Summary:-

- Algorithm: any well defined computational procedure that takes input and returns output.

- Algorithm analysis:
  - Worst-case : Max (T(n))
  - Best-case : Min (T(n))
  - Avg-case:
    - Approx. to worst if best-case is ~~exp~~ exception.
    - Approx to best if worst-case is exception

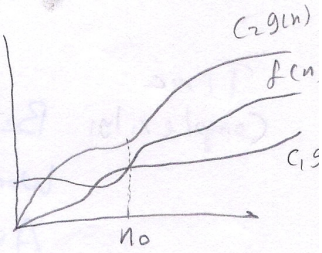
  } Depends on the Input

- Order of Growth: Highest order term represents order of growth, neglect the constants.

  e.g. $\frac{2n+10}{n}$, $\frac{n^2+n+70}{n^2}$, $\frac{n^2+n\log n}{n^2}$

- Theta (Θ): $f(n) = \Theta(g(n))$, c_1, c_2, n_0 +ve consts s.t.
  $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

  g(n) is an asymptotically tight bound for f(n)



- Big O: asymptotic upper bound
  $f(n) = O(g(n))$, c, n_0 +ve consts s.t.
  $0 \le f(n) \le c\, g(n)$ for all $n \ge n_0$

  May be tight or not tight

- Big-Omega: asymptotic lower bound
  $f(n) = \Omega(g(n))$, c, n_0 +ve consts s.t.
  $0 \le c g(n) \le f(n)$ for all $n \ge n_0$.

- little-o: not tight (similar to Big-O but not tight)

- little-omega: similar to Ω but not tight.

Recurrences

(1) Substitution methods

Guess → Prove (math. induction)

Ex: $T(n) = 2T(\lfloor n/2 \rfloor) + n$

- Guess: $O(n \lg n)$
- Prove: $T(n) \leq cn \lg n$, $c > 0$
  - → Assume valid for $m < n$, in particular $m = \lfloor n/2 \rfloor$
  - → $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$
  - → Substitute in the original relation

$$\begin{aligned} T(n) &\leq 2\left[c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)\right] + n \\ &\leq cn \lg(\lfloor n/2 \rfloor) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n \quad (c \geq 1) \end{aligned}$$

Remember:

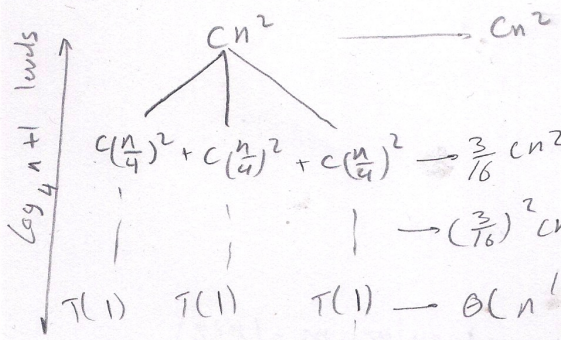
$$\sum_{i=1}^{n} i = \frac{n}{2}(a_1 + a_n)$$

$$\sum_{k=0}^{n} ar^k = \frac{a(1 - r^{n+1})}{1 - r}$$

$$S_\infty = \frac{a_1}{1 - r}, \quad r < 1$$

(2) Recursion tree:

$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$



Note: every subproblem size = $\frac{n}{4^i}$

ex lvl 0 $\frac{n}{4^0} = n$

lvl 1 $\frac{n}{4^1} = \frac{n}{4}$

lvl 2 $\frac{n}{16}$

Note: # of nodes @ depth $i = 3^i$

@ depth $= \log_4 n \rightarrow$ #nodes $= 3^{\log_4 n} = n^{\log_4 3}$ (*1)

$$T(n) = cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + \cdots + (\frac{3}{16})^{\log_4 n - 1} \cdot cn^2 + n^{\log_4 3}$$

$$= \sum_{i=0}^{\log_4 n - 1} cn^2 (\frac{3}{16})^i + \Theta(n^{\log_4 3}) = \frac{(\frac{3}{16})^{\log_4 n} - 1}{(\frac{3}{16}) - 1} cn^2 + \Theta(n^{\log_4 3})$$

