

L16 - Hashing

7/18/24

Hashing demo

- [hash.py](#)

Hashing applications

- When I download a file, how can I verify that the download was successful (no errors or tampering)?
 - Using a common hash function (e.g., SHA256), the website can post the hash of the file. I can then hash my file and compare
- Websites need to verify my password when I login, but they can't store it in plaintext (insecure)
 - They store the hash of my plaintext password
- Cryptocurrency/blockchain (simplified example)
 - The blockchain can be thought of as a linked list of blocks where the links are hashes
 - The system generates a “target” hash value (say, a hash where the first 30 bits are all 0) for the current block, and you need to find a value that, after being hashed, is less than that value
 - If so, you get a Bitcoin (and create a new block)
- For us,
 - Hashing is $O(1)$, allowing us to implement some operations such as insert, delete, find in $O(1)$
 - But ordering operations (findMin, traversal) cannot be done
 - Basic idea, we can index into an array (integer indices) in $O(1)$. What if the key is not an integer? Convert it to an integer by hashing, then use the hash to locate data in an array-like structure

Hash terms

- Hashing is the basic concept of computing an integer (the “hash” or “hash value”) from some data value (the “key”)
- We intend to use that hash integer as an index into an array or table of associated data (keys and associated values)
- Map is an ADT similar to Python’s dict
 - void put(k, v)
 - V get(k)
 - void remove(k)
 - boolean contains(k)
- HashMap is an implementation of Map using a hash function
- Hash table is the array where data is stored

Hash function

- The computation that generates a hash value from a key
 - `hash(key) -> int`
- Used to implement Map via hash table
 - `get(key)` generally becomes `table[hash(key)]` in the implementation

Hash table

- Hash table is an array of key/value pairs
- put("jones", 4834173)
 - Suppose hash("jones") is 5
 - So we put this K, V pair into array slot 5
- get("jones")
 - hash("jones") is again 5
 - We look at array slot 5 and retrieve the associated value 4824173
- What if hash is bigger than table size?
 - Use modulus, i.e., $\text{index} = \text{hash}(\text{"jones"}) \% \text{size}$
 - May omit the % later, but you should assume it's there

0	
1	
2	
3	
4	
5	jones, 4824173
6	
7	
8	
9	
10	

Time complexity

- Since we assume the hash function is $O(1)$ to compute, put, get, and find are $O(1)$
 - Compute hash
 - Look in array slot
- Find is $O(n)$ for simple array, average-case $O(\log n)$ for BST

Collision

- For simplicity, can show hash tables with just the key, but remember for a Map, there can be associated data
- In this table, suppose $\text{hash}(\text{"sam"})$ is 7
- Suppose $\text{hash}(\text{"lara"})$ is also 7
- Slot already taken
- Collision!

0	
1	
2	bob
3	jane
4	bill
5	
6	
7	sam lara
8	
9	amy
10	

Pigeonhole principle

- Are collisions possible to avoid entirely?
- Would be possible if two distinct keys always get two different hash values
- But we allow our keys to be anything, whereas the hash output is fixed-size
- Pigeonhole principle: if there are 8 chicken boxes and 9 chickens, there must be 2 chickens in some box
- Best we can do is design the hash function so that it distributes keys evenly over the available array subscripts



Hash functions and collisions in practice

- A good hash function makes collisions very rare
- E.g., SHA256 has a 256 bit output, thus 2^{256} possible outputs
- How large is 2^{256} ?
 - 10^{78} to 12^{82} atoms in the universe
 - To date, no one has found a collision for SHA256 (doing so would break it for cryptographic purposes)
- But for our purposes, we can't always spare 256 bits of output per key (SHA256 is more for cryptography)
- We have to deal with collisions

Good hash functions

- Hash function must be fast to compute
 - $O(1)$
 - Really, something like $O(K)$ where K is the key size, but we'll assume keys are fixed-size (e.g., strings have a max length) so that this becomes $O(1)$
- Hash function must distribute keys evenly over the available range of values
 - For us, the available range is $\{0, 1, \dots, \text{array size}-1\}$
- Incorporates all data of the key
- Decorrelates keys such that if two keys are similar, they should not get similar hash values
- Ideally, two distinct keys should get two different hash values

