3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%.

"
Symbol table implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered ops?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>red–black BST</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>hashing</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$1^\dagger$</td>
</tr>
</tbody>
</table>

Q. Can we do better?
A. Yes, but with different access to the data.

† under suitable technical assumptions
Hashing: basic plan

Save key–value pairs in a **key-indexed table** (index is a function of the key).

**Hash function.** Method for computing array index from key.

- 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.

**Issues.**

- **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).
Equality test

All Java classes inherit a method `equals()`.

**Java requirements.** For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is true.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is false.

**Default implementation.** `(x == y)`

**Customized implementations.** `Integer`, `Double`, `String`, `java.net.URL`, ...

**User-defined implementations.** Some care needed.
Implementing equals for user-defined types

Seems easy.

```java
class Date {
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Date that) {
        if (this.day != that.day) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year) return false;
        return true;
    }
}
```

check that all significant fields are the same
Implementing equals for user-defined types

Seems easy, but requires some care.

```java
public final class Date {
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Object y) {
        if (y == this) return true;
        if (y == null) return false;
        if (y.getClass() != this.getClass())
            return false;

        Date that = (Date) y;
        if (this.day != that.day) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year) return false;
        return true;
    }
}
```

typically unsafe to use `equals()` with inheritance (would violate symmetry)
must be `Object`
optimization (for reference equality)
check for `null`
objects must be in the same class (religion: `getClass()` vs. `instanceof`)
cast is now guaranteed to succeed
check that all significant fields are the same
Equals design

“Standard” recipe for user-defined types.

- Optimization for reference equality.
- Check against null.
- Check that two objects are of the same type; cast.
- Compare each significant field:
  - if field is a primitive type, use \( \text{==} \)
  - if field is an object, use \( \text{equals()} \) and apply rule recursively
  - if field is an array of primitives, use \( \text{Arrays.equals()} \)
  - if field is an array of objects, use \( \text{Arrays.deepEquals()} \)

Best practices.

- Do not use calculated fields that depend on other fields.
- Compare fields mostly likely to differ first.
- Make \( \text{compareTo()} \) consistent with \( \text{equals()} \).

\[ x.\text{equals}(y) \text{ if and only if } (x.\text{compareTo}(y) == 0) \]
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
Computing the hash function

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

**Practical challenge.** Need different approach for each key type.

**Ex 1.** Last 4 digits of Social Security number.

**Ex 2.** Last 4 digits of phone number.

thoroughly researched problem, still problematic in practical applications
Which is the last digit of your day of birth?

A. 0 or 1
B. 2 or 3
C. 4 or 5
D. 6 or 7
E. 8 or 9
Which is the last digit of your year of birth?

A. 0 or 1
B. 2 or 3
C. 4 or 5
D. 6 or 7
E. 8 or 9
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()`.

![Diagram](image)

**Default implementation.** Memory address of `x`.

**Legal (but useless) implementation.** Always return 17.

**Customized implementations.** Integer, Double, String, java.net.URL, ...

**User-defined types.** Users are on their own.
Implementing hash code: integers, booleans, and doubles

Java library implementations

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

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

```java
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: arrays

31x + y rule.
- Initialize hash to 1.
- Repeatedly multiply hash by 31 and add next integer in array.

```java
public class Arrays {
    ...

    public static int hashCode(int[] a) {
        if (a == null)
            return 0; // special case for null

        int hash = 1;
        for (int i = 0; i < a.length; i++)
            hash = 31*hash + a[i];
        return hash;
    }
}
```

Java library implementation
Implementing hash code: strings

Treat a string as an array of characters.

```
public 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;
    }
}
```

Java library implementation

<table>
<thead>
<tr>
<th>char</th>
<th>Unicode</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>'a'</td>
<td>97</td>
</tr>
<tr>
<td>'b'</td>
<td>98</td>
</tr>
<tr>
<td>'c'</td>
<td>99</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
## Aside: string hash collisions in Java

2\(^n\) strings of length 2\(n\) that all hash to the same value!

<table>
<thead>
<tr>
<th>key</th>
<th>hashCode()</th>
<th>key</th>
<th>hashCode()</th>
<th>key</th>
<th>hashCode()</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Aa&quot;</td>
<td>2112</td>
<td>&quot;AaaAaaAa&quot;</td>
<td>-540425984</td>
<td>&quot;BBAaaAaa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;BB&quot;</td>
<td>2112</td>
<td>&quot;AaaAaaBB&quot;</td>
<td>-540425984</td>
<td>&quot;BBAaaBBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;AaaAaaa&quot;</td>
<td>-540425984</td>
<td>&quot;BBAaaBaa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;AaaAaaBBB&quot;</td>
<td>-540425984</td>
<td>&quot;BBAaaaB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;AaaaBaa&quot;</td>
<td>-540425984</td>
<td>&quot;BBBBaaaA&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;AaaaBaaa&quot;</td>
<td>-540425984</td>
<td>&quot;BBBBBaaA&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;AaaaBaaaB&quot;</td>
<td>-540425984</td>
<td>&quot;BBBBBBaa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;AaaaBaaBB&quot;</td>
<td>-540425984</td>
<td>&quot;BBBBBaaB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;BBBBBaaA&quot;</td>
<td>-540425984</td>
<td>&quot;BBBBBBaa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;BBBBBaaa&quot;</td>
<td>-540425984</td>
<td>&quot;BBBBBBBb&quot;</td>
<td>-540425984</td>
</tr>
</tbody>
</table>

**Java fail**

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:

Implementing hash code: user-defined types