Simplify

$$< \sum_{i=0}^{\infty} cn^2 (\frac{3}{16})^i + \Theta(n^{\log_4 3})$$

$$= \frac{1}{1 - \frac{3}{16}} cn^2 + \Theta(n^{\log_4 3}) = \frac{16}{13}cn^2 + \Theta(n^{\log_4 3}) = O(n^2)$$

Verify using substitution:

Guess: $T(n) = O(n^2)$

$T(n) \leq dn^2$

Substitute: $T(n) \leq 3T(\lfloor n/4 \rfloor) + cn^2$

$$\leq 3d(n/4)^2 + cn^2$$

$$= \frac{3}{16}dn^2 + cn^2$$

$$\leq dn^2 \qquad \frac{3}{16}d + c \leq d$$

$$c \leq \frac{13}{16}d$$

$(d \geq \frac{16}{13}c$) condition

Note: We ommit floor and ceiling for simplicity

(3) Master method:

$$T(n) = a\,T(n/b) + f(n)$$

$a \geq 1$, $b > 1$, $f(n)$ is a +ve function

Cases:

1. $f(n) = O(n^{\log_b a - \epsilon})$ for $\epsilon > 0$, $T(n) = \Theta(n^{\log_b a})$

2. $f(n) = \Theta(n^{\log_b a})$, $T(n) = \Theta(n^{\log_b a} \lg n)$

3. $f(n) = \Omega(n^{\log_b a + \epsilon})$ for $\epsilon > 0$ and
   $a\,f(\frac{n}{b}) \leq c f(n)$, $c < 1$ and all large n
   $T(n) = \Theta(f(n))$

Ex: $T(n) = 9\,T(\frac{n}{3}) + n$

$a = 9$ $b = 3$ $f(n) = n = O(n^{\log_3 9 - 1}) \Rightarrow$ Case 1

$T(n) = \Theta(n^{(\log_3 9)}) = \Theta(n^2)$

- Master method proof consists of 2 parts:

1- Proof under assumption: n is exact power of b
   by breaking the problem into 3 Lemmas

2- Generalize part 1

Lemma 1 (4.2):
$a \geq 1$, $b > 1$, $f(n)$ func. defined on exact powers of b

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ aT(\frac{n}{b}) + f(n) & n = b^i, \ i \text{ positive integer} \end{cases}$$

Then:

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(\frac{n}{b^j})$$

Proof: Use recursion-tree method as before

- Lemma 2 (4.3):

$a \geq 1$, $b > 1$, $f(n)$ non-negative defined on exact powers of b

$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right)$ ← from Lemma 1

has

1. $f(n) = O(n^{\log_b a - \epsilon}), \epsilon > 0 \Rightarrow g(n) = O(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a}) \Rightarrow g(n) = \Theta(n^{\log_b a} \lg n)$
3. $a f\left(\frac{n}{b}\right) \leq c f(n)$ for $c < 1$ and all large $n \Rightarrow$ $g(n) = \Theta(f(n))$

Proof.

Case 1: $f(n) = O(n^{\log_b a - \epsilon}) \Rightarrow f\left(\frac{n}{b^j}\right) = O\left(\left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right)$

$\therefore g(n) = O\left[\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right]$

$$n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} \left(\frac{a b^\epsilon}{b^{\log_b a}}\right)^j = n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} (b^\epsilon)^j$$

$$= n^{\log_b a - \epsilon} \left(\frac{b^{\epsilon \log_b n} - 1}{b^\epsilon - 1}\right) = n^{\log_b a - \epsilon} \left[\frac{n^\epsilon - 1}{b^\epsilon - 1}\right] = O(n^{\log_b a})$$