Probability of collisions

- Probability of collision affected by
 - Quality of hash function
 - How well it evenly distributes keys over the index range
 - Table structure
 - Number of array slots
 - Mathematical properties of the maximum index
 - E.g., if size is prime or not
 - Will show example soon
- For the rest of lecture, assume keys are lowercase Strings with some reasonable maximum upper bound

Example bad hash function

- Suppose our hash function for String is the first letter of the key as its position in the alphabet

```
static int badHash(String key) {  
    return ((int) key.charAt(0)) - ((int)  
'a'));  
}
```

Why is bad hash function (first letter's position in alphabet) bad?

Nobody has responded yet.

Hang tight! Responses are coming in.

Why is it bad?

- Only 26 different range elements
 - Can only store 26 keys before guaranteed collisions
- First character is not evenly distributed over alphabet
 - Lots of “s”, “m”, “t” words
 - Not many “x”, “z”, “q” words

Better hash function

- Sum all chars, mod by table size

```
static int betterHash(String key, int tabSize) {  
    int hval = 0;  
    for (int i=0; i<key.length(); i++) {  
        hval += key.charAt(i);  
    }  
    return hval % tabSize;  
}
```

If tabSize is large (say 10007)
And keys short (say 8-15 chars)

Then the sum of the chars is small
and will cluster at low end of table

Come up with a collision for betterHash?

“cat”, “act” (doesn’t matter what table size is)

Nobody has responded yet.

Hang tight! Responses are coming in.

Pretty good hash

- Use multiplication for bigger numbers to avoid clustering
- Use prime multiplications to avoid small cycles
 - Example cycle: $[0\dots 9] \% 8 = [\underline{0}, 1, 2, 3, 4, 5, 6, 7, \underline{0}, 1\dots]$
- Multiplier and tabSize should be coprime
 - Consider multiplier 2 and tabSize 8, what goes wrong?

```
static int prettyGoodHash(String key, int
tabSize) {
    int hval = 7;
    for (int i = 0; i < key.length(); i++)
    {
        hval = 31 * hval + key.charAt(i);
    }
    hval = hval % tabSize;
    if (hval < 0) {
        hval += tabSize;
    }
    return hval;
}
```

Another idea: multiply by multiplierⁱ, e.g. $31 * \text{charAt}(0)$, $31^2 * \text{charAt}(1)$, etc.

Table size

- Best to use a prime table size
- Or, for convenience (i.e., don't have to choose prime number), power of 2 as the table size, but do not use even multipliers for the multiplication
- Load factor
 - # elements in table / table size
 - 500 elements, size 997 table => load factor $500/997$
 - Table half-full?
 - Depends on how collisions are handled