```java
public final class 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 = 1;
        hash = 31*hash + who.hashCode();
        hash = 31*hash + when.hashCode();
        hash = 31*hash + ((Double) amount).hashCode();
        return hash;
    }
}

for reference types, use hashCode()
for primitive types, use hashCode() of wrapper type
```
public final class 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()
    {
        return Objects.hash(who, when, amount); // shorthand
    }
}
Hash code design

“Standard” recipe for user-defined types.
- Combine each significant field using the $31x + y$ rule.
- Shortcut 1: use `Objects.hash()` for all fields (except arrays).
- Shortcut 2: use `Arrays.hashCode()` for primitive arrays.
- Shortcut 3: use `Arrays.deepHashCode()` for object arrays.

In practice. Recipe above works reasonably well; used in Java libraries.
In theory. Keys are bitstring; “universal” family of 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.
Which function maps hashable keys to integers between 0 and m−1?

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

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

C. Both A and B.

D. Neither A nor B.
**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

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

**bug**

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

**1-in-a-billion bug**

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

```java
private int hash(Key key)
{
    return (key.hashCode() & 0xffffffff) % m;
}
```

**correct**

if $m$ is a power of 2, can use `key.hashCode() & (m-1)`
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.

Bad news. [birthday problem]

- In a random group of 23 people, more likely than not that two people share the same birthday.
- Expect two balls in the same bin after \( \sim \sqrt{\pi \frac{m}{2}} \) tosses.

23.9 when \( m = 365 \)
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.

Good news. [load balancing]

- When $n \gg m$, expect most bins to have approximately $n / m$ balls.
- When $n = m$, expect most loaded bin has $\sim \ln m / \ln \ln m$ balls.

Hash value frequencies for words in Tale of Two Cities ($m = 97$)
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
**Collisions**

**Collision.** Two distinct keys hashing to same index.

- Birthday problem $\Rightarrow$ can’t avoid collisions. 

- Load balancing $\Rightarrow$ no index gets too many collisions. 
  $\Rightarrow$ ok to scan through all colliding keys.

Unless you have a ridiculous (quadratic) amount of memory.
Separate-chaining symbol table

Use an array of $m$ 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 in chain).

separate-chaining hash table ($m = 4$)

```
put(L, 11)
hash(L) = 3
```
Separate-chaining symbol table

Use an array of \( m \) 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 in chain).
- Search: sequential search in \( i^{th} \) chain.

separate-chaining hash table (\( m = 4 \))

\[
\begin{array}{c}
\text{get(E)} \\
\text{hash(E) = 1}
\end{array}
\]
public class SeparateChainingHashST<Key, Value> {
    private int m = 128; // number of chains
    private Node[] st = new Node[m]; // array of chains

    private static class Node {
        private Object key;   // declare key and value of type Object
        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;
    }
}

Separate-chaining symbol table: Java implementation
public class SeparateChainingHashST<Key, Value> {
    private int m = 128; // 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

Recall load balancing. Under uniform hashing assumption, length of each chain is approximately $n/m$.

Consequence. Number of probes for search/insert is proportional to $n/m$.

- $m$ too small $\Rightarrow$ chains too long.
- $m$ too large $\Rightarrow$ too many empty chains.
- Typical choice: $m \sim \frac{1}{4} n \Rightarrow$ constant time per operation.
Resizing in a separate-chaining hash table

**Goal.** Average length of list \( n/m = \text{constant}. \)
- Double length \( m \) of array when \( n/m \geq 8 \).
- Halve length \( m \) of array when \( n/m \leq 2 \).
- Note: need to rehash all keys when resizing.

before resizing (\( n/m = 8 \))

\[
\begin{array}{c}
0 \\
1 \\
2 \\
3 \\
\end{array}
\]

after resizing (\( n/m = 4 \))

\[
\begin{array}{c}
0 \\
1 \\
2 \\
3 \\
\end{array}
\]

x.hashCode() does not change; but hash(x) typically does.
Deletion in a separate-chaining hash table

Q. How to delete a key (and its associated value)?
A. Easy: need to consider only chain containing key.

Before deleting C:

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>N</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>F</td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>O</td>
<td>M</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

After deleting C:

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>N</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>F</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>O</td>
<td>M</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Symbol table implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered ops?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search</td>
</tr>
<tr>
<td>sequential search</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>red-black BST</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>separate chaining</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$1^\dagger$</td>
</tr>
</tbody>
</table>

$\dagger$ under uniform hashing assumption
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
Collision resolution: open addressing

Open addressing. [Amdahh–Boehme–Rocherster–Samuel, IBM 1953]

- Maintain keys and values in two parallel arrays.
- When a new key collides, find next empty slot and put it there.

linear-probing hash table (m = 16, n =10)

<table>
<thead>
<tr>
<th>keys[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>M</td>
<td>A</td>
<td>C</td>
<td>H</td>
<td>L</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>put(K, 14)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash(K) = 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vals[]</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>5</td>
<td>6</td>
<td>12</td>
<td>13</td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 length $m$ must be greater than number of key–value pairs $n$.

<table>
<thead>
<tr>
<th>keys[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>M</td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>L</td>
<td>E</td>
<td>R</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$m = 16$
Linear-probing symbol table: Java implementation