($n^\epsilon - 1 = O(n^\epsilon)$)

$\therefore g(n) = O(n^{\log_b a})$

Case 2: $f(n) = \Theta(n^{\log_b a}) \Rightarrow f\left(\frac{n}{b^j}\right) = \Theta\left(\left(\frac{n}{b^j}\right)^{\log_b a}\right)$

$\therefore g(n) = \Theta\left[\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right]$

$$n^{\log_b a} \sum_{j=0}^{\log_b n - 1} \left(\frac{a}{b^{\log_b a}}\right)^j = n^{\log_b a} \sum_{j=0}^{\log_b n - 1} 1 = n^{\log_b a} \log_b n$$

$\therefore g(n) = \Theta(n^{\log_b a} \log_b n) = \Theta(n^{\log_b a} \lg n)$

Case 3: $\rightarrow c^j f(n) \geq a^j f\left(\frac{n}{b^j}\right)$

$\rightarrow g(n) \leq f(n)\left(\frac{1}{1-c}\right) + O(1)$

$= O(f(n))$

Check reference

Sorting

1 Insertion Sort: Incremental approach

Code:

```
arr [1-indexed]
for j=2 → A.size
    Key = A[j]
    // insert Key in sorted position
    i = j-1
    while(i>0 and A[i]>Key)
        A[i+1] = A[i] //shifting
        i--;
    A[i+1] = Key
```

Analysis:

cost	times
c_1	n
c_2	$n-1$
c_4	$n-1$
c_5	$\sum_{j=2}^{n} t_j$
c_6	$\sum_{j=2}^{n} (t_j - 1)$
c_7	$=$
c_8	$n-1$

$T(n) = \sum_i c_i * times_i$

t_j depends on the input data

Ex: 9 7 6 15 16

7 9 6 15 16

6 7 9 15 16

6 7 9 15 16 ✓

Time Complexity:

Best-Case: Sorted $\Theta(n)$ [The while loop won't excute]

Worst-Case: Sorted in reverse $\Theta(n^2)$

Avg-Case: $t_j = \frac{j}{2}$ (half of the while loop check are true): $\Theta(n^2)$

Space Complexity: $\Theta(1)$ In-place sorting

(2) Merge Sort: Divide-and-Conquer:

- Divide: Divide an n-element sequence to be sorted into 2 subsequences of $\frac{n}{2}$-elements
- Conquer: Sort the 2 subsequences recursively using merge sort.
- Combine: Combine sub problems solution into the original problem solution.

Code:

```
Merge(A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
arrays ⟶ L[1...n1+1], R[1...n2+1]
  for i = 1 → n1
      L[i] = A[p + i - 1]
  for j = 1 → n2
      R[j] = A[q + j]
  L[n1+1] = R[n2+1] = ∞
  i = j = 1
  for k = p → r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i++
      else A[k] = R[j]
          j++
```

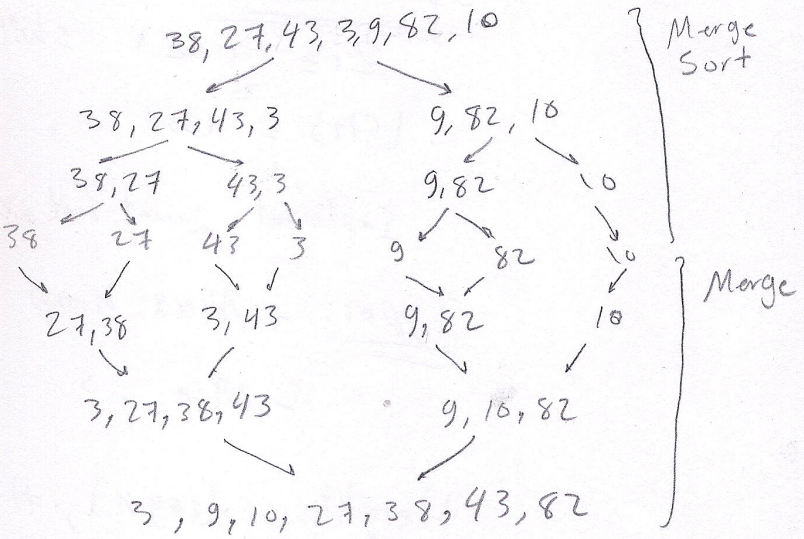
```
Merge Sort(A, p, r)
  if p < r
      q = ⌊(p + r)/2⌋      // Get mid point
      Merge Sort(A, p, q)  // 1st half
      Merge Sort(A, q+1, r) // 2nd half
      Merge(A, p, q, r)    // Merge sorted halves

Merge Sort(A, 1, A.length): // Initial Call
```

