Remember This List?

- How should we resolve collisions?
- What should the table size be?
- What should the hash function be?
- How well does hashing work in the real world?
  - We’ll see a case study today!

Hashing Dilemma

Suppose your WorstEnemy 1) knows your hash function, 2) gets to decide which keys to send you?

Faced with this enticing possibility, WorstEnemy decides to:
- Send you keys which maximize collisions for your hash function.
- Take a nap.

Moral: No single hash function can protect you!

Faced with this dilemma, you:
- Give up and use a linked list for your Dictionary.
- Drop out of software, and choose a career in fast foods.
- Run and hide.
- Proceed to the next slide, in hope of a better alternative.

Universal Hashing

Suppose we have a set $K$ of possible keys, and a finite set $H$ of hash functions that map keys to entries in a hashtable of size $m$.

**Definition:**
$H$ is a universal collection of hash functions if and only if …
For any two keys $k_1, k_2$ in $K$, there are at most $|H|/m$ functions in $H$ for which $h(k_1) = h(k_2)$.

- So ... if we randomly choose a hash function from $H$, our chances of collision are no more than if we get to choose hash table entries at random!

Random Hashing – Not!

How can we “randomly choose a hash function”?
- Certainly we cannot randomly choose hash functions at runtime, interspersed amongst the inserts, finds, deletes! *Why not?*
- We can, however, randomly choose a hash function each time we initialize a new hashtable.

Conclusions:
- WorstEnemy never knows which hash function we will choose — neither do we!
- No single input (set of keys) can always evoke worst-case behavior

Good Hashing:
Universal Hash Function $A$ (UHF$_A$)

Parameterized by prime table size and vector of $r$ integers:
- $a = \langle a_1, \ldots, a_r \rangle$ where $0 \leq a_i < \text{size}$

Represent each key as a vector $k$ of $r$ integers, where $k_i < \text{size}$
- $\text{size} = 11$, key $39752$ => $\langle 3, 9, 7, 5, 2 \rangle$
- $\text{size} = 29$, key “hello world” => $\langle 8, 5, 12, 15, 23, 18, 12, 4 \rangle$

$$h_A(k) = \left( \sum_{i=0}^{r} a_i k_i \right) \mod \text{size}$$
**UHF\(_a\): Example**

- Context: hash strings of length 3 in a table of size 131

  let a = \(<35, 100, 21>\)
  \(h_a(“xyz”) = (35*120 + 100*121 + 21*122) \mod 131 \)
  = 129

  Let b = \(<25, 90, 83>\)
  \(h_b(“xyz”) = (25*120 + 90*121 + 83*122) \mod 131 \)
  = 43

**Thinking about UHF\(_a\)**

**Strengths:**
- Works on any type as long as you can map keys to vectors
- If we’re building a static table, we can try many values of the hash vector \(<a>\)
- Random \(<a>\) has guaranteed good properties no matter what we’re hashing

**Weaknesses:**
- Must choose prime table size larger than any \(k_i\)

---

**Good Hashing:**
Universal Hash Function B (UHF\(_b\))

Parameterized by \(j, a, \) and \(b:\)
- \(j \cdot \text{size}\) should fit into an int.
- \(a\) and \(b\) must be less than \(\text{size}\)

\[h_{j,a,b}(k) = \left((ak + b) \mod (j\cdot\text{size})\right)/j\]

**UHF\(_b\): Example**

Context: hash integers in a table of size 160

Let \(j = 32, a = 13, b = 142\)
\(h_{j,a,b}(1000) = (\left((13\cdot1000 + 142) \mod (32\cdot160)\right)/32 \)
= \((13142 \mod 5120)/32 \)
= 2902 / 32
= 90

Let \(j = 31, a = 82, b = 112\)
\(h_{j,a,b}(1000) = (\left((82\cdot1000 + 112) \mod (31\cdot160)\right)/31 \)
= \((82112 \mod 4960)/31 \)
= 2752 / 31
= 89

**Thinking about UHF\(_b\)**

**Strengths**
- If we’re building a static table, we can try many parameter values
- Random \(a, b\) has guaranteed good properties no matter what we’re hashing
- Can choose any size table
- Very efficient if \(j\) and \(\text{size}\) are powers of 2 - why?

**Weaknesses**
- Need to turn non-integer keys into integers

---

**Perfect Hashing**

When we know the entire key set in advance …

- Examples: programming language keywords, CD-ROM file list, spelling dictionary, etc.

