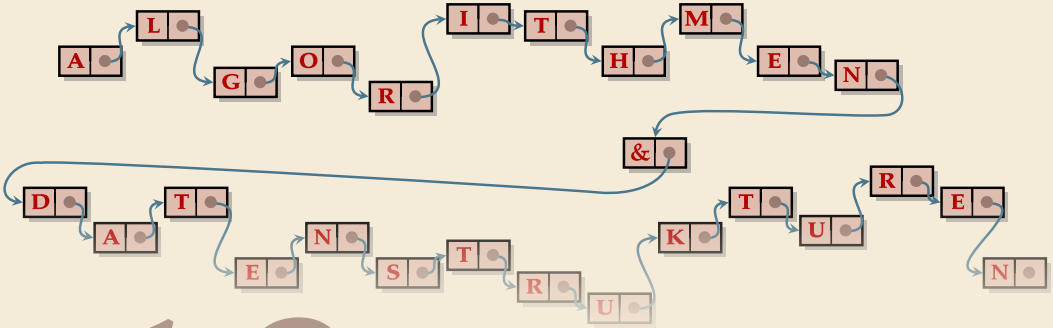


10 Hashing

Algorithmen & Datenstrukturen · Sommersemester 2026

Prof. Dr. Sebastian Wild

CS210 (Sommersemester 2026)
Philipps-Universität Marburg
version `2026-06-21 17:00`

Outline

10 Hashing

Unsortierte Symbol Tables

- balancierte binäre Suchbäume lösen alle Ordered-Symbol-Table-Operationen in $O(\log n)$ Zeit
- man kann beweisen: Für *Vergleichs-basierte* Symbol Tables ist das optimal!

Unsortierte Symbol Tables

- balancierte binäre Suchbäume lösen alle Ordered-Symbol-Table-Operationen in $O(\log n)$ Zeit
- man kann beweisen: Für *Vergleichs-basierte* Symbol Tables ist das optimal!
- *Was geht denn, wenn wir mehr als Vergleiche haben?*
  - . . . alles, was das word-RAM-Modell hergibt

Unsortierte Symbol Tables

- balancierte binäre Suchbäume lösen alle Ordered-Symbol-Table-Operationen in $O(\log n)$ Zeit
- man kann beweisen: Für *Vergleichs-basierte* Symbol Tables ist das optimal!
- *Was geht denn, wenn wir mehr als Vergleiche haben?*
  - . . . alles, was das word-RAM-Modell hergibt
    - Arithmetische Operationen in $O(1)$ Zeit für w-Bit-Zahlen
    - Bit-weise Operationen (Bit-weises Und, Oder, XOR; Bit-Masken . . .)

Unsortierte Symbol Tables

- balancierte binäre Suchbäume lösen alle Ordered-Symbol-Table-Operationen in $O(\log n)$ Zeit
- man kann beweisen: Für *Vergleichs-basierte* Symbol Tables ist das optimal!
- *Was geht denn, wenn wir mehr als Vergleiche haben?*
  - . . . alles, was das word-RAM-Modell hergibt
    - Arithmetische Operationen in $O(1)$ Zeit für w-Bit-Zahlen
    - Bit-weise Operationen (Bit-weises Und, Oder, XOR; Bit-Masken . . .)
    - **indirekte Adressierung / Rechnen mit Pointern** $R_i := \texttt{MEM}[R_j]$
  - Und warum soll das helfen?
  - $\rightsquigarrow$ volle Antwort in *Advanced Data Structures*

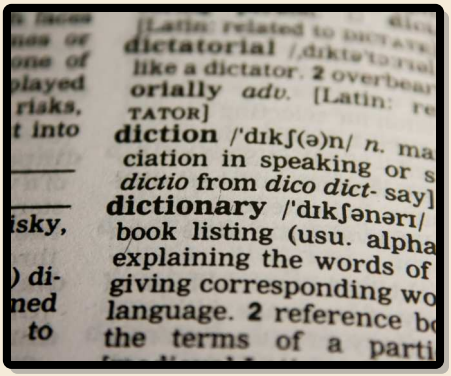
Unsortierte Symbol Tables

- balancierte binäre Suchbäume lösen alle Ordered-Symbol-Table-Operationen in $O(\log n)$ Zeit
- man kann beweisen: Für *Vergleichs-basierte* Symbol Tables ist das optimal!
- *Was geht denn, wenn wir mehr als Vergleiche haben?*
  - . . . alles, was das word-RAM-Modell hergibt
    - Arithmetische Operationen in $O(1)$ Zeit für w-Bit-Zahlen
    - Bit-weise Operationen (Bit-weises Und, Oder, XOR; Bit-Masken . . .)
    - **indirekte Adressierung / Rechnen mit Pointern** $R_i := \texttt{MEM}[R_j]$
  - Und warum soll das helfen?
  - $\rightsquigarrow$ volle Antwort in *Advanced Data Structures*
- **hier:** Was, wenn ich nur `get` und `put` möchte?