Space complexity: $\Theta(n)$, temp. space L, R in merging.

Notes: Small n → Insertion sort
Large n → Merge Sort

Ex

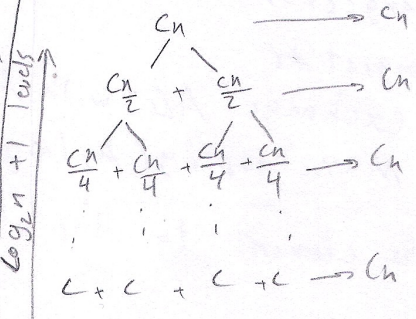


Analysis:

Divide: $\Theta(1)$ → Compute middle
Conquer: $2T(\frac{n}{2})$ – Solve 2 subproblems of size $\frac{n}{2}$
Combine: $\Theta(n)$ → Merging 2 sub arrays L, R, $C(n) = \Theta(n)$

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

$$= \begin{cases} c & , n = 1 \\ 2T(\frac{n}{2}) + cn & , n > 1 \end{cases}$$ where c is constant required to solve problem of size 1

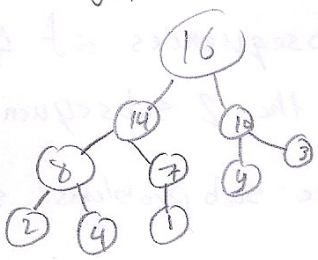


$\therefore$ Total $= (\log_2 n + 1)\, cn = \Theta(n \log_2 n)$

(3) Heap Sort:

Heap:- Data structure (e.g. array), every node has max 2 children

parent(i) = ⌊i/2⌋
left(i) = 2i
right(i) = 2i+1



16, 14, 10, 8, 7, 9, 3, 2, 4, 1

Optimizations:-

2i ≡ i << 1 shift left

⌊i/2⌋ ≡ i >> 1 ; right

Implement parent, left, right funcs as inline.

Types: - Max heap $A[parent(i)] \geq A[i]$

- Min : $A[parent(i)] \leq A[i]$

Height: ⌊log n⌋, #nodes @ height h = $\lceil \frac{n}{2^{h+1}} \rceil$

Operations: $\Theta(\log n)$

```
Max-heapify(A,i):
  l = left(i)
  r = right(i)
  if l ≤ A.hsize and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ A.hsize and A[r] > A[largest]
      Largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      Max-heapify(A, Largest)
```

#Move the element to the right position

$O(\log n)$

```
Build Max heap(A):
  A.hsize = A.length
  for i = ⌊A.length/2⌋ → 1
      Max-heapify(A, i)
```

Note: i starts with last parent

$O(n \lg n)$ ← Max-heapify

```
Heap sort(A):
Build max heap(A):
  for i = A.length → 2
    exchange A[i] → A[1]
    A.hsize --
    Max heapify(A,1)
```

$O(n \log n)$

Used in prority queue.
Same concept

(3) Quick Sort: (In-place), Better Cache Utilization ?!

