Data Structures and Algorithms in Java

Searching: Hash Tables

Outline

Hashing

Hashing

The basic idea is to save items in a key-indexed array, where the index is a function of the key

Hashing

The basic idea is to save items in a key-indexed array, where the index is a function of the key

Hash function provides a method for computing an array index from a key

Hashing

The basic idea is to save items in a key-indexed array, where the index is a function of the key

Hash function provides a method for computing an array index from a key

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

Hashing

The basic idea is to save items in a key-indexed array, where the index is a function of the key

Hash function provides a method for computing an array index from a key

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
- Space and time limitations: hashing (the real world)

Hashing

Hashing

Idealistic goal: scramble the keys uniformly to produce a table index that is

- Efficiently computable
- Equally likely for each key

Hashing

Idealistic goal: scramble the keys uniformly to produce a table index that is

- Efficiently computable
- Equally likely for each key

Example 1: phone numbers

- Bad: first three digits
- Better: last three digits

Hashing

Idealistic goal: scramble the keys uniformly to produce a table index that is

- Efficiently computable
- Equally likely for each key

Example 1: phone numbers

- Bad: first three digits
- Better: last three digits

Example 2: social security numbers

- Bad: first three digits
- Better: last four digits

Hashing

Idealistic goal: scramble the keys uniformly to produce a table index that is

- Efficiently computable
- Equally likely for each key

Example 1: phone numbers

- Bad: first three digits
- Better: last three digits

Example 2: social security numbers

- Bad: first three digits
- Better: last four digits

Practical challenge: need different approach for each type of key

Hashing

Hashing

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: return 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

Hashing

Hashing

Java library implementations

```java
public final class Boolean {
    private final boolean value;

    public int hashCode() { return value ? 1231 : 1237; }
}
```

Hashing

Java library implementations

```java
public final class Boolean {
    private final boolean value;

    public int hashCode() { return value ? 1231 : 1237; }
}
```

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

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

Hashing

Java library implementations

```
public final class Boolean {
    private final boolean value;

    public int hashCode() { return value ? 1231 : 1237; }
}
```

```
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));
    }
}
```

Hashing

Java library implementations

```java
public final class Boolean {
    private final boolean value;

    public int hashCode() { return value ? 1231 : 1237; }
}
```

```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));
    }
}
```

```java
public final class String {
    private int hash = 0;
    private final char[] s;

    public int hashCode() {
        if (hash != 0) { return hash; }
        for (int i = 0; i < length(); i++) { hash = s[i] + (31 * hash); }
        return hash;
    }
}
```

Hashing

Hashing

Implementing hash code for user-defined types

```
public final class Transaction implements Comparable<Transaction> {
    private final String who;
    private final Date when;
    private final double amount;

    public int hashCode() {
        int hash = 17;
        hash = 31 * hash + who.hashCode();
        hash = 31 * hash + when.hashCode();
        hash = 31 * hash + ((Double) amount).hashCode();
        return hash;
    }
}
```