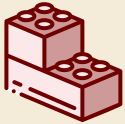
  (Manche Anwendungen verwenden die Ordered-Symbol-Table-Operationen gar nicht)

Recap: Symbol Table ADT

Symbol table (ST) / Dictionary / Map / Assoziatives Array / key-value store:



- `put(`k`,`v`)` *Python* dict: `d[`k`] = `v
  Füge key-value mapping (k, v) zur ST hinzu
- `get(`k`)` *Python* dict: `d[`k`]`
  Gib Wert für key k zurück (falls vorhanden)
- `delete(`k`)` *Python* dict: `del d[`k`]`
  Entferne key k (und assoziierten Wert) aus ST
- `contains(`k`)` *Python* dict: k `in d`
  Ist key k in der ST?
- `isEmpty(),size()`
- `create()`



Der wahrscheinlich wichtigste Baustein der Informatik!

(Jede Programmiersprache liefert eine ST mit.)

Warm-Up

Wie würden Sie eine Symbol Table für natürliche Zahlen speichern?

Warm-Up

Wie würden Sie eine Symbol Table für natürliche Zahlen speichern?

- **Ziel:** Unterstütze `put`, `get`, `delete`, `contains` für Schlüssel aus $U = [0..u)$
  - $\rightsquigarrow$ speichern i.W. eine dynamische Menge $S \subseteq U$
- **Annahme:** Universum nicht sehr groß, $u = |U|$

Warm-Up

Wie würden Sie eine Symbol Table für natürliche Zahlen speichern?

- **Ziel:** Unterstütze `put`, `get`, `delete`, `contains` für Schlüssel aus $U = [0..u)$
  - $\rightsquigarrow$ speichern i.W. eine dynamische Menge $S \subseteq U$
- **Annahme:** Universum nicht sehr groß, $u = |U|$
- 💡 **Idee**: Verwenden Schlüssel $k \in U$ als **Index** in Array

Warm-Up

Wie würden Sie eine Symbol Table für natürliche Zahlen speichern?

- **Ziel:** Unterstütze `put`, `get`, `delete`, `contains` für Schlüssel aus $U = [0..u)$
  - ⇝ speichern i.W. eine dynamische Menge $S \subseteq U$
- **Annahme:** Universum nicht sehr groß, $u = |U|$
- 💡 **Idee**: Verwenden Schlüssel $k \in U$ als **Index** in Array
- </> **Code**:

```
1 public class IntST<Value> {
2     Object[] values;
3     public IntST(int u) { values = new Object[u]; }
4     public void put(int key, Value val) { values[key] = val; }
5     @SuppressWarnings("unchecked")
6     public Value get(int key) { return (Value) values[key]; }
7     public void delete(int key) { values[key] = null; }
8     public boolean contains(int key) { return values[key] != null; }
9 }
```

⇝ 🏔 **Analyse:** $O(1)$ für alle Operationen!!

Warm-Up

Wie würden Sie eine Symbol Table für natürliche Zahlen speichern?

- **Ziel:** Unterstütze `put`, `get`, `delete`, `contains` für Schlüssel aus $U = [0..u)$
  - $\rightsquigarrow$ speichern i.W. eine dynamische Menge $S \subseteq U$
- **Annahme:** Universum nicht sehr groß, $u = |U|$
- 💡 **Idee**: Verwenden Schlüssel $k \in U$ als **Index** in Array
- **</> Code**:

```
1 public class IntST<Value> {
2     Object[] values;
3     public IntST(int u) { values = new Object[u]; }
4     public void put(int key, Value val) { values[key] = val; }
5     @SuppressWarnings("unchecked")
6     public Value get(int key) { return (Value) values[key]; }
7     public void delete(int key) { values[key] = null; }
8     public boolean contains(int key) { return values[key] != null; }
9 }
```

$\rightsquigarrow$ 🏔 **Analyse:** $O(1)$ für alle Operationen!!
Aber auch $\Theta(u)$ Speicher; potentiell $u \gg n$

Hashing

Hashing: *Alles ist eine natürliche Zahl!*

- . . . oder lässt sich zu einer machen!

⇝ **Teil 1:** Wie kommen wir von beliebigen Objekten zu Zahlen?

Hashing

Hashing: *Alles ist eine natürliche Zahl!*

- ... oder lässt sich zu einer machen!

⇝ **Teil 1:** Wie kommen wir von beliebigen Objekten zu Zahlen?

Problem dabei: U wäre dann die Menge aller *möglichen* Objekte!

- Müssten einen Speicherplatz für jede denkbare Instanz einer Klasse vorhalten

⇝ **Teil 2:** Wie damit umgehen?

  - Müssen mit Hash-Kollisionen leben ...

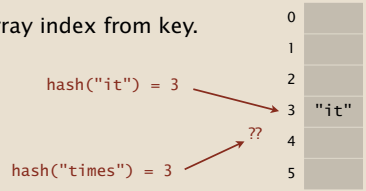
- Idealerweise: Speicherplatz $O(n)$, nicht $O(u)$!

10.1 Hash-Funktionen

Hashing: basic plan

Save items in a key-indexed table (index is a function of the key).

