3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- applications

ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>search</th>
<th>insert</th>
<th>delete</th>
<th>search hit</th>
<th>insert</th>
<th>delete</th>
<th>ordered iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>binary search (array)</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
<td>N/2</td>
<td>N/2</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.38 lg N</td>
<td>1.38 lg N</td>
<td>?</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>red-black tree</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>1.00 lg N</td>
<td>1.00 lg N</td>
<td>1.00 lg N</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

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

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.

Optimize judiciously

“More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity.” — William A. Wulf

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.” — Donald E. Knuth

“We follow two rules in the matter of optimization: Rule 1: Don’t do it. Rule 2 (for experts only). Don’t do it yet—that is, not until you have a perfectly clear and unoptimized solution.” — M. A. Jackson

Reference: Effective Java by Joshua Bloch
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.

<table>
<thead>
<tr>
<th>Index</th>
<th>Hashed Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>&quot;it&quot;</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

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.
• Limitations on both time and space: hashing (the real world).

Equality test

Needed because hash methods do not use compareTo().

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, File, URL, Date, ...
User-defined implementations. Some care needed.

Implementing equals for user-defined types

Seems easy

public class Record {
   private final String name;
   private final long val;
   ...
   public boolean equals(Record y) {
      Record that = y;
      return (this.val == that.val) &&
             (this.name.equals(that.name));
   }
}
Implementing equals for user-defined types

Seems easy, but requires some care.

```java
public final class Record
{
    private final String name;
    private final long val;
    ...
    public boolean equals(Object y)
    {
        if (y == this) return true;
        if (y == null) return false;
        if (y.getClass() != this.getClass())
            return false;
        Record that = (Record) y;
        return (this.val == that.val) &&
            (this.name.equals(that.name));
    }
}
```

Implementing hash code: integers and doubles

```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));
    }
}
```

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`.
**Customized implementations.** `Integer`, `Double`, `String`, `File`, `URL`, `Date`, ...
**User-defined types.** Users are on their own.

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.

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.

Practical challenge. Need different approach for each key type.

Ex 1. Phone 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.

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.
Implementing hash code: strings

```java
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;
  }
}
```

- Horner’s method to hash string of length L: \( L \) multiplies/adds.
- Equivalent to \( h = 31^{L-1} \cdot s_0 + \ldots + 31^2 \cdot s_{L-3} + 31^1 \cdot s_{L-2} + 31^0 \cdot s_{L-1}. \)

**Ex.**
String s = “call”;
int code = s.hashCode();

```
3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0 + 108 \cdot 31^0 (97 + 31 \cdot (99))
```

Implementing hash code: user-defined types

```java
public final class Record{
  private String name;
  private int id;
  private double value;
  public Record(String name, int id, double value) {
    /* as before */
  }
  ...
  public boolean equals(Object y) {
    /* as before */
  }
  public int hashCode() {
    int hash = 17;
    hash = 31*hash + name.hashCode();
    hash = 31*hash + id;
    hash = 31*hash + Double.valueOf(value).hashCode();
    return hash;
  }
}
```

"Standard" recipe for user-defined types:
- Combine each significant field using the \( 31x + y \) rule.
- If field is a primitive type, use built-in hash code.
- If field is an array, apply to each element.
- If field is an object, apply rule recursively.

**In practice.** Recipe works reasonably well; used in Java libraries.
**In theory.** Need a theorem for each type to ensure reliability.

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

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

Correct

Uniform hashing assumption

Assumption J (uniform hashing 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 \(\sim \sqrt{\pi \frac{M}{2}}\) tosses.

Coupon collector. Expect every bin has \(\approx 1\) 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

Assumption J (uniform hashing 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 \(\sim \sqrt{\pi \frac{M}{2}}\) tosses.

Coupon collector. Expect every bin has \(\approx 1\) ball after \(\sim M \ln M\) tosses.

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

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

Collision. Two distinct keys hashing to same index.
- Birthday problem ⇒ can’t avoid collisions unless you have a ridiculous amount (quadratic) of memory.
- Coupon collector + load balancing ⇒ collisions will be evenly distributed.

Challenge. Deal with collisions efficiently.

Separate chaining ST

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: only need to search $i$th chain.

Separate chaining ST: Java implementation

```java
public class SeparateChainingHashST<Key, Value> {
    private int N; // number of key-value pairs
    private int M; // hash table size
    private SequentialSearchST<Key, Value>[] st; // array of STs

    public SeparateChainingHashST() {
        this(997);
    }

    public SeparateChainingHashST(int M) {
        this.M = M;
        st = (SequentialSearchST<Key, Value>[]) new SequentialSearchST[M];
        for (int i = 0; i < M; i++)
            st[i] = new SequentialSearchST<Key, Value>();
    }

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

    public Value get(Key key) {
        return st[hash(key)].get(key); }

    public void put(Key key, Value val) {
        st[hash(key)].put(key, val); }
}
```

Analysis of separate chaining

Proposition K. Under uniform hashing assumption, probability 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.

Consequence. Number of probes for search/insert is proportional to $N/M$.
- $M$ too large ⇒ too many empty chains.
- $M$ too small ⇒ chains too long.
- Typical choice: $M \sim N/5$ ⇒ constant-time ops.
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.

Use an array of size $M > N$.
- **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.

Linear probing

- **Linear probing**
- **Use an array of size $M > N$.**
- **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.

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

 probes sequence wraps to 0

Linear probing ST implementation

```java
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 */ }

    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;
    }

    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;
    }
}
```

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, ...

**Q.** What is mean displacement of a car?

**Empty.** With M/2 cars, mean displacement is $\sim 3/2$.

**Full.** With M cars, mean displacement is $\sim \sqrt{\pi \cdot M} / 8$

Analysis of linear probing

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

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


**Parameters.**
- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N/M \sim \frac{1}{2}$.
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.

Diversion: one-way hash functions

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

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

Applications. Digital fingerprint, message digest, storing passwords.
Caveat. Too expensive for use in ST implementations.
Separate chaining vs. linear probing

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

Linear probing.
• Less wasted space.
• Better cache performance.

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. (separate chaining variant)
• Hash to two positions, put key in shorter of the two chains.
• Reduces average 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.

Hashing vs. balanced trees

Hashing.
• 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).

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

Java system includes both.
• Red-black trees: java.util.TreeMap, java.util.TreeSet.
• Hashing: java.util.HashMap, java.util.IdentityHashMap.