HASHING

The Problem

There is a large Phone Company, they want to provide caller ID capability.

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

→ phone no. range from 0 to $10^8 - 1 = r$

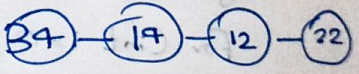
→ There are n phone numbers, $n \ll r$

→ want efficiency.

Soln 1: Unordered Sequence

→ Searching & removing : O(n)

→ inserting : O(1)



→ Used when:

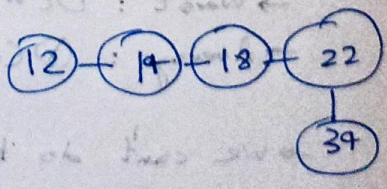
→ Frequent insertion

→ Rare searches & removal

Soln 2: Ordered Sequence

→ Searching : O(log n) [Binary]

→ Insertion & Removing : O(n)



Application

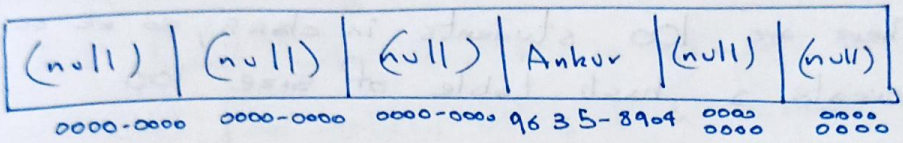
→ Frequent Searches

→ Rare insertion & removal

Soln 3 : Direct Addressing

→ An array indexed by key

→ addressing take O(1) time., but huge amount of space required. [$O(v)$, $v = 10^8$]



Soln 4 : Hash Table

→ Addressing Time : O(1)

→ Size Requirement : $O(n+m)$

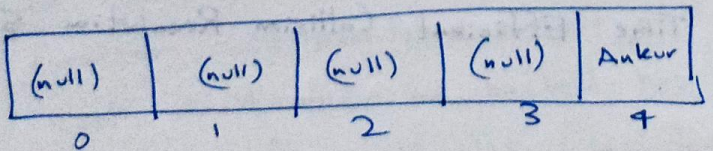
- n : no. of phone number
- m : table size.

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

Eg Function : Take original key, modulo the (relatively small) size of array, and use that as index

Eg : For Ankur, hashed into 5 slots

index = 96358904 % 5 = 4



Example

→ Let keys be entry no. of student in Batch CS 302

Eg: 2018 CS 10 110

→ There are 100 students in class, so we could create a hash table of size 100

→ Let Hash Function output last ~~to~~ 2^2 digit

Eg: 2018 CS 10110 ⟶ 10

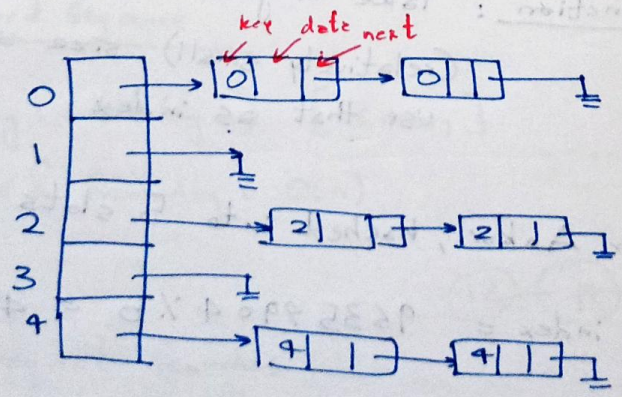
but also 2018 CS 50310 ⟶ 10 } collision.

Collision Resolution

→ To Solve Collision, we can use chaining

→ Set up an array of links (a table), indexed by keys, to list items with same key

Eg:



→ Most Time Efficient Collision Resolution Scheme.

→ To find / insert / delete an element

→ Using Hashing Function, Find at ~~key~~ hash key → h(·)

→ Using the key, you have a linked list, do Linked List Find / Insert / Delete.

Analysis of Hashing

→ An element with key k is stored in slot h(k) [~~is~~ instead of slot k without hashing]

→ The hash function h maps the universe U of keys into slots of hash table

$T[0, \ldots m-1]$

$\therefore h: U \rightarrow \{0, 1, \ldots m-1\}$

→ Let time to compute h(k) be $\Theta(1)$

→ A good hash function is the one which distributes key evenly amongst the slots

→ An ideal hash function would pick a slot, ~~unifo~~ uniformly at random and hash the key to it. But this can't be used as we cant find the slot to use while searching for that element.

