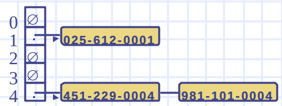
Hash Tables



Hashing:

- A method for directly referencing items in a dictionary by doing arithmetic transformations on keys into dictionary addresses.
- A hush function is perfect if there is no key collision, that is, two keys hash to the same hash value.

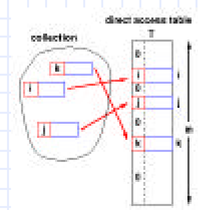
Why Hash Tables?

- All search structures so far
  - Relied on a comparison operation
  - Performance $O(n)$ or $O(\log n)$
- Assume I have a function
  - $f(\, key\,) \rightarrow \; integer$

    ie one that maps a key to an integer
- What performance might I expect now?

Hash Tables - Structure

- Simplest case:
  - Assume items have integer keys in the range $1 \,..\, m$
  - Use the value of the key itself to select a slot in a direct access table in which to store the item
  - To search for an item with key, k, just look in slot k
    - If there's an item there, you've found it
    - If the tag is 0, it's missing.
  - Constant time, $O(1)$



Hash Tables - Constraints

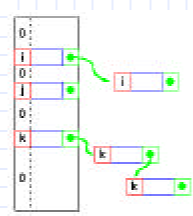
- Constraints
  - Keys must be unique
  - Keys must lie in a small range
  - For storage efficiency, keys must be dense in the range
  - If they're sparse (lots of gaps between values), a lot of space is used to obtain speed
    - Space for speed trade-off

Hash Tables - Relaxing the constraints

- Keys must be unique
  - Construct a linked list of duplicates : "attached" to each slot
  - If a search can be satisfied by *any* item with key, k, performance is still $O(1)$

    but
  - If the item has some other distinguishing feature which must be matched, we get $O(n^{max})$

    where n^{max} is the largest number of duplicates - or length of the longest chain

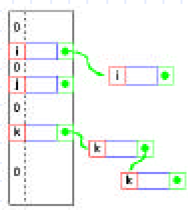


Hash Tables - Relaxing the constraints

- Keys are integers
  - Need a hash function

    h(key) $\rightarrow$ *integer*

    ie one that maps a key to an integer
  - Applying this function to the key produces an address
  - If h maps each key to a *unique integer* in the range $0 .. m\text{-}1$, then search is $O(1)$



An Example: Perfect Hash

- suppose: MagicNumber = 15
- int h(String s) {

```
int h(String s) {
    return ((s[0] + s[1])% MagicNumber);
}
```

- suppose:

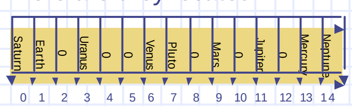
```
typedef struct {
    String name;
    int numMoons;
    double sunDistance;
} planet;

planet solarSystem[MagicNumber];
```


- Suppose:

```
solarSystem[h("Mercury")]  = {"Mercury", 0, 36.0};
solarSystem[h("Venus")]    = {"Venus", 0, 67.27};
solarSystem[h("Earth")]    = {"Earth", 1, 93.0};
solarSystem[h("Mars")]     = {"Mars", 2, 141.71};
solarSystem[h("Jupiter")]  = {"Jupiter", 16, 483.88};
solarSystem[h("Saturn")]   = {"Saturn", 12, 887.14};
solarSystem[h("Uranus")]   = {"Uranus", 5, 1783.98};
solarSystem[h("Neptune")]  = {"Neptune", 2, 2795};
solarSystem[h("Pluto")]    = {"Pluto", 1, 3675};
```

- Where are they located



"Ju" in ASCII are 74 and 117, 74 + 117 = 191;

191 % 15 = 11;

h("Mercury")	=	13
h("Venus")	=	7
h("Earth")	=	1
h("Mars")	=	9
h("Jupiter")	=	11
h("Saturn")	=	0
h("Uranus")	=	4
h("Neptune")	=	14
h("Pluto")	=	8

Thus, our search function is simply:

```
planet search(String s){ return solarSystem[h(s)]; }
```


Hash Functions

- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:

  $h(x) = x \bmod N$

  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x
- The goal of a hash function is to uniformly disperse keys in the range $[0, N-1]$