Hash function. Method for computing array index from key.



Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.

Classic space-time tradeoff.

- No space limitation: trivial hash function with key as index.
- No time limitation: trivial collision resolution with sequential search.
- Space and time limitations: hashing (the real world).

Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications

Ex 1. Phone numbers.

- Bad: first three digits.
- Better: last three digits.

Ex 2. Social Security numbers.

- Bad: first three digits.
- Better: last three digits.

573 = California, 574 = Alaska
(assigned in chronological order within geographic region)

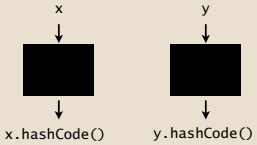
Practical challenge. Need different approach for each key type.

Java's hash code conventions

All Java classes inherit a method `hashCode()`, which returns a 32-bit `int`.

Requirement. If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`.

Highly desirable. If `!x.equals(y)`, then `(x.hashCode() != y.hashCode())`.



Default implementation. Memory address of `x`.

Legal (but poor) implementation. Always return `17`.

Customized implementations. `Integer`, `Double`, `String`, `File`, `URL`, `Date`, ...

User-defined types. Users are on their own.

Implementing hash code: integers, booleans, and doubles

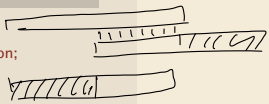
Java library implementations

```
public final class Integer
{
   private final int value;
   ...

   public int hashCode()
   {  return value;  }
}
```

```
public final class Boolean
{
   private final boolean value;
   ...

   public int hashCode()
   {
      if (value) return 1231;
      else       return 1237;
   }
}
```

```
public final class Double
{
   private final double value;
   ...

   public int hashCode()
   {
      long bits = doubleToLongBits(value);
      return (int) (bits ^ (bits >>> 32));
   }
}
```

convert to IEEE 64-bit representation;
xor most significant 32-bits
with least significant 32-bits



Warning: -0.0 and +0.0 have different hash codes

Implementing hash code: strings

Java library implementation

```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

i[th] character of s

char	Unicode
...	...
'a'	97
'b'	98
'c'	99
...	...

- Horner's method to hash string of length L: L multiplies/adds.
- Equivalent to $h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$.

Ex.

```
String s = "call";
int code = s.hashCode();
```

3045982 = 99·31[3] + 97·31[2] + 108·31[1] + 108·31[0]
= 108 + 31· (108 + 31 · (97 + 31 · (99)))
(Horner's method)

Implementing hash code: strings

Performance optimization.

- Cache the hash value in an instance variable.
- Return cached value.

```
public final class String
{
   private int hash = 0;                    <-- cache of hash code
   private final char[] s;
   ...

   public int hashCode()
   {
      int h = hash;                         <-- return cached value
      if (h != 0) return h;
      for (int i = 0; i < length(); i++)
         h = s[i] + (31 * h);
      hash = h;                             <-- store cache of hash code
      return h;
   }
}
```

Q. What if `hashCode()` of string is 0?

Implementing hash code: user-defined types

```
public final class Transaction implements Comparable<Transaction>
{
   private final String  who;
   private final Date    when;
   private final double  amount;

   public Transaction(String who, Date when, double amount)
   {  /* as before */  }

   ...

   public boolean equals(Object y)
   {  /* as before */  }

   public int hashCode()
   {
      int hash = 17;
      hash = 31*hash + who.hashCode();
      hash = 31*hash + when.hashCode();
      hash = 31*hash + ((Double) amount).hashCode();
      return hash;
   }
}
```

nonzero constant

typically a small prime

for reference types, use hashCode()

for primitive types, use hashCode() of wrapper type

Hash code design

"Standard" recipe for user-defined types.

- Combine each significant field using the $31x + y$ rule.
- If field is a primitive type, use wrapper type `hashCode()`.
- If field is `null`, return 0.
- If field is a reference type, use `hashCode()`. ← applies rule recursively
- If field is an array, apply to each entry. ← or use `Arrays.deepHashCode()`

In practice. Recipe works reasonably well; used in Java libraries.

In theory. Keys are bitstring; "universal" hash functions exist.

Basic rule. Need to use the whole key to compute hash code; consult an expert for state-of-the-art hash codes.

`Objects.hash`

Weil am Ende die meisten Implementierungen von `hashCode` das gleiche Rezept verwendet haben, gibt es in Java jetzt eine Abkürzung dafür

```
1  import java.util.Objects;
2
3  class Transaction {
4      private final String who;
5      private final Date when;
6      private final double amount;
7      // ... equals etc. wie gehabt
8      public int hashCode() {
9          Objects.hash(who, when, amount);
10     }
11 }
```

Objects.hash

Weil am Ende die meisten Implementierungen von `hashCode` das gleiche Rezept verwendet haben, gibt es in Java jetzt eine Abkürzung dafür

```java
import java.util.Objects;

class Transaction {
    private final String who;
    private final Date when;
    private final double amount;
    // ... equals etc. wie gehabt
    public int hashCode() {
        Objects.hash(who, when, amount);
    }
}
```