Simple Uniform Hash ~~Function~~ Assumption [SUHA]

→ Hash Function is assumed SUHA when

i) function will evenly distribute ~~it~~ item to slot

ii) each ~~it~~ item to be hashed has equal ~~probabili~~ probability for each slot. [regardless of situation of table]

→ Given a hash table T with m slots holding n elements

Load Factor = $\alpha = \frac{n}{m}$

Unsuccessful search

→ Element not in linked list

→ Simple uniform hashing yeilds an average list of length $\alpha \left[\frac{n}{m}\right]$

→ Expected no. of elements to be examined = α

∴ Search time : $O(1+\alpha)$

1 → hash computing

α → searching

Successful search

→ Assume that a new element is inserted at the end of Linked List

→ upon isertion of i^{th} element, expected length of the list is $\frac{i-1}{m}$

before insertion of i^{th} node

→ in case of a successful search, the expected number of elements examined is 1 more than the number of elements examined when the sought-for element was inserted.

(previous nodes + 1 required node)

~~to find~~

→ For i^{th} element, no. of elements examine = $1 + \frac{i-1}{m}$

→ For average ~~time~~ elements for whole table we do

$$\frac{\sum_{i=1}^{n} \left(1 + \frac{i-1}{m}\right)}{n} = 1 + \frac{1}{nm} \sum_{i=1}^{n} (i-1)$$

$$= 1 + \frac{1}{nm} \times \frac{(n-1)n}{2}$$

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

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

∴ Time Complexity $= \Theta\left[1 + \frac{\alpha}{2} - \frac{1}{2m}\right]$

$= \Theta(1+\alpha)$ [generally $n \to \infty$, m = constant]

→ Assuming the number of hash table slots is proportional to the no. of elements in the table.

$n = O(m)$

$\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$

∴ → Searching takes constant time on average

→ Insertion takes $O(1)$ time

→ deletion takes $O(1)$ time, if linked list are doubly linked.

Good Hash Function

- → The function which can be computed quickly
- → It should distribute the key uniformly over the hash table
- → Very Rare
- → Eg: Birthday Paradox
  - → Even in a group of 35 or more students in the class there is a very high chance that 2 of the students would have same birthday.

How to deal with non-integer key

- → Find some way to convert key to integer

Eg: → 96 35 — 8 90 4 ⟶ 96358904

- → for a string, add up it's ~~ASCI~~ ASCII values.

Hash Functions

- → The mapping of keys to indices of a hash table is called a hash function
- → A hash function is usually the composition of two maps, a hash code map and a compression map
  - → An essential requirement of the hash function is to map equal keys to equal indices
  - → A good hash function minimises the probability of collision.

Hash code Maps

keys → integer

Compression map

integer → $[0, \ldots, m-1]$

Popular Hash Code Map

Integer Cast

→ for numeric types with 32 bits or less, we can reinterpret the bits of the number as int

Component Sum

→ For numeric types with more than 32 bits Eg long & double, we can add 32 bit component

→ Bad for strings.

Polynomial accumulation

→ For strings of a natural language, combine the character values (ASCII or Unicode) $a_0, a_1, a_2, \ldots a_{n-1}$ by viewing them as the coefficient of a polynomial

$$S = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$$

→ Horner's Rule:

$$S = a_0 + x[a_1 + x(a_2 + \ldots + x(a_{n-2} + x a_{n-1}))]$$

[can be done by recursion]

$n = 33, 37, 39, 41$ is observed with at most 6 collision on a vocabulary of 50,000 words

Compression Maps

I) $h(k) = k \bmod m = k \,\%\, m$

k → key
m → size of hash table

m → b^e form → bad.
- if m is power of 2, h(k) gives LSB of k
- all key ~~ending~~ with same ending go to same place.

m → prime → good
- ensures uniform distribution
- prime should not be too close to powers of 2

Eg: n = 2000 character string.

