Hash Tables

Achieving Expected $O(1)$ Insert/Search/Delete

Achieving Expected $O(1)$ Insert/Delete/Search

- Linked data structures (linked lists, trees) require traversal, which will exceed $O(1)$.
- Only way to achieve $O(1)$ would be to utilize “random access” store, with ability to calculate the location for any key in constant time
  - Indexed arrays are the only random access store we have
  - How to map keys to array indices becomes critical
- Already mentioned “key”
  - We assume we store “key”-”value” pairs, with all keys distinct
  - “Dictionary”, “Map”, “Table”: Refer to same key-value store

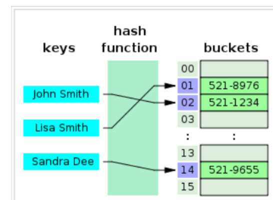
Mapping Keys To Indices

- If keys are integer numbers between 0 and $m-1$ only,
  - We can use those keys as indices to the underlying array store.
  - This is what’s referred to as “direct-address tables” in CLRS 11.1.
- Insert/Search/Delete are guaranteed to be $O(1)$ in all cases.
- But not realistic
  - Keys are mostly not integers.
    - Mapping arbitrary key values to integers in $[0, m-1)$ is nontrivial.
  - m could be really big, and only very small number of keys might be in use.
    - Big waste of maintaining the array of unnecessarily big size m, or it’s even impossible.

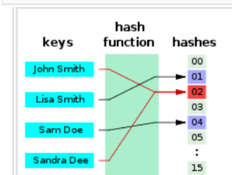
Hashing

http://en.wikipedia.org/wiki/Hash_table

- Given
  - U: Set of all possible keys (universe)
  - $T[0..m-1]$: Hash table of size m
- Hash function
  - $h: U \rightarrow \{0, 1, \ldots, m-1\}$
- Collision
  - Unavoidable if $|U| > m$ and more than m items are inserted
    - Pigeonhole principle

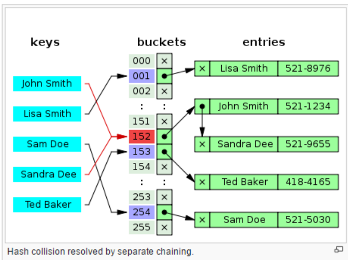


A small phone book as a hash table.

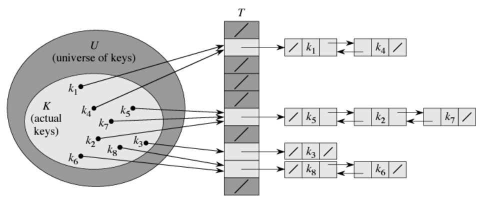


A hash function that maps names to integers from 0 to 15.. There is a collision between keys "John Smith" and "Sandra Dee".

Collision Resolution By Chaining



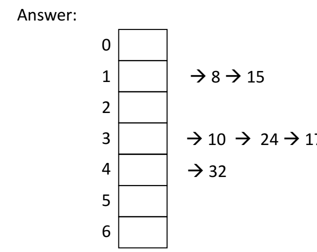
http://en.wikipedia.org/wiki/Hash_table



CLRS Figure 11.3, pp.257: Doubly-linked list in order to make delete-by-node operation $O(1)$

Chaining Example

- Keys are nonnegative integers, $h(k) = k \bmod 7, m = 7$
- If following keys are inserted in the order, and chaining is used to resolve collisions, what's the resulting hash table?
  $8, 10, 24, 15, 32, 17$



Analysis of Hash Table Operations When Hashing With Chaining

- Worst case is terrible: $\Theta(n)$, when all elements hash to the same slot.
  - Degenerates into a linked list
- Average-case performance of hashing with chaining
  - "Depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average"
  - "Simple uniform hashing":
    - Hard to express the distribution precisely, so we make assumption:
      - Any given element is equally likely to hash into any of the m slots.
- Load factor $\alpha = \frac{n}{m}$
  - When n is the number of elements stored in the hash table of size m.
  - It's the average number of elements stored in a chain.

