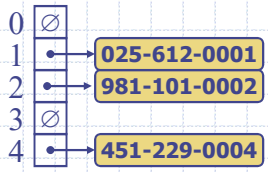
Hash Tables

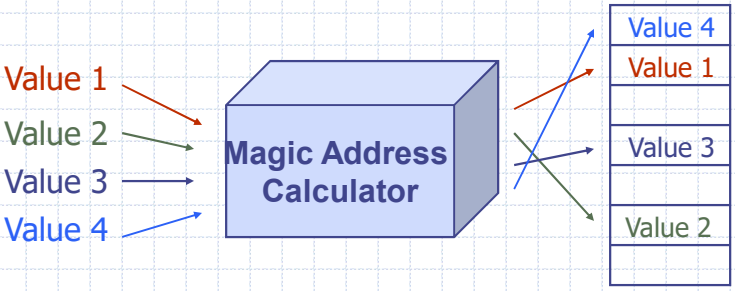


Fast Operations

- Balanced trees perform searches, inserts, and removes in a remarkable amount of time
  - Log 1,000,000 ≈ 13
- Sometimes not fast enough
  - 911 call - search by telephone number
  - Air traffic control - search by flight number

Faster Operations

- What if we could design a ADT that can provide search, insert, and remove in basically O(1) time



Hash Function

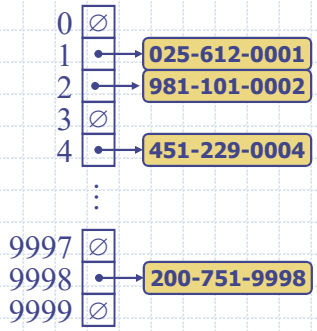
- Address calculator performs hashing using a hash function
  - Hash function is user defined
  - Hash table is typically an array
  - Each element maps into an array position
  - Operates in constant or near constant time
  - Perfect hash function maps each item to a unique table location

Hash Functions and Hash Tables

- 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
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N

Example

- We design a hash table for storing items where a phone number is a ten-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



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.,

  $h(x) = h_2(h_1(x))$

Hash Functions

- Direct addressing - storing items in an array that is the size of the largest key
  - Wastes space if the number of items is small

Hash Functions

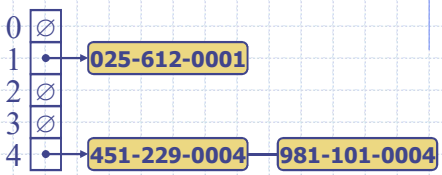
- Good hash function
  - Table size close to the number of items
  - Good distribution of items
  - Easy to compute
  - Ensure any two distinct items get different mappings
  - Deterministic

Perfect Hashing

- Perfect hashing - no two items hash to the same location
- Impossible to give a unique mapping to an infinite number of items stored in a fixed size array
- Function should attempt to distribute items uniformly

Collision Handling

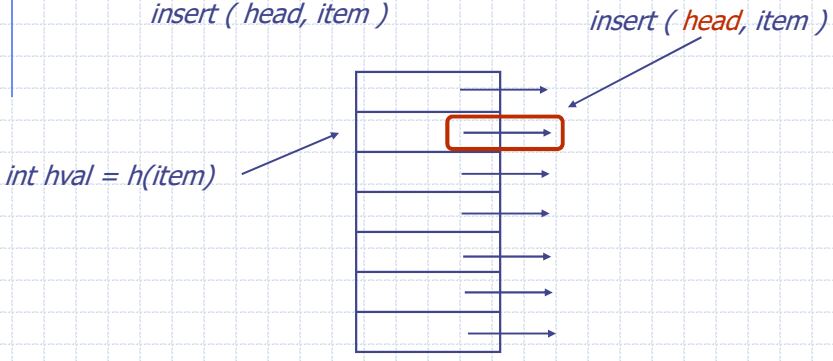
- Collisions occur when different elements are mapped to the same cell
- **Separate Chaining**: let each cell in the table point to a linked list of elements that map there



- Chaining is simple, but requires additional memory outside the table

Separate Chaining

```
void insertChainedHash ( itemtype item )
        int hval = h(item)
        Node *head = hashTable[hval]
        insert ( head, item )
```



Separate Chaining

- In class exercise - hash the following values: 6,45, 754, 34, 4, 66 to a table of size 10 using the following hash function:
  - h(item) = item % SIZE
- Use push back to insert into the chain

Separate Chaining

Separate Chaining

```
Node * searchChainedHash ( itemtype item )
    int hval = h ( item )
    Node *head = hashTable[hval]
    return search ( head, item )
```


Separate Chaining

- You can use any method to maintain set of items that collide
  - Lists, trees, another hash table
  - However, sets of items should be small so it is not useful to do something complicated

Hash Table Values

- n = number of elements stored in the hash table
- m = size of the hash table
- α = load factor
  - Average number of items stored in a list or chain (n/m)

Hash Table Run Times

- Given a good hash function
  - Insert - O(1)
  - Search average case run times
    - Unsuccessful
      - Compute HF and traverse α links
    - Successful
      - Compute HF and traverse $\alpha/2$ links