```java
package java.util;

class Objects { // varargs
    public static int hash(Object... a) {
        // Code äquivalent zu diesem
        if (a == null) { return 0; }
        int res = 1;
        for (int i = 0; i < a.length; i++)
            res = 31 * res + Objects.hashCode(a[i]);
        return res;
    }
}
```

Objects.hash

Weil am Ende die meisten Implementierungen von `hashCode` das gleiche Rezept verwendet haben, gibt es in Java jetzt eine Abkürzung dafür

```java
import java.util.Objects;

class Transaction {
    private final String who;
    private final Date when;
    private final double amount;
    // ... equals etc. wie gehabt
    public int hashCode() {
        Objects.hash(who, when, amount);
    }
}
```

```java
package java.util;

class Objects { // varargs
    public static int hash(Object... a) {
        // Code äquivalent zu diesem
        if (a == null) { return 0; }
        int res = 1;
        for (int i = 0; i < a.length; i++)
            res = 31 * res + Objects.hashCode(a[i]);
        return res;
    }
}
```

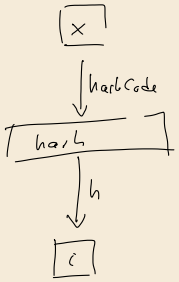
Deutlich lesbarer im Client-Code $\rightsquigarrow$ stets verwenden

10.2 Vom Hash zum Array-Index

hashCode vs. Index

- Speichern N Schlüssel / Key-Value-Paare in unserer Symbol Table
- Verwenden Array $S[0..M)$ mit M "Schubladen/Slots"
- x`.hashCode()` liefert uns einen `int`-Wert $\in [-2^{31}..2^{31})$
- Suchen Hash-Funktion $h : [-2^{31}..2^{31}) \to [0..M)$!

$\rightsquigarrow$ Weisen Objekt x den Slot $h(x$`.hashCode()`$)$ zu



S[i]

Modular hashing

Hash code. An `int` between -2^{31} and $2^{31} - 1$.

Hash function. An `int` between `0` and `M - 1` (for use as array index).

typically a prime or power of 2

mathematisch korrekt

```
private int hash(Key key)
{  return key.hashCode() % M;  }
```

bug

$abs(-2^{31}) = -2^{31}$


```
private int hash(Key key)
{  return Math.abs(key.hashCode()) % M;  }
```

1-in-a-billion bug

hashCode() of "polygenelubricants" is -2^{31}

```
private int hash(Key key)
{  return (key.hashCode() & 0x7fffffff) % M;  }
```

correct

Vorzeichen löschen

10.3 Separate Chaining

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$.

Bins and balls. Throw balls uniformly at random into M bins.

Birthday problem. Expect two balls in the same bin after $\sim \sqrt{\pi M / 2}$ tosses.

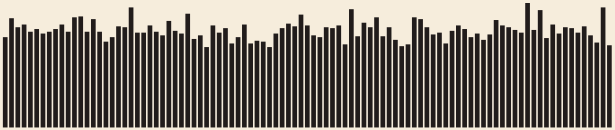
Coupon collector. Expect every bin has ≥ 1 ball after $\sim M \ln M$ tosses.

Load balancing. After M tosses, expect most loaded bin has $\Theta (\log M / \log \log M)$ balls.

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$.

Bins and balls. Throw balls uniformly at random into M bins.



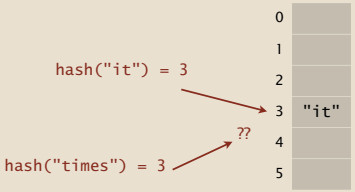
Hash value frequencies for words in Tale of Two Cities (M = 97)

Java's `String` data uniformly distribute the keys of Tale of Two Cities

Collisions

Collision. Two distinct keys hashing to same index.

- Birthday problem $\Rightarrow$ can't avoid collisions unless you have a ridiculous (quadratic) amount of memory.
- Coupon collector + load balancing $\Rightarrow$ collisions are evenly distributed.



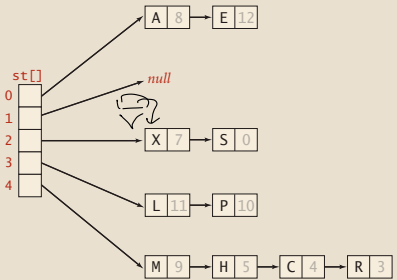
Challenge. Deal with collisions efficiently.

Separate-chaining symbol table

Use an array of $M < N$ linked lists. [H. P. Luhn, IBM 1953]

- Hash: map key to integer i between 0 and $M - 1$.
- Insert: put at front of i^{th} chain (if not already there).
- Search: need to search only i^{th} chain.

key	hash	value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10
L	3	11
E	0	12



Separate-chaining symbol table: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
   private int M = 97;                // number of chains
   private Node[] st = new Node[M];   // array of chains

   private static class Node
   {
      private Object key;
      private Object val;
      private Node next;
      ...
   }

   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M;  }

   public Value get(Key key) {
      int i = hash(key);
      for (Node x = st[i]; x != null; x = x.next)
         if (key.equals(x.key)) return (Value) x.val;
      return null;
   }

}
```

array doubling and halving code omitted

no generic array creation (declare key and value of type Object)

