time complexity of sequential search has been covered in the previous notebook

Searching

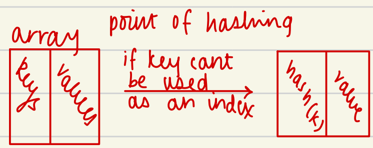
- sequential : worst case : $\Theta(n)$
- binary : worst case : $\Theta(\log_2 n)$

to improve to $\Theta(1)$, we sacrifice space

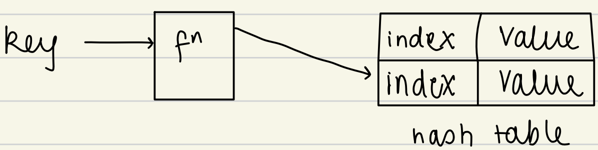
Hash Tables : efficient lookup, insertion and deletion of key-value pairs

- hashing : process of using a hash f^n to map data of arbitrary size to fixed-size values or keys

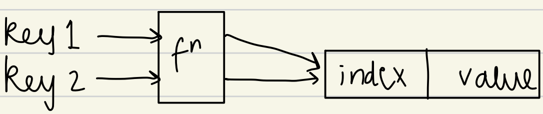
· hash function
· collision & its resolution
· deleting a key
· resizing hash table



- hash fn : {all possible keys} → {0, 1, 2...., h-1}

- 

← hash slot

- 

← collision

- if n records are stored in a table with h slots load factor is $\alpha = \frac{n}{h}$

range from $0.6 \to 0.75$

$\alpha \uparrow$, time complexity $\uparrow$

resize as
$\alpha \to 1$ (up)
$\alpha \to \alpha_{max}/4$ (down)

f : key space $\longrightarrow$ hash code

hash functions

- must map all possible values within the range of the hash table uniquely
- mapping should achieve an even distribution of the keys
- easy & fast, minimise collision

1. modulo

$H(R) = R \bmod h$ ← table size

if key is :
decimal → $h \neq$ power of 10
binary pattern → $h \neq$ power of 2
"real" data → h = prime number, not close to a power of 2

2. folding (all bits contribute to the result)

- partition the key into several parts and combine the parts in a convenient way
- shift folding : divide the key into a few parts and add them up.

"abcdef" ⟶ 599 (add each ascii code)

3. Mid-square (example of folding)
Key is squared and the middle part is used.

4. multiplicative congruential method
· pseudo-random number generator

$$a = 8\left\lfloor \frac{h}{23} \right\rfloor + 5$$

$$H(k) = (a \times k) \bmod h$$

redone next page ⟶ ways of hashing

· direct address table

· closed address hashing — linked list $\square \to \square \to \square$. entries at index

· open address hashing
 - linear probing
 - quad. probing
 - double hashing

for multiple collision soln

· closed worst case : all in same list : $\Theta(n)$

avg case $\frac{n}{h} \rightsquigarrow \Theta(\alpha)$

Direct Address Table

- every key in the universe is mapped to a finite number of hash slots

key domain

clearly, collisions will occur

soln

closed / open address hashing

Closed Address Hashing or Chained Hashing

- maintains original hashed address
- records hashed to the same slot are linked into a list
- address is closed (fixed). each key has a corresponding fixed address

- n records : $\alpha = \frac{n}{h}$ load factor

no of slots

number of elements per list on an avg

analysis

- worst case behavior : all elements in 1 slot

unsuccessful : n searches

successful : $\frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2}$

sequential search

prob of finding 1 item

sum of time taken to find each item

- average case : each item has equal prob of being hashed into any of the h slots.

unsuccessful : $\frac{n}{h}$ key comparisons per list

$\therefore$ α no of comparisons on an avg

successful : because the key is inserted at the end, we have to imagine its not there yet.

(i^{th} being inserted): avg length of all lists: $\frac{(i-1)}{h}$

to find newest one : $1 + \left(\frac{i-1}{h}\right)$ comparisons

avg over n items

$$\frac{1}{n}\sum_{i=1}^{n} 1 + \frac{(i-1)}{h} = \frac{1}{n}\sum_{i=1}^{n}(1) + \frac{1}{nh}\sum_{i=1}^{n}(i-1)$$

$$= 1 + \frac{1}{nh}\sum_{i=0}^{n-1} i$$

$$= 1 + \frac{n-1}{2h} = \frac{n}{2h} + 1 - \frac{1}{2h}$$

h is constant

$$= \frac{\alpha}{2} + 1 - \frac{1}{2h}$$

- $\Theta(1+\alpha)$ comparisons

??? - if $n \propto h \rightarrow n = O(h)$, $\frac{n}{h} = \frac{O(h)}{h} = O(1)$

proportional

$$\lim_{\alpha \to \infty} \frac{\alpha/2 + 1 - 1/2h}{1+\alpha} = \frac{\alpha/2 + 1 - c}{1+\alpha} = \frac{1/2}{1} = \frac{1}{2} \leftarrow \text{constant}$$

$\therefore$ correct.

deleting
- insert in first deleted slot always
- check if key occurs after also
- if not then go back and add it there

Open Address Hashing

- all elements in the hash table
- $\alpha = n/h \leq 1 \quad \therefore$ ↑
- when collision : probe

1. linear probing
2. Quadratic probing
3. Double Hashing

1. Linear Probing

- if filled, go to next

$H(k, i) = (k+i) \bmod h$ (if hashing f^n is $k \bmod h$)
for $i = 1, 2 \ldots\ldots h-1$ until an empty slot is found.