→ we can handle 3 node in LL

∴ we can do $m \approx \frac{2000}{3}$ but prime

= 701 (not near power of 2)

II) $h(k) = \lfloor m\ (kA \bmod 1)\ \rfloor$

$k \rightarrow$ key
$m \rightarrow$ size of table
$A \rightarrow$ constant, $A \in (0, 1)$

Steps involved:
i) map $0, 1, \ldots k_{max}$ to $0, A, \ldots A k_{max}$
ii) take it's fractional part $[\bmod 1]$
iii) map fractional part to $0, 1, \ldots m-1$

→ value of m is not critical, generally use $m = 2^p$
→ value of A depends on characteristic of data.

Fibonacci Hashing: $A = \frac{\sqrt{5}-1}{2}$ [by knuth]
↙ conjugate of golden ratio.

III) MAD [Multiply, Add, Divide]

$$h(k) = |ak + b| \bmod m$$

$k \rightarrow$ key
$m \rightarrow$ size of hash map
$a, b \rightarrow$ const.

→ eliminates pattern, given a not multiple of m
→ same formula used in linear conguential (pseudo) random number generators

Universal Hashing

→ For any choice of hash function, there exists a bad set of identifiers

→ A malicious adversary could choose keys to be hashed such that all go to same slot.

→ Average Time Retrieval becomes $\Theta(n)$

→ Solution:

→ A random hash function

→ Choose hash function independantly of keys!

→ create a set of hash function H, from which h can be randomly selected

h should be same in a runtime

→ A collection H of hash function is universal if for any randomly chosen f from H (and two key k & l),

$$\text{Probability}\left[f(k) = f(l)\right] \leq \frac{1}{m}$$

Collision Handling Techniques

→ Chaining [LL method]

→ Open Addressing
  - → Linear Probing
  - → Double Hashing

Open Addressing

→ All elements stored in hash table [can fill up]
  i.e $n \leq m$

→ Each table entry contains either an element or null

→ When searching for an element, systematically probe table slots.

→ Modify hash function to take the probe number i as the second parameter.

$$h: U \times \{0, 1, \ldots m-1\} \rightarrow \{0, 1, \ldots m-1\}$$

→ Hash function h determines the sequence of slots examined for a given key

→ Probe Sequence for a given key k given by

$$\{h(k,0), h(k,1), \ldots h(k,m-1)\}$$

⇒ a permutation of $0, 1, \ldots m-1$

Linear Probing

→ If current location is ~~full~~ used, try next location

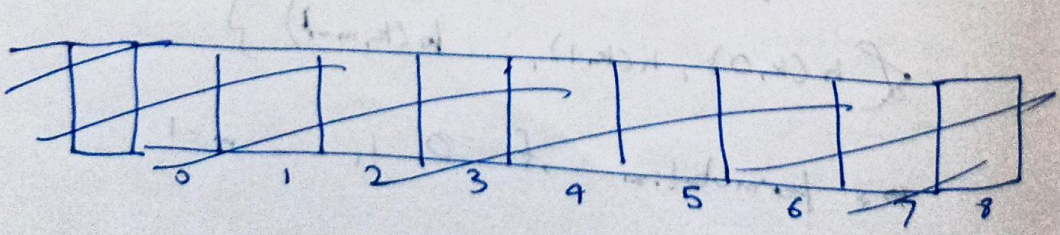
```
Algorithm  Linear Probing Insert (k)
{
    if (table is full)          → can be done O(1) by counter
        error

    probe = h(k)

    while ( table[probe] occupied)
        probe = (probe + 1) mod m
                                → circular queue method

    table[probe] = k
}
```

→ Uses less memory than chaining as ~~we~~ no links or LL

→ slower than chaining since one might have to walk along the table for a long time.