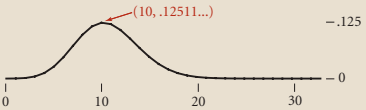
Separate-chaining symbol table: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
   private int M = 97;                // number of chains
   private Node[] st = new Node[M];  // array of chains

   private static class Node
   {
      private Object key;
      private Object val;
      private Node next;
      ...
   }

   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M;  }

   public void put(Key key, Value val) {
      int i = hash(key);
      for (Node x = st[i]; x != null; x = x.next)
         if (key.equals(x.key)) { x.val = val; return; }
      st[i] = new Node(key, val, st[i]);
   }

}
```

Analysis of separate chaining

Proposition. Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of N / M is extremely close to 1.

Pf sketch. Distribution of list size obeys a binomial distribution.



Binomial distribution ($N = 10^4$, $M = 10^3$, $\alpha = 10$)

equals() and hashCode()

Consequence. Number of probes for search/insert is proportional to N / M.

- M too large $\Rightarrow$ too many empty chains.
- M too small $\Rightarrow$ chains too long.
- Typical choice: $M \sim N / 4$ $\Rightarrow$ constant-time ops.

M times faster than sequential search

Resizing in a separate-chaining hash table

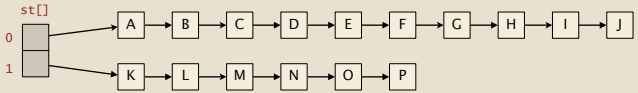
Goal. Average length of list N / M = constant.

- Double size of array M when $N / M \geq 8$.
- Halve size of array M when $N / M \leq 2$.
- Need to rehash all keys when resizing.

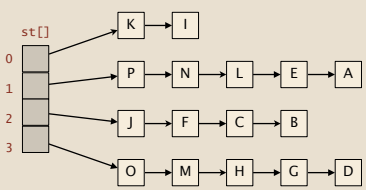
rehash kostet O(n) Laufzeit
aber amortisiert O(1) Laufzeit

x.hashCode() does not change
but hash(x) can change

before resizing



after resizing

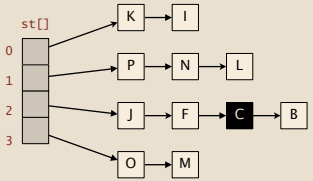


Deletion in a separate-chaining hash table

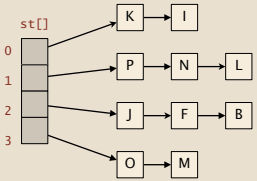
Q. How to delete a key (and its associated value)?

A. Easy: need only consider chain containing key.

before deleting C



after deleting C



Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✔	`compareTo()`
separate chaining	N	N	N	3-5 *	3-5 *	3-5 *		`equals()` `hashCode()`

alle Objekte in gleicher Schublade

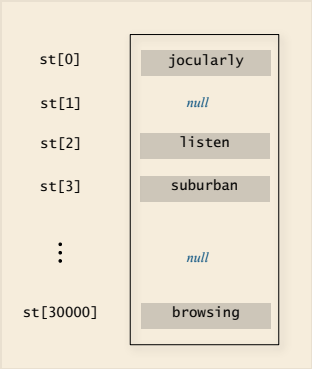
* under uniform hashing assumption

10.4 Linear Probing

Collision resolution: open addressing

Open addressing. [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]

When a new key collides, find next empty slot, and put it there.



linear probing (M = 30001, N = 15000)

Linear-probing hash table demo

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

linear-probing hash table

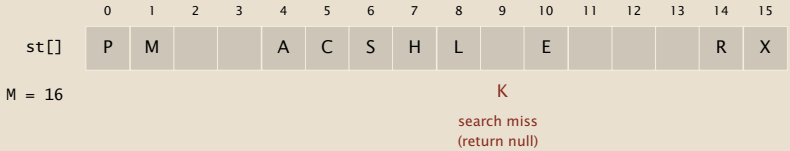
Linear-probing hash table demo

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search K

hash(K) = 5



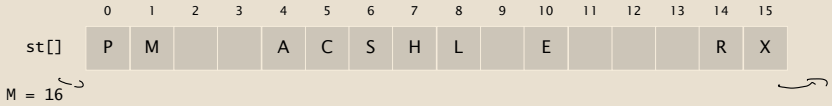
Linear-probing hash table summary

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

Note. Array size `M` must be greater than number of key-value pairs `N`.



Linear-probing hash table demo

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

linear-probing hash table

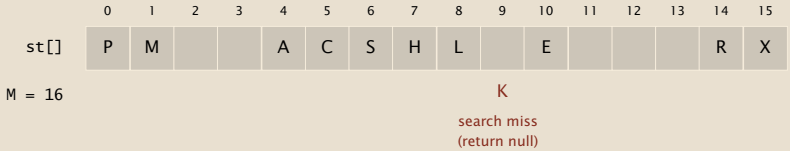
Linear-probing hash table demo

Hash. Map key to integer `i` between `0` and `M-1`.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

search K

hash(K) = 5



Linear-probing hash table summary

Hash. Map key to integer `i` between `0` and `M-1`.

Insert. Put at table index `i` if free; if not try `i+1`, `i+2`, etc.

Search. Search table index `i`; if occupied but no match, try `i+1`, `i+2`, etc.

Note. Array size `M` must be greater than number of key-value pairs `N`.

Linear-probing symbol table: Java implementation