- problem : primary clustering

Algorithm 3 Searching a Key

```
function SEARCH(k)
    code ← hash(k)
    loc ← code
    ans ← ∅
    while H[loc] ≠ ∅ do
        if H[loc].key == k then
            ans ← H[loc]
            break
        else
            loc ← probe(loc)
            if loc == code then
                break
```

time complexity

success : $\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$

- unsucessful : $\frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right)$

$\underbrace{\hspace{6em}}_{\text{don't memorise}}$

2. Quadratic Probing

- $H(k, i) = (k + c_1 i + c_2 i^2) \bmod h$

- c_1, c_2, h must be selected carefully to ensure all slots are probed

- $h = 2^n \longrightarrow c_1 = c_2 = 1/2$

∴ avoids primary clustering, results in secondary clustering

(watch animation in notes)

if 2 keys have the same initial probe position, their probe sequences will be the same.
$\Rightarrow$ later keys will take a longer time

- time complexity

dont memorise

- success : $1 - \ln(1-\alpha) - \frac{\alpha}{2}$

- unsuccess : $\frac{1}{1-\alpha} - \alpha - \ln(1-\alpha)$

3. Double Hashing

- more random method
- $H(k, i) = (k + iD(k)) \bmod n$
  ↳ $D(k)$ is another hash f^n and $i \in [0, n-1]$

- n should be prime · impossible for any number to divide it evenly ∴ probe sequence will check out every slot.

- Time complexity

dont memorise

success : $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

unsuccess : $\frac{1}{1-\alpha}$

note : all complexities are f^ns of $\frac{1}{1-\alpha}$

TIME COMPLEXITY OF BINARY SEARCH

- $\Theta(\log_2(n))$

- in a binary tree w/ height h and size n

  no of edges

$$2^h - 1 < n \leq 2^{h+1} - 1$$
$$2^h \leq n < 2^{h+1}$$
$$h \leq \log_2 n < h+1$$

$\therefore$ minimal height $= \lfloor \log_2 n \rfloor$

- worst case: $\leftarrow$ search till end and still not find.

Binary Search

```
BTNode* findBSTNode(BTNode *cur, char c){        --> T(n)
{
    if (cur == NULL) {
        printf("Not Found\n");
        return cur;
    }

    if(c==cur->item){
        printf("Found\n");
        return cur;
    }                                             --> c

    if(c<cur->item)
        return findBSTNode(cur->left,c);          --> T((n-1)/2)
    else
        return findBSTNode(cur->right,c);         --> T((n-1)/2)
}
```

Listing 1: Recursive Version

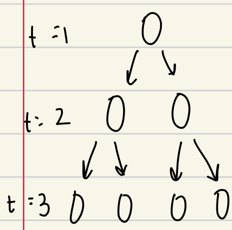
$$T(n) = T\left(\frac{n-1}{2}\right) + c$$
$$\vdots$$
$$= T\left(\frac{n-2^k+1}{2^k}\right) + kc$$
$$= c + \lceil \log_2 n \rceil c$$
$$= c(\lceil \log_2 n \rceil + 1)$$

$$0 < \frac{n-2^k+1}{2^k} \leq 1$$
$$0 < \frac{n+1}{2^k} - 1 \leq 1$$
$$1 < \frac{n+1}{2^k} \leq 2$$
$$2^k < n+1 \leq 2^{k+1}$$
$$k < \log_2(n+1) \leq k+1$$

$$\lceil \log_2(n+1) \rceil = k+1$$
$$\vdots$$
$$\lceil \log_2 n \rceil = k$$

average time for successful search

why??



assume : $n = 2^k - 1$ (complete tree)

2^{t-1} position require t comparisons

$$A = \frac{1}{n} \sum_{t=1}^{k} t\, 2^{t-1}$$

$$= \frac{(k-1)2^k + 1}{n} \; ..... = \log_2 (n+1) - 1 + \frac{\log_2 (n+1)}{n}$$

$n^2 + \frac{n^2}{4} + \frac{n^4}{16} + \ldots$

$n^2 + \left(\frac{n^2}{4}\right)^1 + \left(\frac{n^2}{4^2}\right)^2 + \ldots$

$\frac{n(n+1)}{2}$

tower of hanoi : $T(n) = 2T(n-1) + 1$