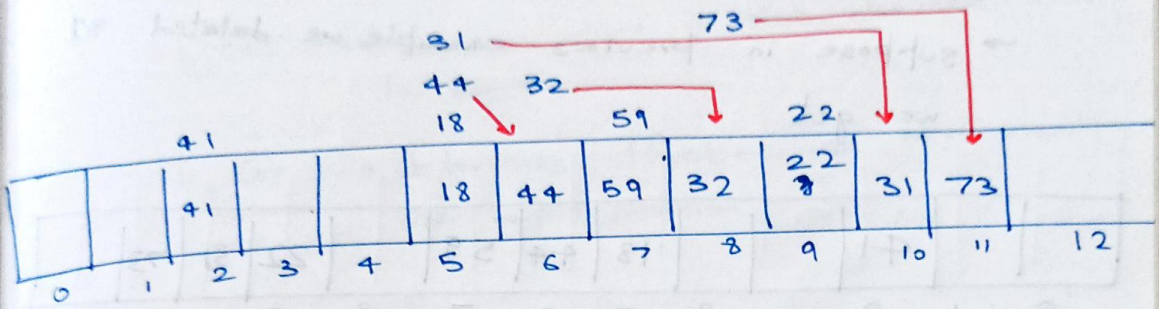
~~Eg: h(k) = k mod 13~~

~~keys → 18, 41, 22, 44, 59, 32~~



Eg: $h(k) = k \bmod 13$ [Table size = 13]

keys : 18, 41, 22, 44, 59, 32, 31, 73



Look up in Linear Probing [Searching]

→ To search key k, we go to h(k) and continue looking at successive location till we find k or encounter empty location.

→ Successful Search:

→ Eg 31

$h(k) = 31 \bmod 13 = 5$

check 5 → 18 ✗
6 → 44 ✗
7 → 59 ✗
8 → 32 ✗
9 → 22 ✗
10 → 31 ✓

∴ 31 is present at loc 10

→ Unsuccessful search.

→ Eg 33

$h(k) = 33 \bmod 13 = 7$

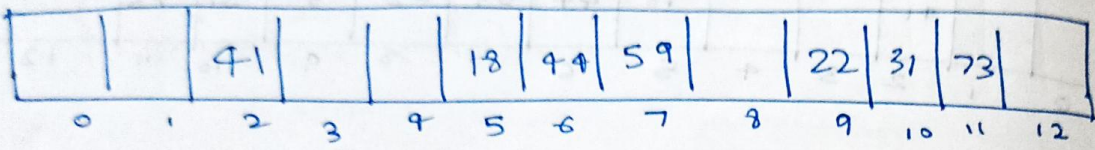
check loc 7, 8, 9, 10, 11 don't find

loc 12 → empty ∴ → 33 not present.

Deletion in Linear Probing

→ Search number and delete it

→ suppose in previous example we deleted 31 we get



→ Now if we search for 31

$h(31) = 31 \bmod 13 = 5$

check 5, 6, 7 not found
& empty → Not present (false)

→ Solution:

→ instead of placing pos & empty [null], place a tombstone [marker].

→ lookup should ignore tomb-stone and proceed normally

→ If insert comes at tombstone, remove tombstone and insert there.

→ Too many tombstones can degrade lookup performance Rehash if too many tombstones.

Double Hashing

→ Uses two hash function h_1, h_2

$h_1(k)$ → position where we first check for key k

$h_2(k)$ → determines offset we use when searching for empty position

Algorithm Double Hashing Insert (k)

```
{
    if (table is full)
        error
    probe  = h1(k)
    offset = h2(k)
    while [table[probe] occupied)
        probe = (probe + offset) mod m
    table[probe] = k
}
```

→ m should be prime for covering every position in table with any offset.

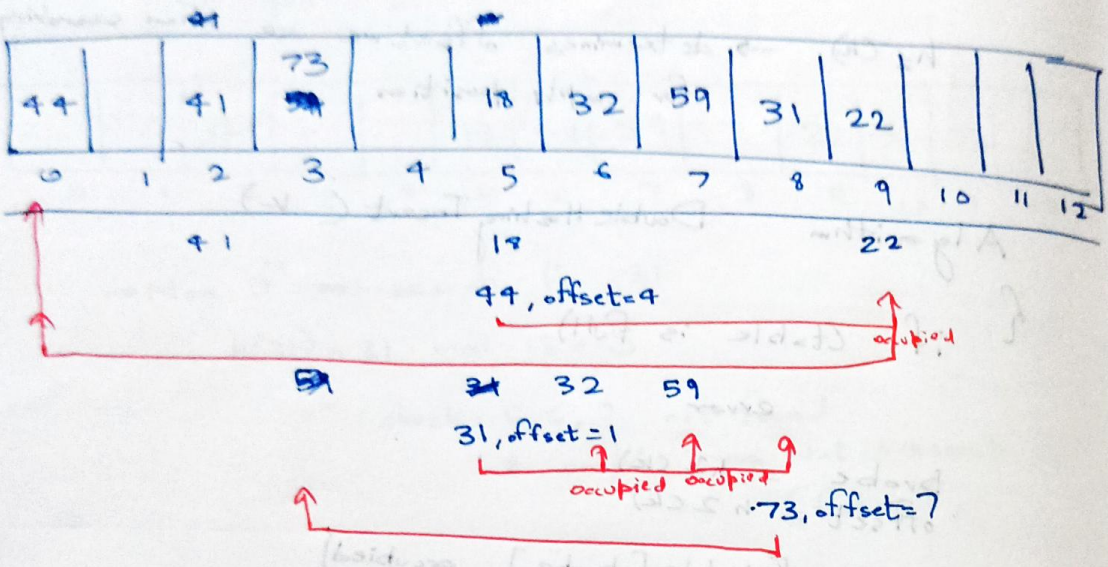
→ Same disadvantages as Linear Probing.