Hashing

Implementing hash code for user-defined types

```java
public final class Transaction implements Comparable<Transaction> {
    private final String who;
    private final Date when;
    private final double amount;

    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

- 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

Hashing

Hashing

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)

Hashing

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)

Implementation

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

Hashing

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)

Implementation

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

Uniform hashing assumption: each key is equally likely to hash to an integer between 0 and $m-1$

Hashing

Modular hashing

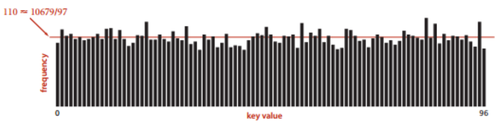
- Hash code: an `int` between -2^{31} and $2^{31} - 1$
- Hash function: an `int` between 0 and $m - 1$ (for use as array index)

Implementation

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

Uniform hashing assumption: each key is equally likely to hash to an integer between 0 and $m - 1$

Example (hash value frequencies for words in Tale of Two Cities; 10,679 keys; $m = 97$)



Hashing

Modular hashing

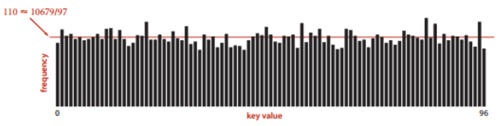
- Hash code: an `int` between -2^{31} and $2^{31}-1$
- Hash function: an `int` between 0 and $m-1$ (for use as array index)

Implementation

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

Uniform hashing assumption: each key is equally likely to hash to an integer between 0 and $m-1$

Example (hash value frequencies for words in Tale of Two Cities; 10,679 keys; $m = 97$)



Collision: two distinct keys hash to the same index

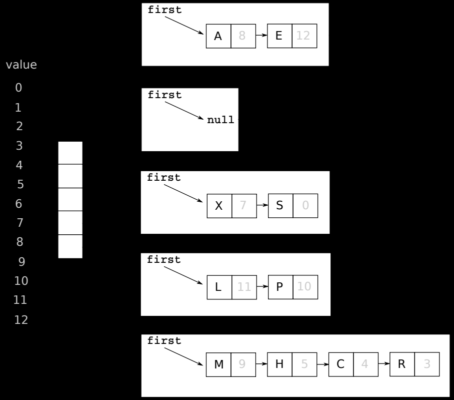
- Can't avoid collisions unless you have a ridiculous amount of memory
- Collisions are evenly distributed
- Challenge: deal with collisions efficiently

Separate-Chaining Symbol Table

Separate-Chaining Symbol Table

Use an array of $m < n$ linked lists

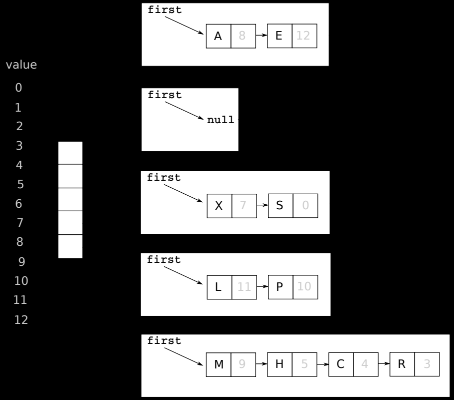
- Hash: map key to integer $i \in [0, m-1]$
- Insert: put at front of ith chain (if not already there)
- Search: need to search only the ith chain



Separate-Chaining Symbol Table

Use an array of $m < n$ linked lists

- Hash: map key to integer $i \in [0, m-1]$
- Insert: put at front of ith chain (if not already there)
- Search: need to search only the ith chain



The ratio n/m is called the load factor and is denoted by α, and is interpreted as the average number of keys per list

Separate-Chaining Symbol Table

Separate-Chaining Symbol Table

Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of α is extremely close to 1

Separate-Chaining Symbol Table

Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of α is extremely close to 1

Consequence: number of probes for search/insert is proportional to α

- m too large $\implies$ too many empty chains
- m too small $\implies$ chains too long

Separate-Chaining Symbol Table

Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of α is extremely close to 1

Consequence: number of probes for search/insert is proportional to α

- m too large $\implies$ too many empty chains
- m too small $\implies$ chains too long

Goal: $\alpha =$ constant

- Double the size of array when $\alpha \geq 10$
- Halve the size of array when $\alpha \leq 2$
- Need to rehash all keys when resizing

Separate-Chaining Symbol Table

Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of α is extremely close to 1

Consequence: number of probes for search/insert is proportional to α

- m too large $\implies$ too many empty chains
- m too small $\implies$ chains too long

Goal: $\alpha =$ constant

- Double the size of array when $\alpha \geq 10$
- Halve the size of array when $\alpha \leq 2$
- Need to rehash all keys when resizing

Deleting a key (and its associated value) is easy — need only consider chain containing key

Separate-Chaining Symbol Table

Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of α is extremely close to 1

Consequence: number of probes for search/insert is proportional to α

- m too large $\implies$ too many empty chains
- m too small $\implies$ chains too long

Goal: $\alpha =$ constant

- Double the size of array when $\alpha \geq 10$
- Halve the size of array when $\alpha \leq 2$
- Need to rehash all keys when resizing

Deleting a key (and its associated value) is easy — need only consider chain containing key

The cost of search, insert, and delete, under the uniform hashing assumption, is constant (between 3 and 5)

Separate-Chaining Symbol Table

Separate-Chaining Symbol Table

</> SeparateChainingHashST.java

```java
package dsa;

import stdlib.StdIn;
import stdlib.StdOut;

public class SeparateChainingHashST<Key, Value> implements BasicST<Key, Value> {
    private LinearSearchST<Key, Value>[] st;
    private int m;
    private int n;

    public SeparateChainingHashST() {
        this(4);
    }

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

    public boolean isEmpty() {
        return size() == 0;
    }

    public int size() {
        return n;
    }

    public void put(Key key, Value value) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        if (value == null) {
```

Separate-Chaining Symbol Table

SeparateChainingHashST.java

```java
            throw new IllegalArgumentException("value is null");
        }
        if (n >= 10 * m) {
            resize(2 * m);
        }
        int i = hash(key);
        if (!st[i].contains(key)) {
            n++;
        }
        st[i].put(key, value);
    }

    public Value get(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        int i = hash(key);
        return st[i].get(key);
    }

    public boolean contains(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        return get(key) != null;
    }

    public void delete(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        int i = hash(key);
        if (st[i].contains(key)) {
            n--;
        }
```

Separate-Chaining Symbol Table

SeparateChainingHashST.java

```java
        st[i].delete(key);
        if (m > 4 && n <= 2 * m) {
            resize(m / 2);
        }
    }

    public Iterable<Key> keys() {
        LinkedQueue<Key> queue = new LinkedQueue<Key>();
        for (LinearSearchST<Key, Value> chain : st) {
            for (Key key : chain.keys()) {
                queue.enqueue(key);
            }
        }
        return queue;
    }

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

    private void resize(int chains) {
        SeparateChainingHashST<Key, Value> temp = new SeparateChainingHashST<Key, Value>(chains);
        for (LinearSearchST<Key, Value> chain : st) {
            for (Key key : chain.keys()) {
                temp.put(key, chain.get(key));
            }
        }
        this.m = temp.m;
        this.n = temp.n;
        this.st = temp.st;
    }

    public static void main(String[] args) {
        SeparateChainingHashST<String, Integer> st = new SeparateChainingHashST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
```

Separate-Chaining Symbol Table

SeparateChainingHashST.java

```java
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys()) {
            StdOut.println(s + " " + st.get(s));
        }
    }
}
```