When Search Fails

- Expected time to search unsuccessfully for key k is:
  - Expected time to search to the end of list $T[h(k)]$.
  - Equivalent to expected length of the chain $T[h(k)] = \alpha$
  - Plus time to compute hash function $h(k)$: Fixed ($\Theta(1)$)
- Therefore, unsuccessful search is $\Theta(1 + \alpha)$.
  - Theorem 11.1 in CLRS pp.259

When Search Succeeds

- Another probabilistic assumption: Element being search for is equally likely to be any of the n elements stored in the table
- Note elements are pushed to the head of a chain for any collision
  - The number of elements examined to hit element x is 1+# elements that appear before x in x's chain
  - # elements appearing before x in x's chain is # elements hashed to the same slot, but inserted to the table after x
- Quite more involved, need to establish other random variables
  - See CLRS pp.260
  - After careful math to compute expected # elements to examine, we still get $\Theta(1+\alpha)$ even for successful searches.
- This means search in hashing is $O(1)$ if α is bounded by a fixed constant!

Crafting Good Hashing Functions

Not All Hashing Functions Work Well

What Makes A Good Hashing Function?

- Should satisfy (approximately) simple uniform hashing:
  - Each key is equally likely to hash to any of the m slots.
    - Independently of where any other key has hashed to.
  - Unfortunately we typically have no way to check this condition:
    - We can't know probability distribution of the keys.
    - Keys may not be independent to each other.
  - In rare cases when we do know the distribution,
    - E.g., If keys are random real numbers k which are independently and uniformly distributed between 0 (incl.) and 1 (excl.),
    - $h(k) = \lfloor km \rfloor$ does satisfy simple uniform hashing.
  - In practice, some heuristically crafted hash functions work pretty well.
    - Only natural numbers are considered as keys, because any keys can be mapped to (interpreted as) natural numbers.
      - E.g., variable name "pt" mapped to ASCII (112,116), then 112x128+116=14452.

Division Method

- $h(k) = k \bmod m$ (k % m)
  - E.g., 100 mod 12 = 4, because 100 / 12 = 6 in integer, and 100 – 12*6 = 4
  - The remainder of integer division of k by m
- Choice of m is important
  - Avoid a power of 2. In the case of string keys, it's like choosing last (or first) few characters as hash values, which wouldn't be very uniformly distributed.
  - A prime number is usually good choice, but not all primes are good.
    - Exercise 11.3-3: $m = 2^p - 1$ is not good when k is a numeric encoding of character string in radix 2^p (e.g., 2^8 would be 8-bit/1-byte code, …)
  - A prime not too close to an exact power of 2 is often a good choice for m
    - Heuristically concluded.

Multiplication Method

- Multiply natural number k by a fractional real number A $(0 \le A < 1)$, then take the fractional part, then multiply by m, then take the integer part.

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

- Advantage: m can be any number, even a power of 2
- A few more restrictions on A to make the computation efficient.
- Some heuristically found A works pretty well
  - E.g., $A \cong \frac{\sqrt{5}-1}{2} = 0.6180339887 \ldots$

Universal Hashing

- A fixed hash function, if known, can be easily exploited by malicious adversary for worst case degeneration (all keys hashing into same slot)
- Should be able to choose hash function for each usage
- Set of hash functions $\mathcal{H}$ is called universal
  - If for each pair of distinct keys $k, l \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $|\mathcal{H}|/m$.
- A good universal set of hash functions
  - $h_{ab}(k) = \big((ak + b)\ \%\ p\big)\ \%\ m$
    - For a prime p, $a \in \{1,2, \ldots, p-1\}$, $b \in \{0,1,2, \ldots, p-1\}$
  - Study CLRS 11.3.3 why this is universal and why universal hash function gives $O(1 + \alpha)$

Hashing with Open Addressing

When Pointers Are Not Desirable

Open Addressing: Alternatives to Chaining

- There are certain situations where chaining shouldn't be used:
  - Limited memory, no memory allocator, …
- Instead of chaining colliding elements as a linked list,
  - Compute the sequence of slots to be examined.
  - Probe this sequence for any operations.
- Probe sequence depends upon the key being used.
- Hash function should be extended to include probe number:
  - $h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$
  - The probe sequence for key k:
    - $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$: This should be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
      - So that every hash-table position is eventually considered for any key, as table fills up.