Code: QuickSort(A, P, r)

```
if p < r
    q = Part(A, P, r)
    QuickSort(A, P, q-1)
    QuickSort(A, q+1, r)
```

```
Part(A, P, r):
    x = A[r]
    i = P - 1
    for j = P to r-1
        if A[j] ≤ x
            i++
            exchange A[i] → A[j]
    exchange A[i+1] → A[r]
    return i+1
```

Worst case (sorted): $T(n) = T(n-1) + \Theta(n)$
$= \Theta(n^2)$

Best case: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$
$= \Theta(n \lg n)$

Improvements: - Use random pivot from the range and put it last item then call part func.

$O(n \lg n)$: expected if distinct elements

For comparison sort:
In the worst case we need $(n \lg n)$ comparisons.

(4) Counting Sort:

Code: CountingSort(A, B, K):

```
let C[0..K] be new array        → Largest element
for i = 0 → K
    C[i] = 0
for j = 1 → A.length
    C[A[j]] ++
for i = 1 → K
    C[i] += C[i-1]
for j = A.length → 1
    B[C[A[j]]] = A[j]
    C[A[j]] --
```

$\Theta(n+K)$
when $K = O(n) \rightarrow \Theta(n)$
Stable, Not in-place

(5) Radix Sort:

```
for i = 1 → d
    use stable sorting (e.g. counting)
    to sort array A on digit i
```

$\Theta(d(n+K))$ ← counting

if we have n b-bits digits → how many bits will each d contain?

$T(n,b) = \Theta(d(n+K)) = \Theta(\frac{b}{r}(n+2^r))$

$r \downarrow \Rightarrow 2^r < n$, $\frac{b}{r} \uparrow\uparrow \Rightarrow (n+2^r) \rightarrow \Theta(n)$

$r \uparrow \Rightarrow 2^r > n$, $\frac{b}{r} \downarrow\downarrow \Rightarrow (n+2^r) \uparrow\uparrow\uparrow\uparrow$

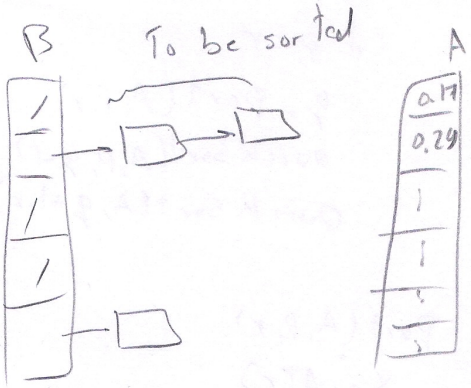
$n + 2^r = \Theta(n) \rightarrow r = \lfloor \log n \rfloor$

$T(n,b) = \Theta(\frac{bn}{\log n})$

if $b = O(\log n) \Rightarrow T(n,b) = \Theta(n)$

[6] Bucket Sort.

```
Bucket sort(A)
    Let B[0...n-1] array
    n = A.length
    for i=0→n-1
        make B[i] empty list
    for i=1→n
        insert A[i] into B[⌊nA[i]⌋]
    for i=0→n-1
        sort B[i] with insertion sort
    concatenate B[0], B[1]...B[n-1]
    in order.
```



If input is uniformally distributed
$O(n)$

To get min. value in A
loop $O(n)$
To get ith smallest element:
Randomized_select(A, p, r, i)

```
if p==r
    return A[p]
q = randomized partation(A,p,r)
// explained in quick sort
k = q-p+1
if i==k
    return A[q]
else if i<k
    return Randomized_select(A,p,q-1,i)
else s        s (A, q+1, r, i)
```

$O(n)$:- $T(n) = T(\frac{n}{2}) + \Theta(n)$
$a=1,\ b=2,\ f=n$
$\therefore O(n)$ on avg

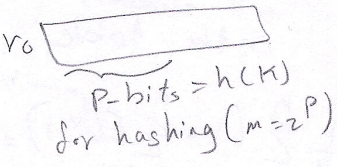
Hashing

Hash table: - size m
- use hash function $h: U \to \{0, 1, \dots, m-1\}$
- Collisions may occur, can be solved:-
  ① Chaining ② open addressing