→ Distributes key more uniformly than Linear Probing.

Eg: $h_1(k) = k \bmod 13$ [size 13]

$h_2(k) = 8 - (k \bmod 8)$

keys: 18, 41, 22, 44, 59, 32, 31, 73



Analysis

→ we assume that every probe looks at a random location in the table.

→ $1 - \alpha$ is the fraction of empty table.

MERGE SORT

Divide and Conquer

→ Divide the problem into a no. of sub
  → Similar sub problem of smaller size.

→ Conquer the Sub Problem
  → solve the subproblem recursively
  → sub problem size small enough: solve problem in straight forward manner

→ Combine the solutions of the sub problem
  → Obtain the solution for the original problem

Merge Sort Approach

I) Divide
  → divide the n-element sequence to be sorted into two subsequence of n/2 element each

II) Conquer
  → Sort subsequence recursively using merge sort
  → when the size of subsequence is 1 there is nothing more to do

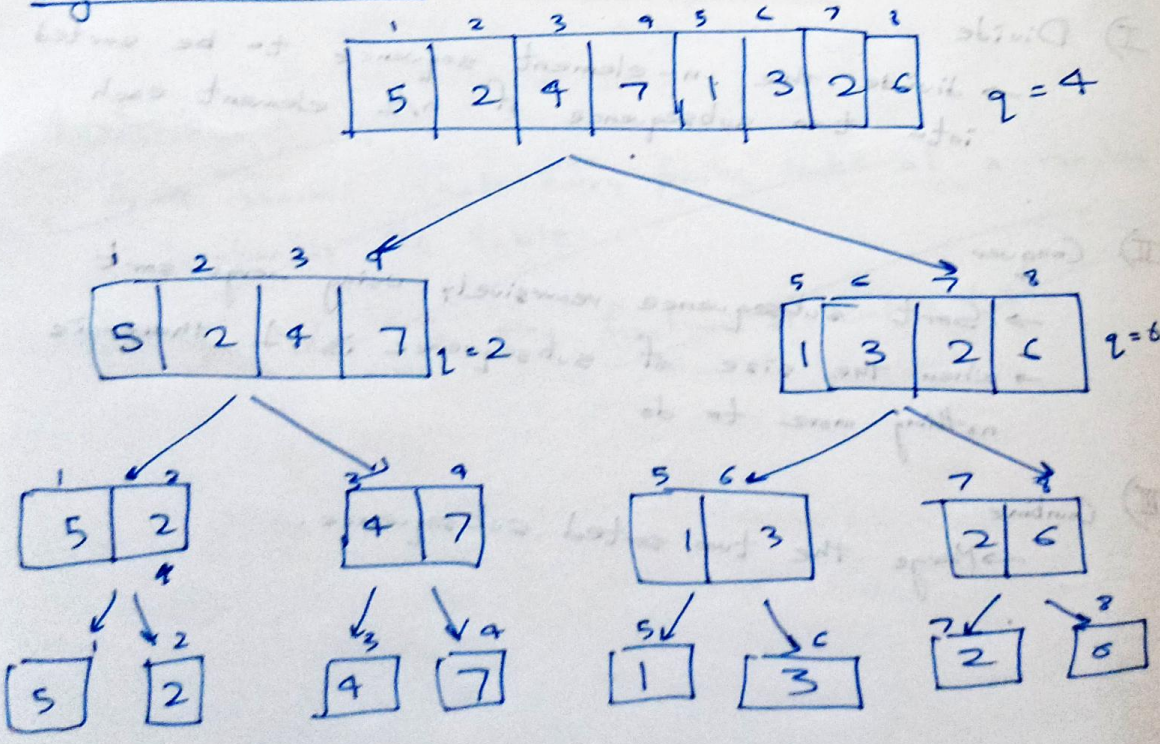
III) Combine
  → Merge the two sorted subsequence.