Choosing the Hash Function

- Uniform hashing
  - Ideal hash function
    - $P(k)$ = probability that a key, k, occurs
    - If there are m slots in our hash table,
    - a uniform hashing function, $h(k)$, would ensure:

$$\sum_{k \mid h(k)=0} P(k) = \sum_{k \mid h(k)=1} P(k) = \; .... \; \sum_{k \mid h(k)=m-1} P(k) = \frac{1}{m}$$

Read as sum over all k such that $h(k) = 0$

- *or, in plain English,*
- the number of keys that map to each slot is equal

Hash Tables - A Uniform Hash Function

- *If* the keys are integers randomly distributed in [0 , r), *then*

Read as $0 \le k < r$

$$h(k) = \left\lfloor \frac{mk}{r} \right\rfloor$$

is a uniform hash function

- Most hashing functions can be made to map the keys to [0 , r) for some r
  - *eg* adding the ASCII codes for characters mod 255 will give values in [0, 256) or [0, 255]
  - Replace + by xor

  ←same range without the mod operation

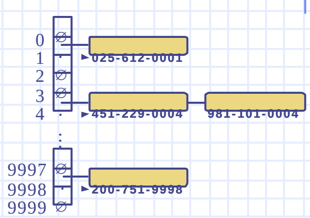
Hash Tables

- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a dictionary with a hash table, the goal is to store item (k, o) at index $i = h(k)$
- A collision occurs when two keys in the dictionary have the same hash value, i.e.,

  h(k) == h(k'), whereas k != k'
- Collision handing schemes:
  - Chaining: colliding items are stored in a sequence
  - Open addressing: the colliding item is placed in a different cell of the table

Example

- We design a hash table for a dictionary storing items (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x) =$ last four digits of x
- We use chaining to handle collisions



Define Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code map:

  h_1: keys $\rightarrow$ integers

  Compression map:

  h_2: integers $\rightarrow [0, N-1]$
- The hash code map is applied first, and the compression map is applied next on the result, i.e.,

  $h(x) = h_2(h_1(x))$
- The goal of the hash function is to "disperse" the keys in an apparently random way

Hash Code Maps

- Memory address:
  - We reinterpret the memory address of the key object as an integer
  - Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float)
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double)

Example: A Hash Function

- Hash function
  - With this hash function

```
int hash( char *s, int n ) {
    int sum = 0;
    while( n-- ) sum = sum + *s++;
    return sum % 256;
}
```

  - **hash("AB", 2)** and **hash("BA", 2)** return the same value!
  - This is called a collision
  - A variety of techniques are used for resolving collisions

Hash Code Maps (cont.)

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
    $a_0\, a_1 \,\ldots\, a_{n-1}$
  - We evaluate the polynomial
    $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$
    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

Hash Code Maps (cont.)

- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time
    $p_0(z) = a_{n-1}$
    $p_i(z) = a_{n-i-1} + z p_{i-1}(z)$
    $(i = 1, 2, \ldots, n-1)$
- We have $p(z) = p_{n-1}(z)$

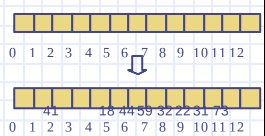
```
int poly(int a[], int z; int n){
    int p = 0;
    for (int i = n-1; i >= 0; i--){
        p = a[i] + z*p;
    }
    return p;
}
```


Compression Maps

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Linear Probing

- Linear probing handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Search with Linear Probing