Linear Probing

- Given original (called auxiliary) hash function
  $h' : U \to \{0, 1, \dots, m-1\}$,
- $h(k, i) = (h'(k) + i) \mathbin{\%} m$
- E.g., if $m = 7$, $h'(k) = k \mathbin{\%} 7$, and $k = 38$, then the probe sequence should be:
  - First, $h'(38) = 38 \mathbin{\%} 7 = 3$.
  - $h(k, 0) = h'(38) \mathbin{\%} 7 = 3$
  - $h(k, 1) = (h'(38) + 1) \mathbin{\%} 7 = 4$
  - $\therefore \langle 3, 4, 5, 6, 0, 1, 2 \rangle$

- Example: For the hash parameters and functions on the left, draw the resulting hash table when the following keys are inserted in the given order:
  $38, 29, 17, 45, 8, 15$

0	
1	29
2	8
3	38
4	17
5	45
6	15

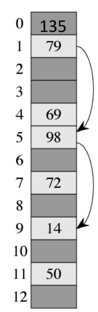
Quadratic Probing

- Linear probing is easy to understand/implement.
- But it suffers from "primary clustering" issue.
  - # probes goes up pretty quickly as α ($= n/m$) approaches 1 due to cluster forming from previous insertions
- Quadratic probing: Use a different hash function to avoid the issue
  - $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mathbin{\%} m$
  - E.g., for the earlier example (k = 38) with $c_1 = 0$ & $c_2 = 1$, the probing sequence will be $\langle 3, 4, 0, 5, 5, 0, 4 \rangle$, which is not a permutation of $\langle 0, 1, 2, 3, 4, 5, 6 \rangle$, thus won't work well for open addressing.
  - In fact, c_1 & c_2 must be chosen carefully to produce permutation for every k.
  - For such good c_1 & c_2, quadratic probing is much better than linear probing.

Double Hashing

- Still clustering on quadratic probing: "secondary clustering"
  - Same probe sequences for two keys if their initial probes are the same.
  - The initial probe fixes the entire sequence, allowing only m permutations.
- Double hashing: Combine 2 hashing functions to allow more perms.
  - $h(k, i) = \big(h_1(k) + i\, h_2(k)\big) \mathbin{\%} m$.
  - It's like linear probing, but the displacement is not fixed to 1, but it varies on k.
  - Easier construction of $h_1(k)$ and $h_2(k)$. E.g., for a prime m,
    - $h_1(k) = k \mathbin{\%} m$
    - $h_2(k) = 1 + (k \mathbin{\%} (m-1))$
  - Double hashing performs close to "ideal" scheme of uniform hashing.

Double Hashing Example (CLRS Fig. 11.5, pp.273)



Q: What is the probe sequence for $k = 135$, and in what slot will the key 135 be inserted?

Probe 0: $h_1(135)$ = 135 mod 13 = 5
Probe 1: 5 + 1*(1+135%11) = 9
Probe 2: 5 + 2*(1+135%11) = 13%13 = 0

Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, we insert the key 14 into empty slot 9, after examining slots 1 and 5 and finding them to be occupied.

Analysis Of Open Addressing

- Study CLRS Theorem 11.6 & 11.8
  - For open addressing's average case performance
  - Using again probability and random variable analysis
  - Note the results are only for "uniform" hashing (ideal case)
    - None of the open addressing schemes we've seen are truly uniform.
    - Double hashing is close to uniform.
  - Theorem 11.6: Expected # probes in unsuccessful search is at most $\frac{1}{1-\alpha}$.
  - Theorem 11.8: Expected # probes in successful search is at most $\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$.

In Practice,

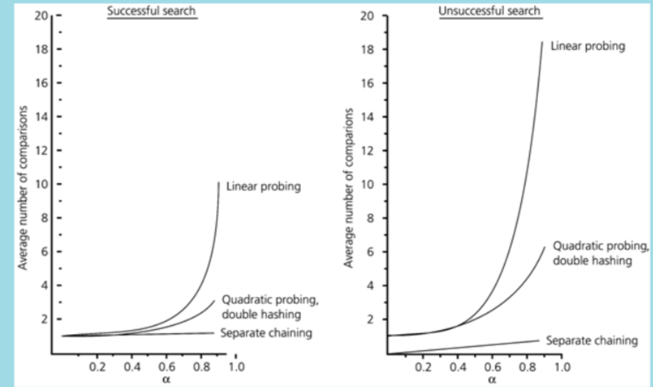


Figure 12-50 The relative efficiency of four collision-resolution methods

From Data Abstraction & Problem Solving with C++: Walls & Mirrors by Carrano

Open Addressing Summary

- Should be used only on limited cases:
  - When load factor is small.
  - When memory is limited (thus has to be saved as much) or dynamic memory is unavailable (no dynamic allocation of pointed nodes).
- Deletion should be also careful.
  - Can't just assign **empty** to the slot.
    - If done this way, then subsequent search for a different value with same hash will fail, when it's still in the table.
  - Must just mark the slot as **deleted**. Search should ignore the deleted mark.
    - Insertion can just overwrite any slot that's marked as deleted.
- Finally, $n \le m$!
  - If more elements are inserted, the entire table must be resized.
    - Meaning all existing elements should be copied, which is a big overhead (rehashing)