```java
public class LinearProbingHashST<Key, Value> {
    private int m = 32768;
    private Value[] vals = (Value[]) new Object[m];
    private Key[] keys = (Key[]) new Object[m];

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

    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 resizing code omitted
public class LinearProbingHashTable<Key, Value> {

    private int m = 32768;
    private Value[] vals = (Value[]) new Object[m];
    private Key[] keys = (Key[]) new Object[m];

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

    private Value get(Key key) { /* prev 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;
    }
}
Under the uniform hashing assumption, where is the next key most likely to be added in this linear-probing hash table (no resizing)?

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| H | I | M | N | D |   | A | B | E | F | G | J |   | K |   | C | L |   |   |   |

A. Index 7.
B. Index 14.
C. Either index 4 or 14.
D. All open indices are equally likely.
**Cluster.** A contiguous block of items.

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

\[
\frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right) \quad \text{search hit}
\]

\[
\frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right) \quad \text{search miss / insert}
\]

**Pf.** [beyond course scope]

---

**Parameters.**

- \( m \) too large \( \Rightarrow \) too many empty array entries.
- \( m \) too small \( \Rightarrow \) search time blows up.
- Typical choice: \( \alpha = n / m \approx \frac{1}{2} \).

# 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 length of array $m$ when $n/m \geq \frac{1}{2}$.
- Halve length of array $m$ when $n/m \leq \frac{1}{8}$.
- Need to rehash all keys when resizing.

<table>
<thead>
<tr>
<th>before resizing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>keys[]</td>
</tr>
<tr>
<td>vals[]</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>after resizing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>keys[]</td>
</tr>
<tr>
<td>vals[]</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
How to delete a key-value pair from a linear-probing hash table?

A. Search for key; remove key (and value) from arrays.

B. Search for key; remove key (and value) from arrays.
   Shift all keys in cluster after deleted key over 1 position to left.

C. Both A and B.

D. Neither A nor B.
## ST implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Average Case</th>
<th>Ordered ops?</th>
<th>Key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search</td>
</tr>
<tr>
<td>sequential search</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>red-black BST</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>separate chaining</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$1^\dagger$</td>
</tr>
<tr>
<td>linear probing</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$1^\dagger$</td>
</tr>
</tbody>
</table>

† under uniform hashing assumption
3-Sum (Revisited)

3-Sum. Given $n$ distinct integers, find three such that $a + b + c = 0$.

Goal. $n^2$ expected time case, $n$ extra space.
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
War story: algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?
A. Obvious situations: aircraft control, nuclear reactor, pacemaker, HFT, ...
A. Surprising situations: denial-of-service attacks.

Real-world exploits. [Crosby–Wallach 2003]
- Linux 2.4.20 kernel: save files with carefully chosen names.
- Bro server: send carefully chosen packets to DOS the server, using less bandwidth than a dial-up modem.
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-256, SHA-512, WHIRLPOOL, ....

String password = "OPEN_SESAME";
MessageDigest sha256 = MessageDigest.getInstance("SHA-256");
byte[] bytes = sha256.digest(password.getBytes());

32 bytes (256 bits) for SHA-256

Applications. Digital signatures, message digests, password verification, cryptocurrencies, blockchain, Git commit identifiers, ....

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.
- More probes because of clustering.
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 \( \sim \lg \ln n \).

**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.
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).

Balanced search trees.
- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement `compareTo()` than `hashCode()`.

Java system includes both.

separate chaining
(if chain gets too long, use red–black BST for chain)

linear probing