```
public class LinearProbingHashST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   private int hash(Key key)            {  /* as before */   }

   private void put(Key key, Value val) {  /* next slide */  }

   public Value get(Key key)
   {
      for (int i = hash(key); keys[i] != null; i = (i+1) % M)
         if (key.equals(keys[i]))
             return vals[i];
      return null;
   }



}
```

array doubling and halving code omitted

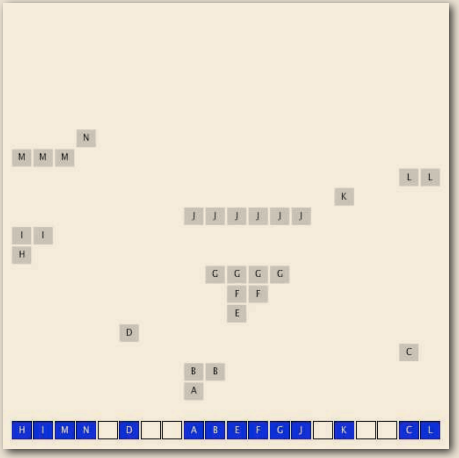
Linear-probing symbol table: Java implementation

```
public class LinearProbingHashST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   private int hash(Key key)   { /* as before      */  }

   private Value get(Key key)  { /* previous slide */  }

   public void put(Key key, Value val)
   {
      int i;
      for (i = hash(key); keys[i] != null; i = (i+1) % M)
         if (keys[i].equals(key))
             break;
      keys[i] = key;
      vals[i] = val;
   }