Algorithm MergeSort (A, p, r)

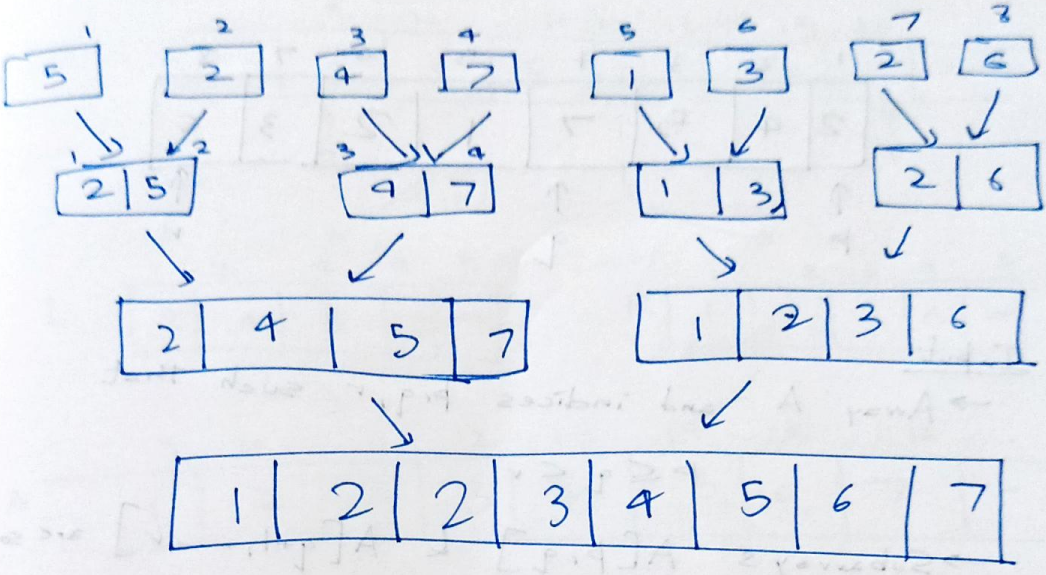
```
1  if (p < r)        // Check base case
     then  q ← ⌊(p+r)/2⌋   // Divide
           MergeSort (A, p, q)   // Conquer
           MergeSort (A, q+1, r)   // Conquer
           Merge (A, p, q, r)      // Combine.
}
```

Initial call: MergeSort (A, 1, n)

