CMSC 132: Object-Oriented Programming II

Hash Tables
Key Value Map

- Red Black Tree: $O(\log n)$
- BST: $O(n)$
- 2-3-4 Tree: $O(\log n)$

- Can we do better?
Hash Tables

- a data structure used to implement (a dictionary) an associative array, a structure that can map keys to values.

- \(O(1)\) best case. (Or average case).

- \(O(n)\) worst case. extremely unlikely to arise by chance, but a malicious adversary with knowledge of the hash function may be able to supply information to a hash that creates worst-case behavior by causing excessive collisions, resulting in very poor performance.
Hash Table Example

• A hash table uses a magic hash function to compute an index into an array slots.
• An object can be found from the array using the index.

\[ H(K) = K \% 13 \]

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

Average Search Times:
Hash Table Example

- A hash table uses a magic hash function to compute an index into an array slots.
- An object can be found from the array using the index.

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>91</td>
<td>66</td>
<td>80</td>
<td>93</td>
<td>95</td>
<td>96</td>
<td>82</td>
<td>85</td>
<td>87</td>
<td>88</td>
<td>63</td>
<td>77</td>
</tr>
</tbody>
</table>

H(K) = K % 13

Grades: 85, 91, 66, 96, 80, 88, 95, 87, 77, 63, 93, 82

To find a number:
85, 91, 66, 96, 80, 88, 95, 87, 77, 63 need just 1 comparison,
93 needs 2 comparison, 82 needs 3 comparison

Average Search Times: \((10 \ast 1 + 1 \ast 2 + 1 \ast 3) / 12 = 1.25\)
Hash Function

- a “magic function” that, given a value to search for, would tell us exactly where in the array to look
  - If it’s in that location, it’s in the array
  - If it’s not in that location, it’s not in the array
- When applied to an Object, returns a number
- When applied to equal Objects, returns the same number for each
- When applied to unequal Objects, is very unlikely to return the same number for each
- Hash functions is very important for searching.
Hash Functions

• Save items in a key-indexed table
• Array index is the function of the key

• Hash function: Method for computing array index from key

• Example:
  • Items are social security numbers
  • Hash function: \( H(\text{Key}) = \text{Key} \mod 100 \)
Hash Functions

- 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 index
Hash Functions

• Classic space-time tradeoff
  • No space limit: trivial hash function with key as index
  • No time limit: trivial collision resolution with sequential search
  • Space and time limit: hashing in 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.

- 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.
Java’s hash code conventions

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

• Requirement:
  \[
  \text{If } x \text{.equals}(y) \text{ then } (x \text{.hashCode()} == y \text{.hashCode()}) .
  \]

• Highly desirable:
  \[
  \text{If } !x \text{.equals}(y) \text{ then } (x \text{.hashCode()} \neq y \text{.hashCode()})
  \]
Java’s Hash Code Conventions

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

final class Double{
    private final double value;
    ...
    public int hashCode(){
        long bits = doubleToLongBits(value);
        return (int) (bits ^ (bits >>> 32));
    }
}
public final class Boolean{
    private final boolean value;
    ...
    public int hashCode() {
        if (value)
            return 1231;
        else
            return 1237;
    }
}
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;
    }
}

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

3045982 = 99\cdot31^3 + 97\cdot31^2 + 108\cdot31^1 + 108\cdot31^0
= 108 + 31\cdot (108 + 31 \cdot (97 + 31 \cdot (99)))
(Horner's method)
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;
    }
}
```
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()`.
  - If field is an array, apply to each entry.

- In practice:
  - Recipe works reasonably well; used in Java libraries.
  - In theory. Keys are bitstring; "universal" hash functions exist.
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.
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.

![Diagram showing open addressing with linear probing](image)

linear probing (M = 30001, N = 15000)
Linear probing: Example

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\hline
\text{st[]} & & & & & & & & & & & & & & & \\
M = 16
\end{array}
\]
Clustering

- Cluster. A contiguous block of items.
- Observation. New keys likely to hash into middle of big clusters.
- Solutions:
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.
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).

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`.
Quiz 1

What is the time complexity to retrieve from a hash table if there are no collisions?

A. $O(1)$
B. $O(n)$
C. $O(n^2)$
D. $O(\log n)$
Quiz 1

What is the time complexity to retrieve from a hash table if there are no collisions?

A. $O(1)$
B. $O(n)$
C. $O(n^2)$
D. $O(\log n)$
Quiz 2

A hash function should have which properties?

A. Uniform distribution
B. Efficient hash code computation
C. Range is a subset of the integers
D. Equivalent objects produce equal hash codes
Quiz 2

A hash function should have which properties?

A. Uniform distribution
B. Efficient hash code computation
C. Range is a subset of the integers
D. Equivalent objects produce equal hash codes
Quiz 3

A hash table of length 10 uses open addressing with hash function \( h(k) = k \mod 10 \), and linear probing. After inserting 6 values into an empty hash table, the table is as shown below. Which one of the following choices gives a possible order in which the key values could have been inserted in the table?

A. 46, 42, 34, 52, 23, 33
B. 34, 42, 23, 52, 33, 46
C. 46, 34, 42, 23, 52, 33
D. 42, 46, 33, 23, 34, 52
Quiz 3

A hash table of length 10 uses open addressing with hash function \( h(k) = k \mod 10 \), and linear probing. After inserting 6 values into an empty hash table, the table is as shown below. Which one of the following choices gives a possible order in which the key values could have been inserted in the table?

A. 46, 42, 34, 52, 23, 33  
B. 34, 42, 23, 52, 33, 46  
C. **46, 34, 42, 23, 52, 33**  
D. 42, 46, 33, 23, 34, 52
Quiz 4

The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function $h(k) = k \text{ mod } 10$ and linear probing. What is the resultant hash table?

(A)  

(B)  

(C)  

(D)
Quiz 4

The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function \( h(k) = k \mod 10 \) and linear probing. What is the resultant hash table?

(A)  
(B)  
(C)  
(D)  

(C)
Quiz 5

Hash table of size seven, with starting index zero, and a hash function \((3x + 4) \mod 7\). Keys 1, 3, 8, 10 are inserted into an empty table.

Which of the following is the contents of the table when?

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
<td>3</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>8</td>
<td>10</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>10</td>
<td>8</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>10</td>
<td>8</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Quiz 5

Hash table of size seven, with starting index zero, and a hash function \((3x + 4) \mod 7\). Keys 1, 3, 8, 10 are inserted into an empty table.

Which of the following is the contents of the table when?

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
<td>3</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>8</td>
<td>10</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>10</td>
<td>8</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>10</td>
<td>8</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Quiz 6

Hash table keys are ordered.

A. True
B. False
Quiz 6

Hash table keys are ordered.

A. True
B. False
What is the worst case time complexity to retrieve from a hash?

A. $O(1)$  
B. $O(n)$  
C. $O(n^2)$  
D. $O(\log n)$
Quiz 7

What is the worst case time complexity to retrieve from a hash?

A. $O(1)$
B. $O(n)$
C. $O(n^2)$
D. $O(\log n)$