… then perfect hashing lets us achieve:
- Worst-case \(O(1)\) time complexity!
- Worst-case \(O(n)\) space complexity!
Perfect Hashing Technique

- Static set of \( n \) known keys
- Separate chaining, two-level hash
- Primary hash table size = \( n \)
- \( j \)th secondary hash table size = \( n_j^2 \)
  (where \( n_j \) keys hash to slot \( j \) in primary hash table)
- Universal hash functions in all hash tables
- Conduct (a few) random trials, until we get collision-free hash functions

Perfect Hashing Theorems

- Theorem: If we store \( n \) keys in a hash table of size \( n \) using a randomly chosen universal hash function, then the probability of any collision is < \( \frac{1}{2} \).
- Theorem: If we store \( n \) keys in a hash table of size \( m = n \) using a randomly chosen universal hash function, then
  \[ E \left( \sum_{j=1}^{m} n_j \right) < 2n \]
  where \( n_j \) is the number of keys hashing to slot \( j \).
- Corollary: If we store \( n \) keys in a hash table of size \( m = n \) using a randomly chosen universal hash function and we set the size of each secondary hash table to \( m_j = n_j^2 \), then:
  a) The probability that the total storage used for all secondary hash tables exceeds \( 2n \) is less than \( \frac{1}{2} \).
  b) The expected amount of storage required for all secondary hash tables is less than \( 2n \).

Perfect Hashing Conclusions

Perfect hashing theorems set tight expected bounds on sizes and collision behavior of all the hash tables (primary and all secondaries).

- Conduct a few random trials of universal hash functions, by simply varying UHF parameters, until we get a set of UHFs and associated table sizes which deliver …
  - Worst-case \( O(1) \) time complexity!
  - Worst-case \( O(n) \) space complexity!

Extendible Hashing: Cost of a Database Query

- Cost of a Database Query
  - I/O to CPU ratio is 300-to-1!

Extendible Hashing

Hashing technique for huge data sets
- Optimizes to reduce disk accesses
- Each hash bucket fits on one disk block
- Better than B-Trees if order is not important – why?

Table contains:
- Buckets, each fitting in one disk block, with the data
- A directory that fits in one disk block is used to hash to the correct bucket

Extendible Hash Table

- Directory entry: key prefix (first \( k \) bits) and a pointer to the bucket with all keys starting with its prefix
- Each bucket contains keys matching on first \( j \leq k \) bits, plus the value associated with each key
### Inserting (easy case)

- **insert(11011)**

### Splitting a Leaf

- **insert(10000)**

### Splitting the Directory

1. **insert(10010)**
   - But, no room to insert and no adoption!
2. **Solution:** Expand directory
3. **Then, it's just a normal split.**

### If Extendible Hashing Doesn’t Cut It

- Store only pointers/references to the items: (key, value) pairs are in disk
  - (Potentially) much smaller $M$
  - Fewer items in the directory
  - One extra disk access!

- **Rehash**
  - Potentially better distribution over the buckets
  - Fewer unnecessary items in the directory
  - Can’t solve the problem if there’s simply too much data

- What if these don’t work?
  - Use a B-Tree to store the directory!

### Hash Wrap-up

**Hash function:** maps keys to integers; table size should be prime

**Choosing a Hash Function**

- **Universal hashing**
  - Guarantees no (always) bad input
- **Perfect hashing**
  - Requires known, fixed keyset
  - Achieves $O(1)$ time, $O(n)$ space guaranteed!

**Collision resolution**

- **Separate Chaining**
  - Expand beyond hashtable via secondary Dictionaries
  - Allows $\lambda = 1$
- **Open Addressing**
  - Expand within hashtable
  - Secondary probing (linear, quadratic, double hash)
  - $\lambda \leq 1$ (by definition!)
  - $\lambda \leq 1/2$ (by preference!)

**Rehashing**

- Tunes up hashtable when $\lambda$ crosses the line

### Hash Wrap-up (part 2)

- **Also:** Extendible hashing
  - For disk-based data
  - Combine with B-tree directory if needed
Dictionary ADT Wrapup: Case Study

- Your company, Procrastinators Inc., will release its highly hyped word-processing program, *WordMaster2000* (yeah, they’re a little behind the times), next month.
- Your highly successful alpha-test was marred by user requests for a spell-checker.
- Your mission: write and test a spell-checker module before *WordMaster2000* is released.
- For now, you only need to worry about the English language, although *WordMaster2000* is successful, you may need to port your spell-checker to other languages/character sets.

Case Study: Assumptions

You will be given a spelling dictionary of English words
- 30,000 words
- Static (i.e., does not support adding user-supplied words yet)
- Arbitrary(ish) preprocessing time

Practical notes
- Almost all searches are successful – *Why?*
- Words average about 8 characters in length
- 30,000 words at 8 bytes/word – .25 MB
- There are many regularities in the structure of English words

Case Study: Design Considerations

Issues:
- Which data structure should we use?
- What are our design goals?

Possible Solutions?