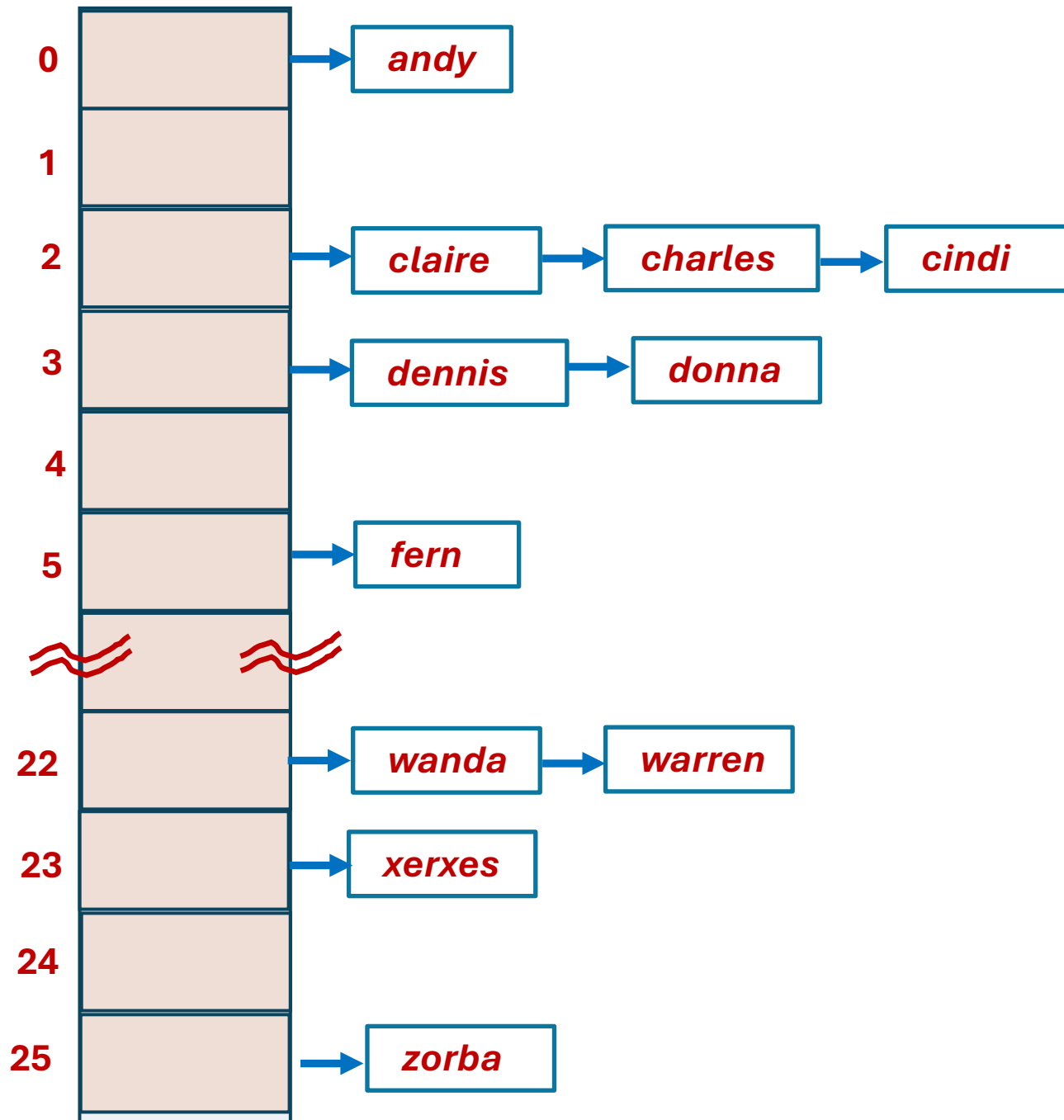
Collision resolution

- Two main forms
- Chaining
 - Each array slot contains not a single element but a list
- Linear probing
 - Each array slot contains one element
 - If we hash to full slot, we have a plan for going to a next slot to try
- How does this affect $O(1)$ of insert and find?

Chaining

- Each entry is null or a list of cells
- If a new key hashes to an empty slot, start a new list with that key data
- If a new key hashes to an occupied slot, add that key data to the list

Example



Keys: *hash*

<i>andy,</i>	0
<i>dennis,</i>	3
<i>zorba,</i>	25
<i>claire,</i>	2
<i>wanda,</i>	22
<i>charles,</i>	2
<i>fern,</i>	5
<i>warren,</i>	22
<i>cindi,</i>	2
<i>xerxes,</i>	23
<i>donna</i>	3

Use bad hash function (first char) for simplicity

Chaining operations

- put(key, value)
 - hash(key) to get table index
 - Look for key in the list at that index in the table
 - If key exists in that list, replace associated value with new value
 - If key does not exist in that list, add key/value pair to that list
- find(key)
 - hash(key) to get table index
 - Look for key in the list at that index in the table
 - If key exists in that list, return associated value
 - Otherwise, does not exist (return null or throw exception, etc.)

