Hash tables

Gagnaskipan

Hjalti Magnússon (hjaltim@ru.is)

HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY

26. febrúar 2015

Implementations of symbol table

Method	**Get**	**Add**	**Remove**
Key-indexed array	$O(1)$	$O(1)$	$O(1)$
Unordered vector	$O(n)$	$O(1)$	$O(n)$
Unordered list	$O(n)$	$O(1)$	$O(n)$
Ordered list	$O(n)$	$O(n)$	$O(n)$
Binary search	$O(\log(n))$	$O(n)$	$O(n)$
BST (Average)	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
BST (WC)	$O(n)$	$O(n)$	$O(n)$

Implementations of symbol table

Method	Get	Add	Remove
Key-indexed array	$O(1)$	$O(1)$	$O(1)$
Unordered vector	$O(n)$	$O(1)$	$O(n)$
Unordered list	$O(n)$	$O(1)$	$O(n)$
Ordered list	$O(n)$	$O(n)$	$O(n)$
Binary search	$O(\log(n))$	$O(n)$	$O(n)$
BST (Average)	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
BST (WC)	$O(n)$	$O(n)$	$O(n)$

Hashing – The major players

- The *universe* of elements, U
  - The set of all integers (2^{32})
  - The set of all floating point numbers ($\sim 2^{64}$)
  - Points in the integer lattice (2^{64})
  - The set of all strings
    - More strings of length 40 than there are atoms in the universe
- The *hash function*, h
  - Maps elements of U to M *buckets*
- The *hash map* H is an array of element from U (indexed by integers)
  - If $u \in U$, $h(u)$ gives the index of the element u in H

Hashing

Hash function

A hash function $h : U \to [0, M-1]$, is function that maps data of variable length to data of fixed length.

Hash function

A function that turns a (long) sequence of bits into a fixed length sequence of bits.

What is a good hash function?

- Simplicity
  - Lines of code (number of instructions)
- Speed
  - CPU benchmarks
- Strength
  - Hard to measure
  - Few *collisions*
  - Good distribution
  - Avalanche condition
- Security?

Hashing strings

- All hashing can be reduced to hashing strings
- Reduce a sequence of bytes (8-bit blocks) to 32 or 64 bits (possibly more).

Idea 1

```
int hash(const char *str, int size) {
    return size;
}
```

Idea 1

```
int hash(const char *str, int size) {
    return size;
}
```

- Not good!

Idea 1

```
int hash(const char *str, int size) {
    return size;
}
```

- Not good!
- Seriously, not good!

Idea 1

```
int hash(const char *str, int size) {
    return size;
}
```

- Not good!
- Seriously, not good!
- DO NOT USE!

Idea 1

```
int hash(const char *str, int size) {
    return size;
}
```

- Not good!
- Seriously, not good!
- DO NOT USE!
- Once used by PHP

Idea 2 – LoseLose

```
int hash(const char *str, int size) {
    int hash = 0;
    for(int i = 0; i < size; i++) {
        hash += str[i];
    }
    return hash;
}
```

Idea 2 – LoseLose

```
int hash(const char *str, int size) {
    int hash = 0;
    for(int i = 0; i < size; i++) {
        hash += str[i];
    }
    return hash;
}
```

- A little bit better

Idea 2 – LoseLose

```
int hash(const char *str, int size) {
    int hash = 0;
    for(int i = 0; i < size; i++) {
        hash += str[i];
    }
    return hash;
}
```

- A little bit better
- Still terrible

Idea 2 – LoseLose

```
int hash(const char *str, int size) {
    int hash = 0;
    for(int i = 0; i < size; i++) {
        hash += str[i];
    }
    return hash;
}
```

- A little bit better
- Still terrible
- Appeared in *The C Programming Language*

Idea 3 – djb2

```
int hash(const char *str, int size) {
    int hash = 5381;
    for(int i = 0; i < size; i++) {
        hash = hash * 33 + s[i];
    }
    return hash;
}
```

$$h(w_n \dots w_1) = 5381 \cdot 33^n + w_1 \cdot 33^{n-1} + w_2 \cdot 33^{n-2} + \cdots + w_n$$

Idea 3 – djb2

```
int hash(const char *str, int size) {
    int hash = 5381;
    for(int i = 0; i < size; i++) {
        hash = hash * 33 + s[i];
    }
    return hash;
}
```

$$h(w_n \dots w_1) = 5381 \cdot 33^n + w_1 \cdot 33^{n-1} + w_2 \cdot 33^{n-2} + \cdots + w_n$$

- Created by Daniel J. Bernstein

Idea 3 – djb2

```
int hash(const char *str, int size) {
    int hash = 5381;
    for(int i = 0; i < size; i++) {
        hash = hash * 33 + s[i];
    }
    return hash;
}
```

$$h(w_n \dots w_1) = 5381 \cdot 33^n + w_1 \cdot 33^{n-1} + w_2 \cdot 33^{n-2} + \cdots + w_n$$

- Created by Daniel J. Bernstein
- Efficient

Idea 3 – djb2

```
int hash(const char *str, int size) {
    int hash = 5381;
    for(int i = 0; i < size; i++) {
        hash = hash * 33 + s[i];
    }
    return hash;
}
```

$$h(w_n \dots w_1) = 5381 \cdot 33^n + w_1 \cdot 33^{n-1} + w_2 \cdot 33^{n-2} + \cdots + w_n$$

- Created by Daniel J. Bernstein
- Efficient
- Used by Java and C# until recently

Idea 3 – djb2

```
int hash(const char *str, int size) {
    int hash = 5381;
    for(int i = 0; i < size; i++) {
        hash = hash * 33 + s[i];
    }
    return hash;
}
```

$$h(w_n \dots w_1) = 5381 \cdot 33^n + w_1 \cdot 33^{n-1} + w_2 \cdot 33^{n-2} + \cdots + w_n$$

- Created by Daniel J. Bernstein
- Efficient
- Used by Java and C# until recently
- (Susceptible to attacks)

Modern hashes

- MurmurHash
  - Currently used by Java
- lookup3
- One-at-a-Time Hash
- FNV
- SuperFastHash

MurmurHash

```
int MurmurHash2(char *str, int len, int seed) {
    int m = 0x5bd1e995;
    int r = 24; int h = seed ^ len;
    while(len >= 4) { int k = *str;
        k *= m; k ^= k >> r; k *= m;
        h *= m; h ^= k; str += 4; len -= 4;
    }
    switch(len) {
        case 3: h ^= data[2] << 16;
        case 2: h ^= data[1] << 8;
        case 1: h ^= data[0];
                h *= m;
    }
    h ^= h >> 13; h *= m; h ^= h >> 15;
    return h;
}
```

Hash table

- Create an array, H, of size M
- H is called a *hash map*
- Store $u \in U$ in position $h(u)$ in A

Collisions

- A *collision* is when
  - $u, v \in U$,
  - $u \neq v$, and
  - $h(u) = h(v)$
- Resolving collisions
  - Closed hashing (open addressing)
  - Open hashing (separate chaining)
  - Cuckoo hashing