3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
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>hashing</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>1†</td>
</tr>
</tbody>
</table>

Q. Can we do better?

A. Yes, but only with different access to the symbol table keys.

† subject to certain technical assumptions
Hashing: basic plan

Save key-value pairs in a key-indexed table, where the index is a function of the key.

**Hash function:** Mathematical function that maps (hashes) a key to an array index.

**Collision:** Two distinct keys that hash to same index.

**Issue.** Collisions are unavoidable.

- How to limit collisions?
  [good hash functions]
- How to accommodate collisions?
  [novel algorithms and data structures]
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
Designing a hash function

**Required properties.** [for correctness]
- Deterministic.
- Each key hashes to a table index between 0 and \( m - 1 \).

**Desirable properties.** [for performance]
- Very fast to compute.
- For any subset of \( n \) input keys, each table index gets approximately \( n / m \) keys.

leads to good hash-table performance  
\( (m = 10, n = 20) \)

leads to poor hash-table performance  
\( (m = 10, n = 20) \)
Designing a hash function

**Required properties.** [for correctness]
- Deterministic.
- Each key hashes to a table index between 0 and $m - 1$.

**Desirable properties.** [for performance]
- Very fast to compute.
- For any subset of $n$ input keys, each table index gets approximately $n / m$ keys.

**Ex 1.** $[m = 10,000]$ Last 4 digits of U.S. Social Security number.

**Ex 2.** $[m = 10,000]$ Last 4 digits of phone number.
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
Hash tables: quiz 2

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 hashCode() conventions

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

**Required.** [for correctness] If `x.equals(y)`, then `x.hashCode() == y.hashCode()`.

**Highly desirable.** [for efficiency] If `!x.equals(y)`, then `x.hashCode() != y.hashCode()`.

![Diagram](image)

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

Legal (but highly undesirable) implementation. Always return 17.

User-defined types. Users are on their own.
Implementing `hashCode()`: integers and doubles

Java library implementations

```java
public final class Integer {
    private final int value;
    ...
    public int hashCode() {
        return value;
    }
}

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
- If used only least significant 32 bits, all integers between $-2^{21}$ and $2^{21}$ would have same hash code (0)
Implementing `hashCode()`: user-defined types

**31x + y rule.**

- Initialize hash to 1.
- Repeatedly multiply hash by 31 and add hash of each significant field.

```java
public final class Transaction {
    private final String who;
    private final Date when;
    private final double amount;

    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.
Implementing hashCode(): user-defined types

31x + y rule.
- Initialize hash to 1.
- Repeatedly multiply hash by 31 and add hash of each significant field.

```java
public final class Transaction {
    private final String who;
    private final Date when;
    private final double amount;

    public int hashCode() {
        return Objects.hash(who, when, amount);
    }
}
```

A varargs method that applies the 31x + y rule to its arguments.
Implementing `hashCode()`

“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 array of primitives.

**Principle.** Every significant field contributes to hash.

**In practice.** Recipe above works reasonably well; used in Java libraries.
Which Java function maps hashable keys to integers between 0 and m−1?

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

B. 
```java
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).

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

$m$ typically a prime or a power of 2

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

**Bug**
the remainder operator can evaluate to a negative integer

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

1-in-a-billion bug

hashCode() of "polygenelubricants" and new Double(-0.0) is $-2^{31}$

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

**Correct**
Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to any of \( m \) possible indices.

Bins and balls. Toss \( n \) 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 \( \sim \sqrt{\frac{\pi m}{2}} \) tosses.

23.9 when \( m = 365 \)
Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to any of \( m \) possible indices.

Bins and balls. Toss \( n \) balls uniformly at random into \( m \) bins.

Good news. [load balancing]

- When \( n >> m \), expect most bins to have approximately \( n / m \) balls.
- When \( n = m \), expect most loaded bin has \( \sim \ln n / \ln \ln n \) balls.
3.4 Hash Tables

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

Collision. Two distinct keys that hash to the same index.

- Birthday problem $\Rightarrow$ can’t avoid collisions.
  - unless you have a ridiculous (quadratic) amount of memory

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

\[
\begin{array}{c}
\text{hash("USA") = 3} \\
\text{hash("ITA") = 3}
\end{array}
\]
Separate-chaining hash table

Use an array of \( m \) linked lists.

- **Hash**: map key to table index \( i \) between 0 and \( m - 1 \).
- **Insert**: add key-value pair at front of chain \( i \) (if not already in chain).

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

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

Use an array of $m$ linked lists.

- **Hash:** map key to table index $i$ between 0 and $m - 1$.
- **Insert:** add key–value pair at front of chain $i$ (if not already in chain).
- **Search:** perform sequential search in chain $i$.

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

get(E)
hash(E) = 1
Separate-chaining hash table: Java implementation

```java
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) {
        // as before
    }