Chaining operations

- remove(key)
 - hash(key) to get table index
 - Look for key in the list at that index in the table
 - If key exists in that list, remove key/value pair from list

Chaining operations time complexity

- Get (average and worst)
 - Calculate hash to find right list
 - $O(k) \Rightarrow O(1)$ for bounded key size
 - Traverse list looking for key
 - $O(\text{average list size})$
 - Average list size == load factor
 - If we resize table when load factor hits a constant limit, this is $O(1)$
- **$O(1)$**

Chaining operations time complexity

- put (average)
 - Calculate hash to find right list
 - $O(K) \Rightarrow O(1)$
 - Traverse list
 - $O(\text{avg. list size}) \Rightarrow O(\text{load}) \Rightarrow O(1)$
 - Insert into list if not found
 - $O(1)$, add to head of list
 - No need for tail pointer
 - **$O(1)$**
- put (worst)
 - May need to resize table if load limit exceeded to reduce average list size and spread keys out
 - Each existing K needs to be rehashed
 - **$O(n)$**
 - Best practice: resize when load exceeds **1.0**

Is BST instead of list worth it?

- In the case of a collision where we have to traverse a list to find some element, why don't we store a BST to make traversal faster? That is, $\log(\text{list length})$ instead of $\frac{1}{2}$ list length
- BST is over-complicated for little gain, if any
- Prefer to focus on keeping lists short so that we can consider $O(\text{list length})$ to be **$O(1)$**
- Make table size bigger, make hash function distribute over more slots

Probing

- Table entry is null or a single cell
- If a new key hashes to an empty slot, then store a cell there with that key data
- If a new key hashes to an occupied slot, compute a next slot to try (repeat as needed)

Linear probing

- Linear probing says to try other “nearby” slots
- If $\text{table}[\text{hash}(\text{key})] \% \text{size}$ is full
- Try $\text{table}[\text{hash}(\text{key})+1] \% \text{size}$, and if that's full
- Try $\text{table}[\text{hash}(\text{key})+2] \% \text{size}$
- ...
- Until a slot is open

Example

Keys: **hash**

andy, 0
dennis, 3
zorba, 25
claire, 2
wanda, 22
charles, 2 → 3 → 4
fern, 5
warren, 22 → 23
cindi, 2 → 3 → 4 → 5 → 6
xerxes, 23 → 24
donna, 3 → 4 → 5 → 6 → 7

*Shows “clustering”
or “clumping”
where you get
heavily used
crowded parts,
empty parts...*

0	andy
1	
2	claire
3	dennis
4	charles
5	fern
6	cindi
7	donna
8	
22	wanda
23	warren
24	xerxes
25	zorba

Probing operations

- put(key, value)
 - Hash(key) to get table index
 - If table entry contains correct key or is empty, then replace value or store key/value pair at that spot as appropriate
 - Else, try the next table entry
 - Continue until success
- find(key)
 - Hash(key) to get table index
 - If table entry contains matching key, then return associated value
 - Else, try the next table entry
 - Continue until key is found or empty spot is encountered
 - If empty spot is encountered, then key is not in map

Probing operations - remove

- remove(key)
 - **Not so simple**, can't just empty the table cell which might create a gap in some probe chain
 - E.g., on slide before previous one, suppose we actually delete "dennis"
 - Then when attempting to find "charles", we go 2 -> 3, but 3 is now empty, so we assume "charles" is not in the table and stop finding
 - **Lazy deletion** required
 - Replace removed key with some "inactive" marker
 - On find, "inactive" says "occupied, keep probing"
 - On put, "inactive" says "open, free for use"

Clustering issue

- Clustering slows access
 - It's like having to search a list in hashing to lists
- Solution: larger table size
 - Table space in probing is like the list cells in chaining
- More space means more open slots for initial hash, less hopping to probe
- Load λ should be $\frac{1}{2}$ for probing (assumes well distributed hash function)
 - Probing guaranteed to work if $\lambda < 1.0$ (i.e., not full), but 0.5 for better performance

Clustering solution?