}
```

Clustering

Cluster. A contiguous block of items.

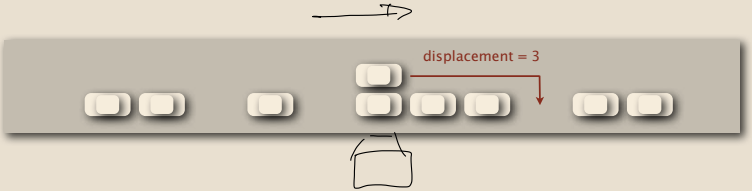
Observation. New keys likely to hash into middle of big clusters.



Knuth's parking problem

Model. Cars arrive at one-way street with M parking spaces.
Each desires a random space i : if space i is taken, try $i + 1, i + 2,$ etc.

Q. What is mean displacement of a car?



Half-full. With $M / 2$ cars, mean displacement is $\sim 3 / 2$.
Full. With M cars, mean displacement is $\sim \sqrt{\pi M / 8}$.

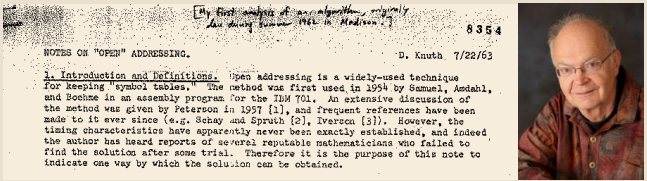
Analysis of linear probing

Proposition. Under uniform hashing assumption, the average # of probes in a linear probing hash table of size M that contains $N = \alpha M$ keys is:

$$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad \sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

search hit **search miss / insert**

Pf.



8354

NOTES ON "OPEN" ADDRESSING. D. Knuth 7/22/63

1. Introduction and Definitions. Open addressing is a widely-used technique for keeping "symbol tables." The method was first used in 1954 by Samuel, Amdahl, and Boehme in an assembly program for the IBM 701. An extensive discussion of the method was given by Peterson in 1957 [1], and frequent references have been made to it ever since (e.g. Schay and Spruth [2], Iverson [3]). However, the timing characteristics have apparently never been exactly established, and indeed the author has heard reports of several reputable mathematicians who failed to find the solution after some trial. Therefore it is the purpose of this note to indicate one way by which the solution can be obtained.

Parameters.

- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N / M \sim ½$. ← # probes for search hit is about 3/2; # probes for search miss is about 5/2

Resizing in a linear-probing hash table

Goal. Average length of list $N / M \leq \frac{1}{2}$.

- Double size of array M when $N / M \geq \frac{1}{2}$.
- Halve size of array M when $N / M \leq \frac{1}{8}$.
- Need to rehash all keys when resizing.

before resizing

	0	1	2	3	4	5	6	7
keys[]		E	S			R	A	
vals[]		1	0			3	2	

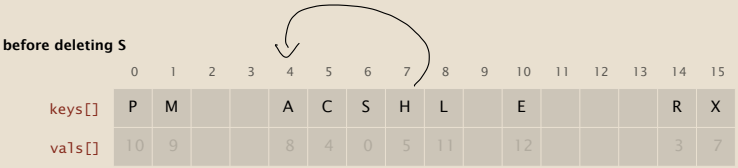
after resizing

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]					A		S				E				R	
vals[]					2		0				1				3	

Deletion in a linear-probing hash table

Q. How to delete a key (and its associated value)?

A. Requires some care: can't just delete array entries.



ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2} N$	N	$\frac{1}{2} N$		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✔	`compareTo()`
separate chaining	N	N	N	3-5 *	3-5 *	3-5 *		`equals()` `hashCode()`
linear probing	N	N	N	3-5 *	3-5 *	3-5 *		`equals()` `hashCode()`

* under uniform hashing assumption

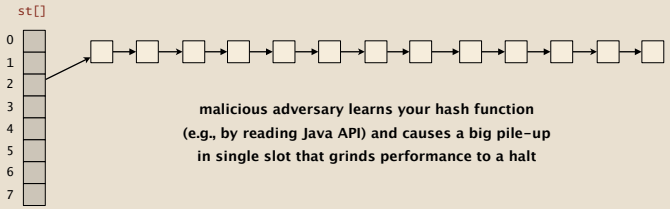
10.5 Context

War story: algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?

A. Obvious situations: aircraft control, nuclear reactor, pacemaker.

A. Surprising situations: denial-of-service attacks.



Real-world exploits. [Crosby-Wallach 2003]

- Bro server: send carefully chosen packets to DOS the server, using less bandwidth than a dial-up modem.
- Perl 5.8.0: insert carefully chosen strings into associative array.
- Linux 2.4.20 kernel: save files with carefully chosen names.

War story: algorithmic complexity attacks

A Java bug report.

Jan Lieskovsky 2011-11-01 10:13:47 EDT **Description**

Julian Wälde and Alexander Klink reported that the String.hashCode() hash function is not sufficiently collision resistant. hashCode() value is used in the implementations of HashMap and Hashtable classes:

http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html
http://docs.oracle.com/javase/6/docs/api/java/util/Hashtable.html

A specially-crafted set of keys could trigger hash function collisions, which can degrade performance of HashMap or Hashtable by changing hash table operations complexity from an expected/average O(1) to the worst case O(n). Reporters were able to find colliding strings efficiently using equivalent substrings and meet in the middle techniques.

This problem can be used to start a denial of service attack against Java applications that use untrusted inputs as HashMap or Hashtable keys. An example of such application is web application server (such as tomcat, see ~~bug #750521~~) that may fill hash tables with data from HTTP request (such as GET or POST parameters). A remote attack could use that to make JVM use excessive amount of CPU time by sending a POST request with large amount of parameters which hash to the same value.

This problem is similar to the issue that was previously reported for and fixed in e.g. perl:
http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003.pdf

Algorithmic complexity attack on Java

Goal. Find family of strings with the same hash code.

Solution. The base-31 hash code is part of Java's string API.

key	hashCode()
"Aa"	2112
"BB"	2112

key	hashCode()
"AaAaAaAa"	-540425984
"AaAaAaBB"	-540425984
"AaAaBBAa"	-540425984
"AaAaBBBB"	-540425984
"AaBBAaAa"	-540425984
"AaBBAaBB"	-540425984
"AaBBBBAa"	-540425984
"AaBBBBBB"	-540425984

key	hashCode()
"BBAaAaAa"	-540425984
"BBAaAaBB"	-540425984
"BBAaBBAa"	-540425984
"BBAaBBBB"	-540425984
"BBBBAaAa"	-540425984
"BBBBAaBB"	-540425984
"BBBBBBAa"	-540425984
"BBBBBBBB"	-540425984

2^N strings of length 2N that hash to same value!

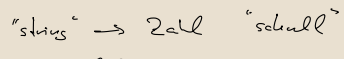
HashMap wechselt verkettete
Liste zu BST

Diversion: one-way hash functions

One-way hash function. "Hard" to find a key that will hash to a desired value (or two keys that hash to same value).

Ex. MD4, MD5, SHA-0, SHA-1, SHA-2, WHIRLPOOL, RIPEMD-160, ...

known to be insecure




```
String password = args[0];
MessageDigest sha1 = MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.digest(password);