Hash function:- (Goal reduce collisions)

1- Division method:
$h(k) = k \bmod m$ → should be prime not too close to powers of 2
if $m = 2^p$ then $h(k) =$ lower-p bits of k, it's better if $h(k)$ depends on all k bits

2- Multiplication method:
$h(k) = \lfloor m(kA \bmod 1) \rfloor \quad 0 < A < 1$
↳ $kA - \lfloor kA \rfloor$ = fractional part of kA
m is not critical → most likely make it $= 2^p$ to implement the function in an easy way:-
- Suppose machine word size = w and k fits within 1 word
- Strict A to be in the form $\frac{s}{2^w}$ where $0 < s < 2^w$
- Select A → Get s → fits in one word
- $k \times s = r_1 2^w + r_0$
  (k: w bits, s: w bits; $2w$-bits; upper part r_1; lower part r_0)

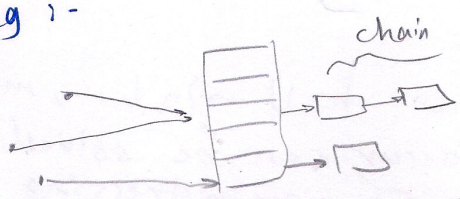


Ex: $k = 123456$, $p = 14$, $m = 2^{14} = 16384$, $w = 32$
A of the form $\frac{s}{2^{32}}$, Let $A \approx 0.618034 \Rightarrow s = 2654435769$
$k \cdot s = 327706022297664 = \underbrace{76300 \cdot 2^{32}}_{r_1} + \underbrace{17612864}_{r_0}$
Get 14 most-significant bits of $r_0 \to h(k) = 67$

- Universal hashing:
Using the same hash function allows adversary to hash all data to the same position, to avoid this determinism problem, pick random hash function from pre-defined hash functions and use it for the whole run (change the next run and so on).

- Collision solutions:

(1) Chaining :-



Load factor $\alpha = \frac{n}{m}$ — elements
— table size

↓ avg chain length

for computing $h(k)$

Unsuccessful search: $\Theta(1+\alpha)$, assuming uniform hashing

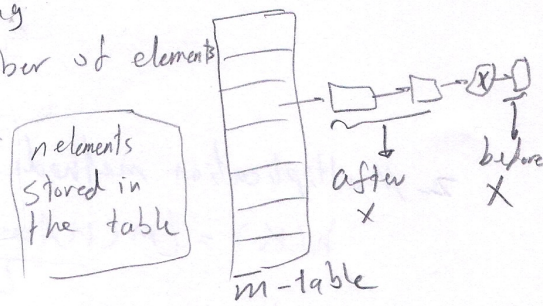
↳ no element in the table has the requested key

if $n = O(m) \Rightarrow \alpha = o(1) \Rightarrow \Theta(1)$

- Successful search takes avg-case $\Theta(1+\alpha)$, assuming uniform hashing

Proof:-

- The # elements examined before finding x = 1 + (number of elements hashing having the same $h(k)$ as x which came after x while saving n elements in the table).



x_i : ith element from n with key k_i

X_{ij} : $h(k_i) = h(k_j)$ → collision

assume uniform hashing $\Rightarrow E[X_{ij}] = \frac{1}{m}$

$$T(n) = E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right]$$

$\frac{1}{n}\sum_{i=1}^{n}$: Sum for all elements in the table

1 + number of (...) mentioned ↑

$\sum_{j=i+1}^{n}$: x_j came after x_i (i+1)

X_{ij} : collision probability

$$= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} \underbrace{E[X_{ij}]}_{1/m}\right) = 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i) = 1 + \frac{1}{nm}\left[\sum_{i=1}^{n} n - \sum_{i=1}^{n} i\right] = 1 + \frac{1}{nm}\left[n^2 - \frac{n(n+1)}{2}\right]$$

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

$\therefore T(n) = \Theta(1+\alpha)$ then if $\underbrace{o(m) = n}_{\text{↳ } n \text{ is proportional to } m} \Rightarrow \alpha = o\left(1 + \frac{o(m)}{m}\right) = o(1+1) = O(1)$

(2) Open addressing.