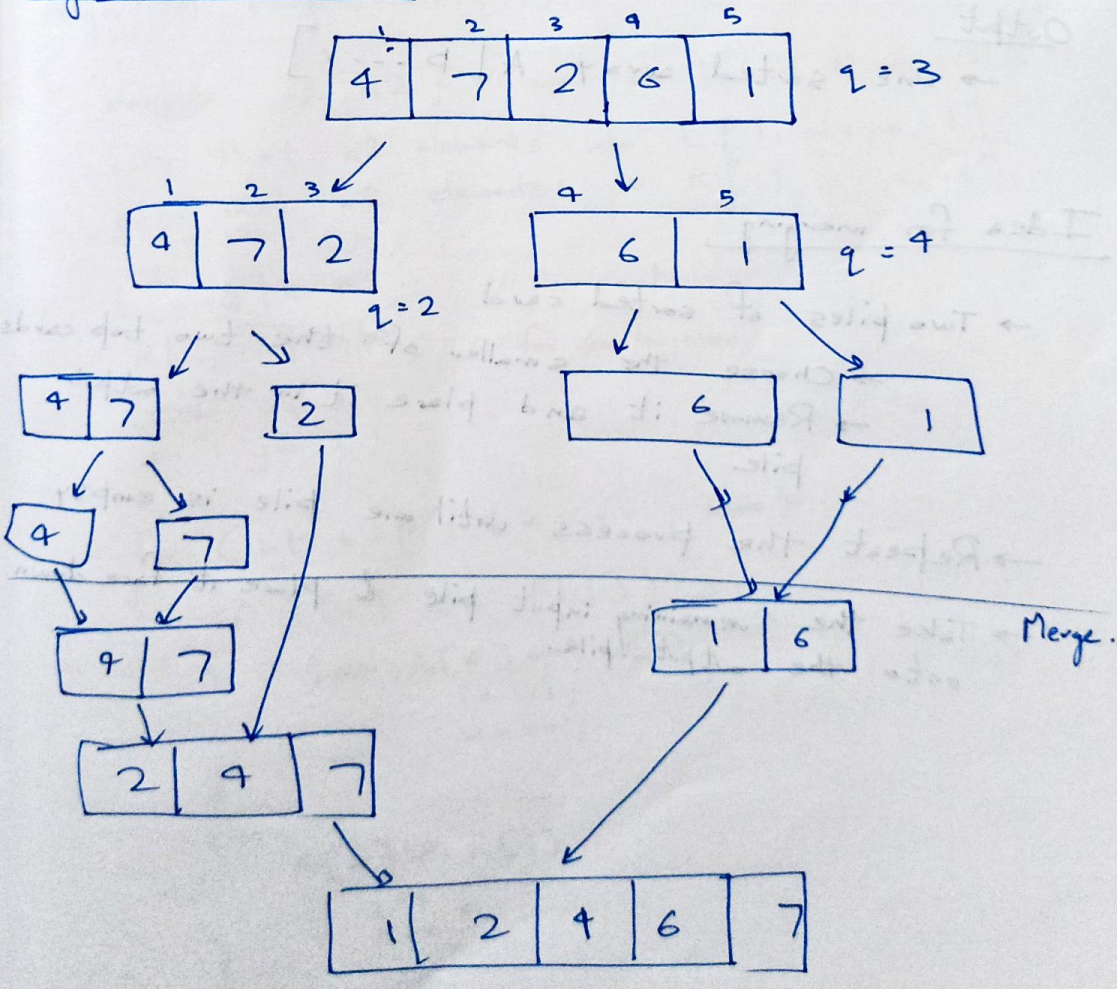
Eg: n power of 2



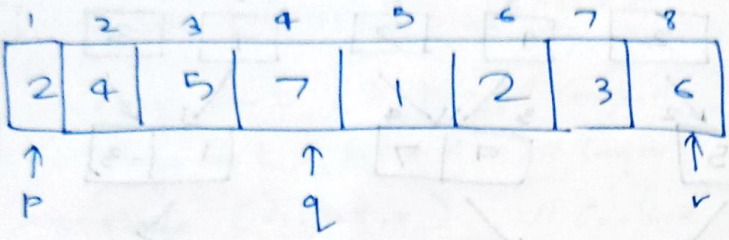
conquer & merge.



Eg not power of 2



Merging (very similar to merging we did in polynomial addition, sparse matrix add^n ...)