/* prints bytes as hex string */
```

Applications. Digital fingerprint, message digest, storing passwords.

Caveat. Too expensive for use in ST implementations.

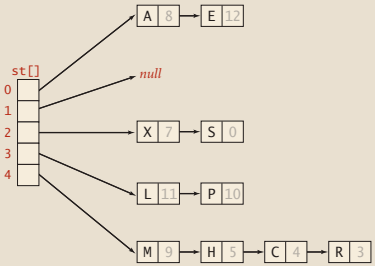
Separate chaining vs. linear probing

Separate chaining.

- Performance degrades gracefully.
- Clustering less sensitive to poorly-designed hash function.

Linear probing.

- Less wasted space.
- Better cache performance.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C	S	H	L		E				R	X
vals[]	10	9			8	4	0	5	11		12				3	7

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. [separate-chaining variant]

- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to $\log \log N$.

Double hashing. [linear-probing variant]

- Use linear probing, but skip a variable amount, not just 1 each time.
- Effectively eliminates clustering.
- Can allow table to become nearly full.
- More difficult to implement delete.

Cuckoo hashing. [linear-probing variant]

- Hash key to two positions; insert key into either position; if occupied, reinsert displaced key into its alternative position (and recur).
- Constant worst-case time for search.

10.6 Universal Hashing

Hashing Worst Case

- Für *jede* feste Hash-Funktion (aus einem hinreichend großen Universum) gibt es Worst-Case Eingaben (Menge von Objekten), die auf den gleichen Hash-Wert abgebildet werden. (*Pigeon-hole principle* / Taubenschlagprinzip)

⇝ *Im Worst-Case bringt Hashing* ***gar nichts****!*

Hashing Worst Case

- Für *jede* feste Hash-Funktion (aus einem hinreichend großen Universum) gibt es Worst-Case Eingaben (Menge von Objekten), die auf den gleichen Hash-Wert abgebildet werden. (*Pigeon-hole principle* / Taubenschlagprinzip)

⇝ *Im Worst-Case bringt Hashing **gar nichts**!*

- **Aber:** Für *sinnvolle* Hash-Funktionen sind die *meisten* Eingaben OK.

😈 *Können wir uns darauf verlassen?*

Hashing Worst Case

- Für *jede* feste Hash-Funktion (aus einem hinreichend großen Universum) gibt es Worst-Case Eingaben (Menge von Objekten), die auf den gleichen Hash-Wert abgebildet werden. (*Pigeon-hole principle* / Taubenschlagprinzip)

⇝ *Im Worst-Case bringt Hashing **gar nichts**!*

- **Aber:** Für *sinnvolle* Hash-Funktionen sind die *meisten* Eingaben OK.

😈 *Können wir uns darauf verlassen?*

⇝ *Randomisierung!*

- Wenn wir eine **zufällige** sinnvolle Hash-Funktion wählen ist die erwartete Performance top!
- Niemand weiß welche Hash-Funktion verwendet wird (wir auch nicht!) 🤷

Hashing Worst Case

- Für *jede* feste Hash-Funktion (aus einem hinreichend großen Universum) gibt es Worst-Case Eingaben (Menge von Objekten), die auf den gleichen Hash-Wert abgebildet werden. (*Pigeon-hole principle* / Taubenschlagprinzip)

⇝ *Im Worst-Case bringt Hashing **gar nichts**!*

- **Aber:** Für *sinnvolle* Hash-Funktionen sind die *meisten* Eingaben OK.

😈 *Können wir uns darauf verlassen?*

⇝ *Randomisierung!*

- Wenn wir eine **zufällige** sinnvolle Hash-Funktion wählen ist die erwartete Performance top!
- Niemand weiß welche Hash-Funktion verwendet wird (wir auch nicht!) 🤷

⇝ Dafür benötigen wir aber eine Klasse von Hash-Funktionen die

1. Umfassend genug ist, dass die erwartete Performance einer zufälligen Funktion gut ist
2. Kompakt zu speichern ist (da die Funktion ja nicht mehr fest steht)
3. Schnell zu berechnen ist

Universal Hashing

Tatsächlich gibt es solche Klassen!

Universelle Klassen von Hashfunktionen
(*universal family of hash functions*)

- Theorie dahinter führt uns zu weit $\rightsquigarrow$ *Effiziente Algorithmen*
- hier nur das praktisch relevanteste Beispiel (ohne Beweise)

Universal Hashing

Tatsächlich gibt es solche Klassen!

Universelle Klassen von Hashfunktionen
(*universal family of hash functions*)

- Theorie dahinter führt uns zu weit ⇝ *Effiziente Algorithmen*
- hier nur das praktisch relevanteste Beispiel (ohne Beweise)

Multiplicative Universal Hashing

- Brauchen $M = 2^d$ Slots und $u = 2^w$ (Zweierpotenzen, w Wortbreite)
- Betrachte Funktionen $h_z(x) = ((z \cdot x) \bmod 2^w) \operatorname{div} 2^{w-d}$
  div als Bit-Shift realisierbar:

```
int hash(Object x) { return (z * x.hashCode()) >>> (w-d); }
```

- Verwende h_Z für Z eine zufällige **ungerade** Zahl in $[1..2^w)$

⇝ **Lemma:** Für $x \neq y \in [0..2^w)$ gilt: $\mathbb{P}[h_Z(x) = h_Z(y)] \leq \frac{2}{M}$

⇝ genug für $O(1)$ erwartete Average-Case Laufzeit mit chaining hashing

Hash tables vs. balanced search trees

Hash tables.

- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus $\log N$ compares).
- Better system support in Java for strings (e.g., cached hash code).

amortized?

Balanced search trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement `compareTo()` correctly than `equals()` and `hashCode()`.

Java system includes both.

- Red-black BSTs: `java.util.TreeMap`, `java.util.TreeSet`.
- Hash tables: `java.util.HashMap`, `java.util.IdentityHashMap`.