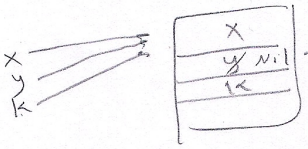
- No lists
- Solve collisions → insert in the next empty table cell after looking (probing) for it using defined equation.

- When searching → you must search all (possible) next elements.

```
Hash_Search (T, K)
    i = 0
    repeat
        j = h(K, i)
        if (T[j] == K) return j
        i++
    until T[j] == Nil or i == m
    return Nil
```

- When deleting (T[i] = Nil), the above search will fail if we delete element which wasn't last one in this cluster

Solution: Mark deleted elements with something else.



→ if deleted then search for K will stop after X

- If keys may be deleted → chaining is better.

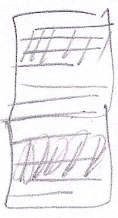
- Linear probing:

$h(K,i) = (h'(K) + i) \bmod m$

search next one if current is filled.

Problem: primary clustring

Empty slot proceeded by i full slots will be filled with prob. $\frac{i+1}{m}$



i+1: Prob. of K to be one of the previous full slots or current slot, in both cases will lead to this empty one to fill it

- Quadratic probing:

$h(K,i) = (h'(K) + c_1 i + c_2 i^2) \bmod m$

h': Auxiliary hash func.

c_1, c_2 → positive

Better than ↪

Problem: Secondary clustring:

if $h(K_1,0) = h(K_2,0) \Rightarrow h(K_1,i) = h(K_2,i)$

So initial positions are important

Double hashing:

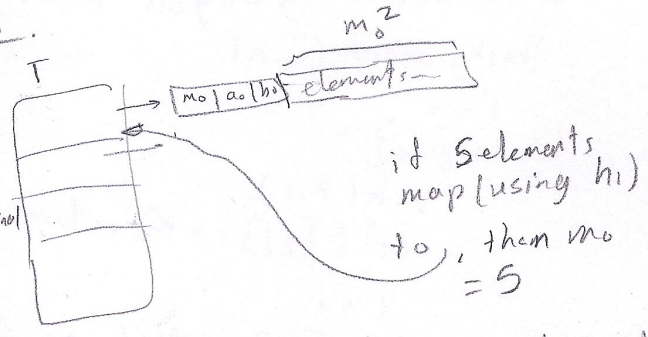
$h(k,i) = (h_1(k) + i h_2(k)) \bmod m$

(h_1, h_2: Auxiliary hash funcs)

if $\alpha = \frac{n}{m} < 1$ and uniform hashing the expected # of probes in an unsuccessful search = $\frac{1}{1-\alpha}$ at most

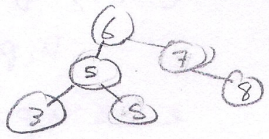
Perfect hashing:

- Good for static data { reserved words in programming langs. / file names in CD-ROM
- O(1) memory access worst case.
- Each cell is hash table with hash function depends on a_i, b_i and then the elements storage with size m_i^2 (m_i the number of elements which map to this cell of the original table), this guarantees (when choosing proper hash functions) that no collisions happen.