Many Variants of Hashing

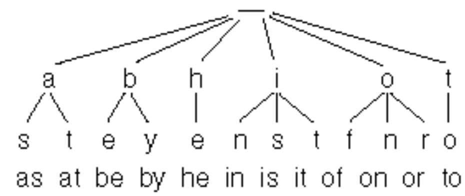
- Perfect Hashing
  - Constructing a hashing function with no collisions (hard problem)
  - Useful for fixed data sets
  - O(1) search time in the worst case
- Cuckoo Hashing (2001)
  - Use two hash tables with two different hashing functions
  - While inserting, if slot 1 in the first table is occupied, we go to second table
  - If the slot 2 is again occupied, kick the element there, and put the new one
  - The kicked element applies the same algorithm…. may never stop!
  - We can use a single table with two hashing functions
  - Insertion is slower (may need resize and rehashing), but search is faster
- Many more ...

Digital or Radix Search

Achieving Worst Case $O(1)$ Insert/Search/Delete

Searching Strings

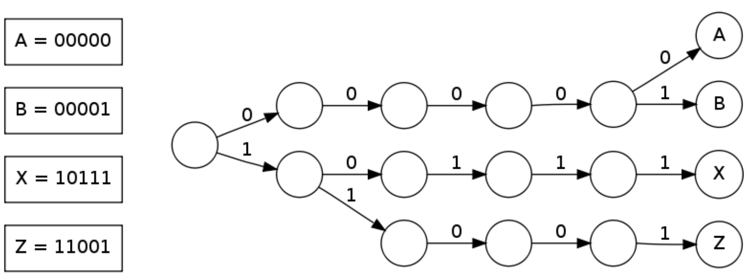
- Instead of comparing numbers we can use letters to guide the search in a *trie* (labels can be in edges or nodes)



- Searching, inserting and deleting in a trie takes time proportional to the length of the string independent on the number of strings!
- Notice that the leaves are lexicographically sorted as labels are sorted

Binary Tries

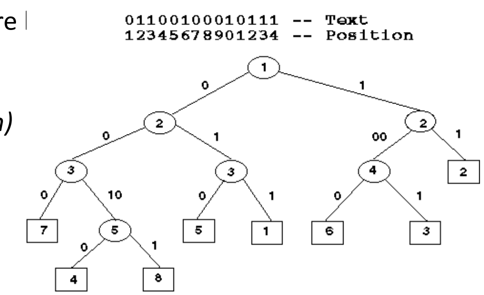
- We do not need edge labels, they are implicit (0 left, 1 right)



- To store words you concatenate the codes of letters
- Space can be larger than *O(n),* the number of words

PATRICIA Trees

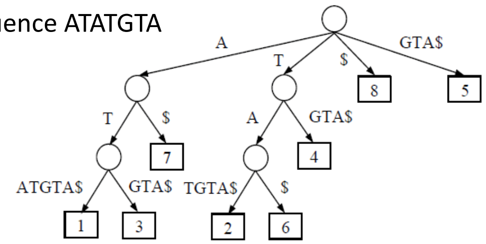
- We do not need to store unary nodes as they do not guide the search
- Hence we can skip them, adding the letter position in each node
- Search is equal as before but we need to check the whole string after
- Space is reduced to *O(n)*



Suffix or PAT trees

- Each suffix of a string is an entry in the tree
- Consider the DNA sequence ATATGTA

ATATGTA$
12345678



- Then we can search any substring of a long string

Suffix Arrays

0	1	2	3	4	5	6	7	8	9	10	11	12	13
t	g	t	g	t	g	t	g	c	a	c	c	g	$

- We keep only the leaves of the suffix tree
- We use less memory and we can search in O(log n) time using two indirect binary searches (≤ & ≥ to the query string)
- Many applications in computational biology

0	13	$
1	9	a c c g $
2	8	c a c c g $
3	10	c c g $
4	11	c g $
5	12	g $
6	7	g c a c c g $
7	5	g t g c a c c g $
8	3	g t g t g c a c c g $
9	1	g t g t g t g c a c c g $
10	6	t g c a c c g $
11	4	t g t g c a c c g $
12	2	t g t g t g c a c c g $
13	0	t g t g t g t g c a c c g $