- Consider a hash table A that uses linear probing
- findElement(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
function findElement(k){
    i = h(k);
    p = 0;
    repeat {
        c = A[i];
        if (c == ∅)
            return NO_SUCH_KEY;
        else if (c.key () == k)
            return c.element()
        else {
            i = (i + 1) mod N;
            p = p + 1;
        }
    until ( p == N);
    return NO_SUCH_KEY;
}
```


Updates with Linear Probing

- To handle insertions and deletions, we introduce a special key flag, called *AVAILABLE*, which replaces deleted elements
- removeElement(k)
  - We search for an item with key k
  - If such an item (k, o) is found, we replace it with the special item *AVAILABLE* and we return element o
  - Else, we return *NO_SUCH_KEY*
- insert Item(k, o)
  - We report an error if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell i is found that is either empty or stores *AVAILABLE*, or
    - N cells have been unsuccessfully probed
  - We store item (k, o) in cell i

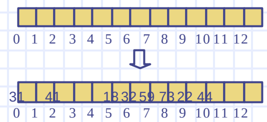
Double Hashing

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
  $(i + jd(k)) \bmod N$
  for $j = 0, 1, \ldots, N-1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells
- Common choice of compression map for the secondary hash function:
  $d_2(k) = q - k \bmod q$
  where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are
  $1, 2, \ldots, q$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Performance of Probing:

- Let N be the number of slots of a hash table, n be the number of items in the table, we define load factor as:
  $\alpha = n/N$
- If the hash function randomly distributes keys through the table, then the expected length of a successful search path is:
  $\text{length}_{succ} = ½ (1 + 1/(1- \alpha))$

Performance of Probing:

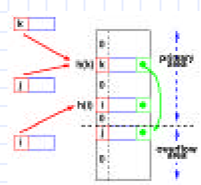
- The expected length of an unsuccessful search is approximately:
  $\text{length}_{unsucc} = ½ (1 + 1/(1 - \alpha)^2)$

Problems with Probing:

- The size of the hash table must be fixed in advance.
- The search costs increase dramatically as the table becomes nearly full.
- Need a special object, called ***AVAILABLE***, to implement “delete” operation.

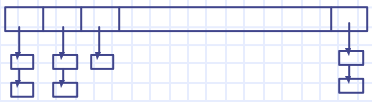
Collision resolution using Overflow area

- Overflow area
  - Linked list constructed in special area of table called overflow area
- h(k) == h(j)
- k stored first
- Adding j
  - Calculate h(j)
  - Find k
  - Get first slot in overflow area
  - Put j in it
  - k's pointer points to this slot
- Searching - same as linked list



Collision resolution using Linked Lists:

- Dynamically allocate space.
- Easy to insert/delete an item
- Need a link for each node in the hash table.



Performance:

- Let N be the size of the hash table, n the number of items in the table's linked lists, if all input sequences are equally likely and the hash function randomly distributes keys over the table, the expected length of a linked list is n/N.

  $length_{succ}$ = ½ (n/N)

  $length_{unsucc}$ = n/N

Collision Resolution Summary

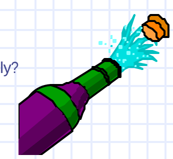
- Chaining
  - + Unlimited number of elements
  - + Unlimited number of collisions
  - - Overhead of multiple linked lists
- Re-hashing
  - + Fast re-hashing
  - + Fast access through use of main table space
  - - Maximum number of elements must be known
  - - Multiple collisions become probable
- Overflow area
  - + Fast access
  - + Collisions don't use primary table space
  - - Two parameters which govern performance need to be estimated

Conclusion:

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the dictionary collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is $1/(1-\alpha)$
- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

Collision Frequency

- Birthdays *or* the von Mises paradox
  - There are 365 days in a normal year
    - ← Birthdays on the same day unlikely?
  - How many people do I need before "it's an even bet" (*ie* the probability is > 50%) that two have the same birthday?
  - View
    - the days of the year as the slots in a hash table
    - the "birthday function" as mapping people to slots
  - Answering von Mises' question answers the question about the probability of collisions in a hash table



Distinct Birthdays

- Let $Q(n)$ = probability that n people have distinct birthdays
- $Q(1) = 1$
- With two people, the 2nd has only 364 "free" birthdays

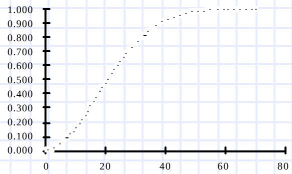
$$Q(2) = Q(1) * \frac{364}{365}$$

- The 3rd has only 363, and so on:

$$Q(n) = Q(1) * \frac{364}{365} * \frac{364}{365} * \ldots * \frac{365-n+1}{365}$$

Coincident Birthdays

- Probability of having two identical birthdays
- $P(n) = 1 - Q(n)$
- $P(23) = 0.507$



- With 23 entries, table is only 23/365 = 6.3% full!

Hash Tables - Load factor

- Collisions are very probable!
- Table load factor

$$a = \frac{n}{m}$$

n = number of items
m = number of slots

must be kept low
- Detailed analyses of the average chain length (or number of comparisons/search) are available
- Separate chaining
  - linked lists attached to each slot gives best performance
  - but uses more space!