\right).$$

Independence fulfilled by $h(k) = k \bmod m$ and $h'(k) = 1 + k \bmod (m-2)$ (m prime).

Analysis Double Hashing

Let h and h' be independent, then:

Analysis Double Hashing

Let h and h' be independent, then:

1. Unsuccessful search. Average number of considered entries:

$$C'_n \approx \frac{1}{1-\alpha}$$

Analysis Double Hashing

Let h and h' be independent, then:

1. Unsuccessful search. Average number of considered entries:

$$C'_n \approx \frac{1}{1-\alpha}$$

2. Successful search. Average number of considered entries:

$$C_n \approx \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

Overview

	$\alpha = 0.50$		$\alpha = 0.90$		$\alpha = 0.95$	
	C_n	C'_n	C_n	C'_n	C_n	C'_n
Separate Chaining	1.250	1.110	1.450	1.307	1.475	1.337
Direct Chaining	1.250	0.500	1.450	0.900	1.475	0.950
Linear Probing	1.500	2.500	5.500	50.500	10.500	200.500
Quadratic Probing	1.440	2.190	2.850	11.400	3.520	22.050
Double Hashing	1.39	2.000	2.560	10.000	3.150	20.000

: C_n: Anzahl Schritte erfolgreiche Suche, C'_n: Anzahl Schritte erfolglose Suche, Belegungsgrad α.

Perfect Hashing

If the set of used keys is known up-front the hash function can be chosen perfectly, i.e. such that there are no collisions. The practical construction is non-trivial.

Example: table of key words of a compiler.

Universal Hashing

- $|\mathcal{K}| > m \Rightarrow$ Set of “similar keys” can be chose such that a large number of collisions occur.
- Impossible to select a “best” hash function for all cases.
- Possible, however[18]: randomize!

Universal hash class $\mathcal{H} \subseteq \{h : \mathcal{K} \to \{0, 1, \ldots, m-1\}\}$ is a family of hash functions such that

$$\forall k_1 \neq k_2 \in \mathcal{K} : |\{h \in \mathcal{H} | h(k_1) = h(k_2)\}| \leq \frac{1}{m}|\mathcal{H}|.$$

[18] Similar as for quicksort

Universal Hashing

Theorem

A function h randomly chosen from a universal class $\mathcal{H}$ of hash functions randomly distributes an arbitrary sequence of keys from $\mathcal{K}$ as uniformly as possible on the available slots.

Universal Hashing

Initial remark for the proof of the theorem:

Define with $x, y \in \mathcal{K}$, $h \in \mathcal{H}$, $Y \subseteq \mathcal{K}$:

$$\delta(x,y,h) = \begin{cases} 1, & \text{if } h(x) = h(y), x \neq y \\ 0, & \text{otherwise,} \end{cases}$$

$$\delta(x,Y,h) = \sum_{y \in Y} \delta(x,y,h),$$

$$\delta(x,y,\mathcal{H}) = \sum_{h \in \mathcal{H}} \delta(x,y,h).$$

$\mathcal{H}$ is universal if for all $x, y \in \mathcal{K}$, $x \neq y$: $\delta(x,y,\mathcal{H}) \leq |\mathcal{H}|/m.$

Universal Hashing

Proof of the theorem

$S \subseteq \mathcal{K}$: keys stored up to now. x is added now:

$$\begin{aligned}
\mathbb{E}_{\mathcal{H}}(\delta(x,S,h)) &= \sum_{h\in\mathcal{H}} \delta(x,S,h)/|\mathcal{H}| \\
&= \frac{1}{|\mathcal{H}|}\sum_{h\in\mathcal{H}}\sum_{y\in S}\delta(x,y,h) = \frac{1}{|\mathcal{H}|}\sum_{y\in S}\sum_{h\in\mathcal{H}}\delta(x,y,h) \\
&= \frac{1}{|\mathcal{H}|}\sum_{y\in S}\delta(x,y,\mathcal{H}) \\
&\le \frac{1}{|\mathcal{H}|}\sum_{y\in S}|\mathcal{H}|/m = \frac{|S|}{m}.
\end{aligned}$$

■

Universal Hashing is Relevant!

Let p be prime and $\mathcal{K} = \{0, \ldots, p-1\}$. With $a \in \mathcal{K} \setminus \{0\}$, $b \in \mathcal{K}$ define

$$h_{ab} : \mathcal{K} \to \{0, \ldots, m-1\}, h_{ab}(x) = ((ax + b) \bmod p) \bmod m.$$

Then the following theorem holds:

Theorem

The class $\mathcal{H} = \{h_{ab} | a, b \in \mathcal{K}, a \neq 0\}$ is a universal class of hash functions.