Dictionaries

→ dictionary ADT
→ binary search
→ Hashing.

Dictionaries store elements so that they can be located quickly using keys.

→ A dictionary may hold bank accounts
  → each account is an object that is identified by an account number.
  → each account stores a wealth of information
    → name, current balance, transactions etc.

→ an application wishing to operate on an account would have to provide the account number as the search key.

The dictionary ADT

Dictionary is an abstract model of a database.
  → A dictionary stores key - element pairs
  → The main operation supported by a dictionary is searching by key.

→ simple container methods:
  (i) size(), isEmpty(), elements()

returns all elements

→ Query methods : find Elem (k) , find All Elem (k)

→ update method : insert Item (k, e) , remove Elem (k), remove All Elem (k)

→ special element : NIL : returned by an unsuccessful search.

$\underbrace{\hspace{6em}}$

No key or no element on that key.

⇒ keys are only comparable for equality. There is no larger or smaller key.

→ Different data structures to realize dictionaries :-

(i) Arrays, linked lists (inefficient)

(ii) Hash table

(iii) Binary trees

(iv) Red / Black trees

(v) AVL trees

(vi) B- trees

Searching

INPUT

- seq of numbers (database)
- a single number (query)

OUTPUT

- Index of the found number or NIL

$a_1\ a_2\ a_3 \cdots a_n : q$

2 5 4 10 7 : 5 — 2

2 5 4 10 7 : 9 — NIL

⇒ Binary search

(i) Divide and Conquer: a key design technique.

→ just compare with middle element in a sorted array.

Recursive approach

Algorithm Binary_search (A, k, low, high):

if low > high then return NIL

else:

mid = (low + high) / 2

if k = A[mid], then return mid

elseif k < A[mid], then

return B.S (A, k, low, mid-1)

else

return B.S (A, k, mid+1, high)

→ iterative procedure

```
low ← 1
high ← n
while (low ≤ high)
        mid = (low + high) / 2
        if A[mid] = k then return mid
        else if A[mid] > k then high ← mid - 1
                           else low ← mid + 1

return NIL
```

→ Running time of binary search

$O(\log_2 n)$.

⇒ **Problem**: T & T is a large phone company, and they provide caller ID capability:

→ given a phone number, return the caller's name

→ phone numbers range from 0 to $r = 10^8 - 1$

→ There are n phone numbers, $n \ll r$.

→ want to do as efficiently as possible.

(i) using unordered sequence

34 — 14 — 12 — 22 — 18

→ searching and removing takes $O(n)$ time.

→ inserting takes $O(1)$ time.

applications : log files → very frequent insertions,
rarely searches & removals

unordered sequence is used here.

→ using an ordered sequence

12 — 14 — 18 — 22 — 34

→ searching takes $O(\log n)$ time
→ insertion takes $O(n)$ time.

applications: lookup tables → frequent searches
rare insertion or deletions

→ Other suboptimal ways

direct addressing : an array indexed by key:

→ every operation takes $O(1)$ time
→ $O(r)$ space where r is the range of numbers (10^8)
→ huge amount of space wasted.

null	null	Ankur	null	Riya
xxxx_xxx	xxx	984356	xxxx	9413358

Phone number is the index.

Hash table → $O(1)$ expected time;

$O(n+m)$ space, where m is table size

→ Like an array, but come up with a function to map the large range into one which we can manage.

eg. take the original key, modulo the (relatively small) size of the array, and use that as an index.

→ Insert (9635-8904, Ankur) into a hashed array with, say, five slots. 96358904 mod 5 = 4.

An example

→ Let keys be entry no's of students in CSL201.

eg. 2004CS10110.

→ There are 100 students. We create a hash table of size, say 100.

→ Hash function is say, last two digits.

→ Then 2004CS10110 goes to location 10.

→ where does 2004CS50310 goes?

- We are really good if no two elements clash.

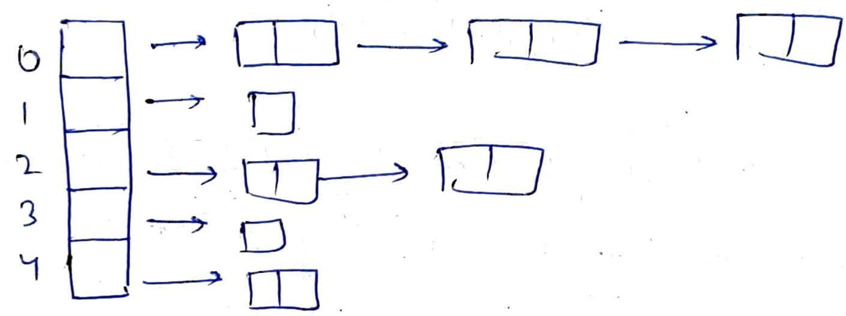
How do we deal with collision ?

collision resolution

→ Two keys which hash to the same spot in the array.

→ Use chaining

→ set up an array of links (a table), indexed by the keys, to lists of items with the same keys



worst case O(n) (searching)

→ Most efficient (time-wise) collision resolution scheme.

→ To find / insert / delete an element

→ using hash function, look up its position in table T

→ search / insert / delete the element in the linked list of the hashed slot.

Analysis of Hashing

(i) An element with key k is stored in slot $h(k)$

(ii) The hash function maps the universe U of keys into the slots of hash table.

$T[0 \dots m-1]$

$$h : U \rightarrow \{0, 1, \dots, m-1\}.$$

(iii) Assume time to compute $h(k)$ is $O(1)$

(iv) A good hash function is one which distributes keys evenly amongst the slots.

(v) An ideal hash function would pick a slot, uniformly $\underbrace{\text{at random and hash the key to it}}_{\text{not feasible.}}$

(vi) However, this is not a hash function since we would not know which slot to look up when searching for key.

(vii) For our analysis we will use simple uniform hash function.

(viii) Given hash table T with m slots holding n elements, the load factor is defined as $\alpha = n/m$.

→ Unsuccessful search

(i) element is not in linked list

(ii) Simple uniform hashing yields an average list length $\alpha = n/m$

(iii) expected no of elements to be examined α.

(iv) search time $O(1+\alpha)$ (includes computing the hash value).

→ Successful search

(i) Assume that a new element is inserted at the end of the linked list.

(ii) upon insertion of the ith element, the expected length of the list is $(i-1)/m$.

(iii) In case of successful search, the expected number of elements examined is 1 more than the number of elements examined when the sought for element was inserted.

→ Assuming that number of hash table slots is proportional to the number of elements in the table.

(i) $n = O(m)$ or $n = k \cdot m$.

(ii) $\alpha = n/m = O(m)/m = O(1)$

(iii) searching then takes constant time on average.

(iii) insertion takes $O(1)$ worst-case time

(iv) deletion takes $O(1)$ worst-case time when the lists are doubly linked.