CS171 Introduction to Computer Science II

Hash Tables
Symbol table implementations

- Sequential search using linked list
- Binary search using ordered arrays
- Binary search tree (BST)
- Balance search tree (future class)
## ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search (linked list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.38 lg N</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>
</tr>
</tbody>
</table>
Hashing - motivation

• If keys are small integers, can use array for symbol table and use key as an array index

• What if keys are large integers?
  – (SSN, Name) search
  – Use an array and use SSN as array index if no space limitations

• What if keys are not integers?
  – (Word, definition) search

• Hashing: mapping keys to indexes (small integers)
Hashing

- **Hash table**
  - Saves items in an indexed array (table) of size $M$

- **Hashing function**
  - Transforms a key into an index of $[0, M-1]$

- **Collision-resolution**
  - Handles the case when two or more keys hash to the same index

- **Space-time tradeoff**
  - No space limitation: key as index, no collision
  - No time limitation: single index, sequential search
  - Space and time limitations: hashing

---

Hashing: the crux of the problem
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.
Hash Functions

• A hash function is usually specified as the composition of two functions:

Hash code:
\( h_1 \): keys \( \rightarrow \) integers

Compression function:
\( h_2 \): integers \( \rightarrow [0, M - 1] \)

• The hash code is applied first, and the compression function is applied next on the result:
\( h(x) = h_2(h_1(x)) \)
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 \( x \).

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

$$3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$$
$$= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$$
Implementing hash code: user-defined types

```java
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**
- **for reference types, use `hashCode()`**
- **for primitive types, use `hashCode()` of wrapper type**
- **typically a small prime**
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 an array, apply to each element.
- If field is a reference type, use `hashCode()` or use `Arrays.deepHashCode()` applies 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.
Compression Function

• **Modular hashing:**
  - \( k \% M \)
  - dispersing the keys evenly between 0 and \( M-1 \)
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).

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

**Wrong:**

```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() & 0x7fffffff) % M;  
}
```

**Correct**
Collision Handling

- Separate Chaining
- Linear Probing
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.
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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>S</th>
<th>H</th>
<th></th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>R</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

**insert I**

hash(I) = 11

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>S</th>
<th>H</th>
<th></th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>R</th>
<th>I</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

**insert N**

hash(N) = 8
### Linear probing: trace of standard indexing client

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

- **entries in red** are new
- **entries in gray** are untouched
- **keys in black** are probes
- **probe sequence wraps to 0**
- **keys[]**
- **vals[]**
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;
    }
}
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.
## ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search (linked list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.38 lg N</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>
</tr>
<tr>
<td>separate chaining</td>
<td>lg N *</td>
<td>lg N *</td>
<td>lg N *</td>
<td>3-5 *</td>
</tr>
<tr>
<td>linear probing</td>
<td>lg N *</td>
<td>lg N *</td>
<td>lg N *</td>
<td>3-5 *</td>
</tr>
</tbody>
</table>

* under uniform hashing assumption
Hashing vs. balanced search 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 search trees.
- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement `compareTo()` correctly than `equals()` and `hashCode()`.

Java system includes both.