- Need custom hash function?
- Some data may have a form that makes clusters with some hash function, not others
- Consider a hash function that uses first 3 chars
 - McDuff, MacBeth, McBride, McDaniel, MacGraw, MacDonald, MacLean, McKensie, McDermott, ...
 - These will collide
- No hash function is perfect for all data

Clustering solution: probe randomly

- We'll actually probe “more randomly”
- Probe farther away from collision site and leave some slots near the collision open for future keys
- General probing formula
 - $h_i = \text{hash}(\text{key}) + f(i)$
 - $f(0) = 0$
- Can get different probing patterns using different formulas $f(i)$

General Probing

Let's formalize this

A key defines a sequence of hash values

$$h_0, h_1, h_2, \dots, h_n, \dots$$

We try each hash val in sequence until we get an open slot

$$h_i = \text{hash}(\text{key}) + f(i)$$

$$f(0) = 0$$

this makes

$$h_0 = \text{hash}(\text{key}) \quad \text{the basic hash value}$$

Linear Probing

We get different probing patterns by defining different $f(i)$ functions

Linear Probing:

$$h_i = \text{hash}(\text{key}) + f(i)$$

$$f(0) = 0, \quad f(i) = i \quad \text{for } i > 0$$

Sequence: *hash(key)+0*

hash(key)+1

hash(key)+2

hash(key)+3 ... % table length

Quadratic Probing

Probe via skipping by squares

$$h_i = \text{hash}(\text{key}) + f(i)$$

$$f(0) = 0, \quad f(i) = i^2 \quad \text{for } i > 0$$

Sequence: 0: $\text{hash}(\text{key})+0$

1: $\text{hash}(\text{key})+1^2$ is $\text{hash}(\text{key})+1$

2: $\text{hash}(\text{key})+2^2$ is $\text{hash}(\text{key})+4$

3: $\text{hash}(\text{key})+3^2$ is $\text{hash}(\text{key})+9$

...

8: $\text{hash}(\text{key})+8^2$ is $\text{hash}(\text{key})+64$

% table length

Exponential Probing

Probe via skipping by powers of 2

$$h_i = \text{hash}(\text{key}) + f(i)$$

$$f(0) = 0, \quad f(i) = 2^i \quad \text{for } i > 0$$

Sequence: 0: $\text{hash}(\text{key})+0$

1: $\text{hash}(\text{key})+2^1$ is $\text{hash}(\text{key})+2$

2: $\text{hash}(\text{key})+2^2$ is $\text{hash}(\text{key})+4$

3: $\text{hash}(\text{key})+2^3$ is $\text{hash}(\text{key})+8$

...

8: $\text{hash}(\text{key})+2^8$ is $\text{hash}(\text{key})+256$

% table length

Probing performance

- get (average and worst)
 - Hash(key) to find initial slot
 - $O(1)$
 - Traverse probing sequence looking for key
 - $O(\text{avg. cluster size})$
 - Avg. cluster size based on load factor
 - If we resize table when load factor hits constant limit, then $O(1)$
- **$O(1)$**

Probing performance

- put (average)
 - Hash(key) to find initial slot
 - $O(1)$
 - Traverse probing sequence looking for key
 - $O(\text{avg. cluster size}) = O(\text{load}) = O(1)$
 - Insert into empty slot $O(1)$
 - **$O(1)$**
- put (worst)
 - May need to resize table if load limit exceeded
 - Each existing K rehashed
 - **$O(n)$**

Practice problem 1

- Using an initially empty HashTable of size 11 and the Hash Function $H(k) = k \% 11$, insert the following keys, in the given order, using the linear probing method: 0, 1, 8, 9, 41, 33, 45, 42, 61, 53

Practice problem 1 solution

- To save work, first apply the hash function to all values (though not what would happen in reality)
- $[0, 1, 8, 9, 41, 33, 45, 42, 61, 53] \pmod{11} =$
- $[0, 1, 8, 9, 8, 0, 1, 9, 6, 9]$

Index	Key
0	0
1	1
2	33
3	45
4	42
5	53
6	61
7	
8	8
9	9
10	41

Practice problem 2

- Using an initially empty HashTable of size 11 and the Hash Function $H(k) = k \% 11$, insert the following keys, in the given order, using the quadratic probing method: 0, 1, 8, 9, 41, 33, 45, 42, 61, 53

Practice problem 2 solution

- To save work, first apply the hash function to all values (though not what would happen in reality)
- $[0, 1, 8, 9, 41, 33, 45, 42, 61, 53]$
 $\% 11 =$
- $[0, 1, 8, 9, 8, 0, 1, 9, 6, 9]$

Index	Key
0	0
1	1
2	45
3	53
4	33
5	
6	41
7	61
8	8
9	9
10	42