Hash Table Run Times

- Given a good hash function
  - Search -
    - Best case - O(1)
    - Worst case - O(n)
    - Average case - O(α)
- Search run times depend on hash function
- Do not use hash tables for their worst-case performance

Hash Function Analysis

- Give *hashFunction(item) = item%size*
  - Must be aware of the type of items being stored
  - Hash table of size 10 with all items ending in a 0 would be very poor

Hash Table Size Analysis

- Bad table sizes
  - Multiple of 10
  - Powers of 2
- Good table sizes
  - Prime numbers not close to a power of 2

Hashing Methods for Strings

- Use ASCII values of characters
- Method 1
  - Hash on ASCII value of the first character
    - Clumping on particular characters and some may be empty
- Method 2
  - Hash on ASCII values of characters added up
    - If size is large (ie 10,007), then does not take advantage of all slots - 127*10 = 1270

Hashing Methods for Strings

- Method 3
  - Look at the first 3 characters
  - str[0] + str[1]*X + str[2]*X*X
  - Good for random characters and large table sizes
- Method 4
  - Same as method 3 but with all characters
  - Takes too long to calculate

Hashing Methods for Strings

- Method 5
  - Variation of methods 4 using a power of 2
  - Can be done with shifts instead of multiplications
  - Still may not use all table locations

*S[0] +32*S[1] + 32*32*S[2] = S[0] + 32(S[1] + 32(S[2]))*

```
hash = 0
while (*k != '\0')
        hash += (hash << 5) + *k
        k++
hval = hash % size
```

Open Address Hashing

- Doesn't use chaining
- If a collision occurs, try another cell until an empty cell is found
- All items are stored in the hash table itself and not chained
  - Need larger table size than with chaining
- α is always <= 1

Open Address Collision Resolution Strategies

- hval(item) = (HF(item) + F(i)) % size with F(0) = 0
  - F is the collision resolution strategy
- 3 collision resolution strategies
  - Linear probing
  - Quadratic probing
  - Double hashing

Linear Probing

- F(i) = i
- **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"

Linear Probing

hval(item) = (HF(item) + F(i)) % size
HF(item) = item%size F(i) = i

13, 15, 24, 6, 20

0	6
1	15
2	20
3	24
4	
5	
6	13

((13%7) + 0)%7 = 6
((15%7) + 0)%7 = 1
((24%7) + 0)%7 = 3
((6%7) + 0)%7 = 6 - COLLISION
((6%7) + 1)%7 = 0 - PROBE
((20%7) + 0)%7 = 6 - COLLISION
((20%7) + 1)%7 = 0 - PROBE
((20%7) + 2)%7 = 1 - PROBE
((20%7) + 3)%7 = 2 - PROBE

Linear Probing

- As long as the table is large enough, a free cell will always be found
- Even if the table is relatively empty, it may take many probes
- Primary clustering
  - Results from clumping due to linear probing

Primary Clustering

- In class exercise - hash the following values using linear probing into a hash table of size 25
  - 0, 25, 50, 75, 100

Primary Clustering

Search with Linear Probing

- *Consider a hash table A that uses linear probing*
- *find(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*

Quadratic Probing

- $F(i) = i^2$
- Eliminates primary clustering

Quadratic Probing

0	0
1	25
4	50
9	75
16	100
24	

$((0\%25) + 0^2)\%25 = 0$
$((25\%25) + 0^2)\%25 = 0$ - COLLISION
$((25\%25) + 1^2)\%25 = 1$ - PROBE
$((50\%25) + 0^2)\%25 = 0$ - COLLISION
$((50\%25) + 1^2)\%25 = 1$ - PROBE
$((50\%25) + 2^2)\%25 = 4$ - PROBE
$((75\%25) + 0^2)\%25 = 0$ - COLLISION
$((75\%25) + 1^2)\%25 = 1$ - PROBE
$((75\%25) + 2^2)\%25 = 4$ - PROBE
$((75\%25) + 3^2)\%25 = 9$ - PROBE
$((100\%25) + 0^2)\%25 = 0$ - COLLISION
$((100\%25) + 1^2)\%25 = 1$ - PROBE
$((100\%25) + 2^2)\%25 = 4$ - PROBE
$((100\%25) + 3^2)\%25 = 9$ - PROBE
$((100\%25) + 4^2)\%25 = 16$ - PROBE

Quadratic Probing

- In class exercise - hash the following values in the order shown into a table of size 7 using quadratic probing
  - 0, 6, 7, 3, 14, 13

Quadratic Probing

Quadratic Probing

- Cannot take advantage of the entire table size.
  - May not find an empty slot
- Must carefully choose table size
- Quadratic probing causes secondary clustering
  - Same initial probe gives same probe sequence

Double Hashing

- F(i) = i * HF2(item)
  - H(item) = (HF(item) + i*HF2(item))%size

Rehashing

- If the table becomes too full, insertions can begin to take a long time or be denied
- Given a threshold, make a new hash table of twice the size and rehash the values to the new table size
  - Threshold ≈ 70% full
- Rehashing takes O(n)