Input

→ Array A and indices p, q, r such that

$$p \leq q < r$$

→ Subarrays $A[p, q]$ & $A[q+1, \ldots, r]$ are sorted

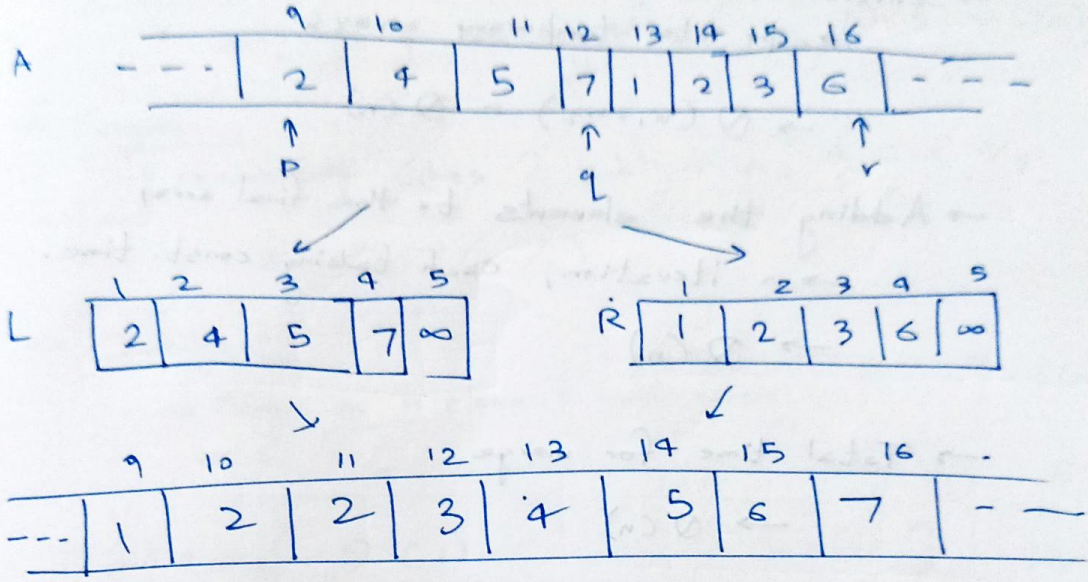
Output

→ One sorted array $A[p \ldots r]$

Idea for merging

→ Two piles of sorted card
  → Choose the smaller of the two top cards
  → Remove it and place it in the output pile
→ Repeat the process until one pile is empty
→ Take the remaining input pile & place it face down onto the output pile.

Eg Merge (A, 9, 12, 16)



Algorithm Merge(A, p, q, r)

{

compute n_1, n_2 ; // size of L and R array

copy first n_1 elements into $L[1, \ldots n_1+1]$

copy next n_2 elements into $R[1, \ldots n_2+1]$

$L[n_1+1] \leftarrow \infty$; // for comparision

$R[n_2+1] \leftarrow \infty$; // for comparision.

$i \leftarrow 1$;

$j \leftarrow 1$;

for (int k = p ; k <= r ;)

{

if (L[i] <= R[j])

{ A[k] ← L[i]

i = i+1 ;

k = k+1 ;

}

else{

A[k] ← R[j]

j++ ;

k++ ;

}

}

add all remaining values as it is

}

Running Time for Merge ( ) [assume last for loop]

→ Initialization
  → copy into temporary arrays
    → $\Theta(n_1+n_2) = \Theta(n)$

→ Adding the elements to the final array
  → n iteration, each taking const. time.
  → $\Theta(n)$

→ Total time for merge.
  → $\Theta(n)$

Analysing Divide and Conquer Algorithms

→ The recurrance is based on the three steps.
  → $T(n)$ ⇒ running time on a problem of size n
  → Divide the problem to a subproblems, each of size n/b : take $D(n)$ time.
  → Conquer (solve) the subproblem, $aT(n/b)$ time.
  → Combine the sol^n $C(n)$

$$\therefore T(n) = \begin{cases} \Theta(1) & n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Merge Sort Running Time

→ **Divide:**
→ computes q as avg of p & r : $\Theta(1) = D(n)$

→ **Conquer:**
→ recursively solves 2 subproblem, of size $n/2$

$\Rightarrow 2T(n/2)$

→ **Combine:**
→ Merge on n element subarray : $\Theta(n) = C(n)$

$$\therefore T(n) = \begin{cases} \Theta(1) & n=1 \\ 2T(n/2) + \Theta(n) & n>1 \end{cases}$$

$\therefore T(1) = \Theta(1)$

$T(2) = 2T(1) + \Theta(2)$

$\vdots$

$T(n-1)$

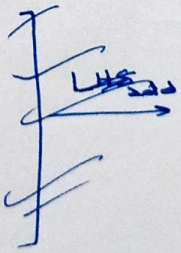
$T(n) = 2T(\frac{n}{2}) + \Theta(n)$

$\therefore T(n) = 2T(\frac{n}{2}) + n$

$T(\frac{n}{2}) = 2T(\frac{n}{4}) + \frac{n}{2}$

$\vdots$

$T(1) = 1$



no. of levels: $\frac{n}{2^k} = 1$

$n = 2^k$

$k = \log_2 n.$

$T(n) = 2T(\frac{n}{2}) + n \quad \times 1$

$T(\frac{n}{2}) = 2T(\frac{n}{4}) + \frac{n}{2} \quad \times 2$

$T(\frac{n}{4}) = 2T(\frac{n}{8}) + \frac{n}{4} \quad \times 4$

$\vdots \qquad \vdots$

$T(1) = 1 \quad \times 2^{\log_2 n} = n$

---

add.

$T(n) = n + n + n + \cdots \quad (k \text{ time})$

$= nk$

$= n \log_2 n$

$= n \log n$