    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 hash table: Java implementation

```java
public class SeparateChainingHashTable<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) {
        /* as before */
    }

    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 the uniform hashing assumption, the length of each chain is tightly concentrated around mean $= \frac{n}{m}$.

Consequence. Expected number of probes for search/insert is $\Theta\left(\frac{n}{m}\right)$.

- $m$ too small $\Rightarrow$ chains too long.
- $m$ too large $\Rightarrow$ too many empty chains.
- Typical choice: $m \sim \frac{1}{4}n$ $\Rightarrow$ $\Theta(1)$ time for search/insert.

calls to either equals() or hashCode()

$m$ times faster than sequential search

hash value frequencies for words in Tale of Two Cities (m = 97)
Resizing in a separate-chaining hash table

**Goal.** Average length of chain $n/m$ is $\Theta(1)$.

- 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.

![Diagram of hash table before and after resizing](https://via.placeholder.com/150)

- $x$.hashCode() does not change; but hash(x) typically does.
## 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†</td>
</tr>
</tbody>
</table>

can achieve $\Theta(1)$ probabilistic, amortized guarantee by choosing a hash function at random (see "universal hashing")

† under uniform hashing assumption
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
Linear-probing hash table: insert

- Maintain key–value pairs in two parallel arrays, with one key per cell.
- Resolve collisions by **linear probing**: search successive cells until either finding the key or an unused cell.

## Inserting into a linear-probing hash table.

### linear-probing hash table

<table>
<thead>
<tr>
<th></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>keys[]</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>R</td>
<td>X</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></td>
<td></td>
<td></td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**put(K, 14)**

**hash(K) = 7**

K

14
Linear-probing hash table: search

- Maintain key-value pairs in two parallel arrays, with one key per cell.
- Resolve collisions by linear probing:
  search successive cells until either finding the key or an unused cell.

Searching in a linear-probing hash table.

<table>
<thead>
<tr>
<th>keys[]</th>
<th>P</th>
<th>M</th>
<th>A</th>
<th>C</th>
<th>H</th>
<th>L</th>
<th>K</th>
<th>E</th>
<th>R</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>vals[]</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>5</td>
<td>6</td>
<td>12</td>
<td>14</td>
<td>13</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

get(K)  hash(K) = 7
get(Z)  hash(Z) = 8
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 $i + 1, i + 2, \ldots$.

**Search.** Search table index $i$; if occupied but no match, try $i + 1, i + 2, \ldots$.

**Note.** Array length $m$ must be greater than number of key-value pairs $n$.

---

<table>
<thead>
<tr>
<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>P</td>
<td>M</td>
<td>A</td>
<td>C</td>
<td>S</td>
<td>H</td>
<td>L</td>
<td>E</td>
<td></td>
<td></td>
<td>R</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

keys[]

$m = 16$
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)
    { /* 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;
    }
}
```
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)
    { /* as before */ }

    public 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;
    }

    // array resizing code omitted
}
```
Hash tables: quiz 4

Under the uniform hashing assumption, where is the next key most likely to be added in this linear-probing hash table (no resizing)?

A. Index 4.
B. Index 17.
C. Either index 4 or 17.
D. All open indices are equally likely.
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

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

- search hit
- 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: $\alpha = n / m \sim \frac{1}{2}$. 
  
  # probes for search hit is about 3/2  
  # probes for search miss is about 5/2
## 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linear probing</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
<td>$1^\dagger$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

† under uniform hashing assumption
Separate chaining vs. linear probing

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

Linear probing.
• Unrivaled data locality.
• More probes because of clustering.
3-SUM (revisited)

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

Goal. $\Theta(n^2)$ expected time; $\Theta(n)$ extra space.
3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
Hashing: variations on the theme

Many many improved versions have been studied.

Use different probe sequence, i.e., not \( h(k) \), \( h(k) + 1 \), \( h(k) + 2 \), ...
[ quadratic probing, double hashing, pseudo-random probing, ... ]

During insertion, relocate some of the keys already in the table.
[ Cuckoo hashing, Robin Hood hashing, Hopscotch hashing, ... ]

Insert tombstones prophylactically, to avoid primary clustering.
[ graveyard hashing ]

Google Swiss Table  Facebook F14  Python 3

eliminates primary clustering, which enables higher load factor / less memory (but sacrifices data locality)

reduces worst-case time for search

eliminates primary clustering; maintains data locality
Hash tables vs. balanced search trees

Hash tables.
- Simpler to code.
- Typically faster in practice.
- No effective alternative for unordered keys.

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

Java includes both.
- BSTs: `java.util.TreeMap`.

Separate chaining (Java 8: if chain gets too long, use red-black BST for chain)
Algorithmic complexity attacks

**Q.** Is the uniform hashing assumption important in practice?

**A1.** Yes: aircraft control, nuclear reactor, pacemaker, HFT, missile-defense system, …

**A2.** Yes: denial-of-service (DoS) 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.
File verification. When downloading a file from the web:

- Vendor publishes hash of file.
- Client checks whether hash of downloaded file matches.
- If mismatch, file corrupted. (e.g., error in transmission or infected by virus)
Hashing: cryptographic applications

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

Ex. MD5, SHA-1, SHA-256, SHA-512, SHA3-512, Whirlpool, BLAKE3, ....

Applications. File verification, digital signatures, cryptocurrencies, password authentication, blockchain, non-fungible tokens, Git commit identifiers, ....
ALGORITHM (NOUN)
WORD USED BY
PROGRAMMERS WHEN
THEY DO NOT WANT TO
EXPLAIN WHAT THEY DID.