Binary Search Trees (BST)

- For all nodes y in left sub-tree of $x \Rightarrow y.key \leq x.key$
- " " " " right " " " $\Rightarrow y.key \geq x.key$



- Operations Complexity: $O(h)$
  for complete/balanced trees: $O(\lg n)$
  for linear chain: $O(n)$

- BST traversal:

Inorder:

```
in(x)
  if x≠NiL
    in(x.left)
    Print x.key
    in(x.right)
```

Preorder:

```
in(x):
  if x≠NiL
    Print(x.key)
    in(x.left)
    in(x.right)
```

Post order:

```
in(x)
  if x≠NiL
  in(x.left)
  in(x.right
  Print(x.key)
```

$\Theta(n)$

- BST search,

```
TS(x,k)
  if(x==NIL or k==x.key)
      return x
  if k<x.key
      return TS(x.left,k)
  else return TS(x.right,k)
```

Iterative

```
while(x≠NIL and k≠x.key)
  if k<x.key
      x=x.left
  else x=x.right
return x
```

- Minimum:-

```
TMi(x)
  while x.left ≠Nil
    x=x.left
  return x
```

- Maximum:

```
TMx(x)
  while x.right ≠Nil
    x=x.right
return x
```

- Successor

Successor of node x is the node with smallest key > x.key

```
Tsu(x)
  if x.right ≠Nil
      return TMi(x.right)
  y=x.p
  while(y≠Nil and x==y.right)
    x=y
    y=y.p
return y
```

- Insert

```
TI(T,z)
  y=NiL
  x=T.root
  while(x≠Nil)
    y=x
    if z.key<x.key
        x=x.left
    else x=x.right
  z.p=y
  if(y==NIL)
    T.root=z
  else if z.key<y.key
    y.left=z
  else y.right=z
```

- Delete:

```
Transplant(T, u, v)
  if (u.p == NIL)
     T.root = v
  else if u == u.p.left
     u.p.left = v
  else u.p.right = v
  if (v ≠ Nil)
     v.p = u.p
```

```
TD (T, z)
  if z.left == Nil
     Transplant(T, z, z.right)
  elseif z.right == Nil
     Transplant(T, z, z.left)
  else y = TM(z.right)
  if y.p ≠ z
     Transplant(T, y, y.right)
     y.right = z.right
     y.right.p = y
  Transplant(T, z, y)
  y.left = z.left
  y.left.p = y
```

- Linked list trees cost O(n) for operations
- To minimize worst case → randomize insertion ⇒ Avg case O(log n)
- AVL, Red-black has worst case O(log n): Self balanced.

- AVL trees:
  - Heights of 2 child subtrees of any node differ by one at most
  - If after insertion or deletion, the condition above breaks, the rebalancing takes place.

- Red black trees:
  - Every node has color [R|B]
  - No path from root to lead is twice as long as another. (balanced)

Properties:

1. Every node is either black or red.
2. The root is black
3. Every leaf (Nil) is black (External nodes)
4. If node is red, then both children are black
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Red-black with n internal nodes has height @ most = $2 \lg(n+1)$

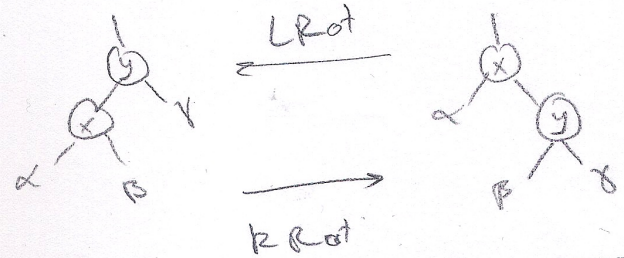
- Black height of x = # of black nodes on any simple path from (not including x) x down to a leaf. bh(x)

Rotations:

- Insert & delete on RBT (Red-Black-tree) takes O(lgn) because they modify it.
- If, after operation, any condition gets broken, we use rotations (right or left) to fix the tree.

Assuming x right child ≠ T.Nil and root's parent is T.Nil

T.Nil → Represents all Nil nodes and root's parent




```
Left-Rotate(T,x):
y = x.right
x.right = y.left
if y.left ≠ T.Nil
    y.left.p = x
y.p = x.p
if x.p == T.Nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```

Insertion: Insert normally, color = red

```
RBInsert(T,z):
y = T.Nil
x = T.root
while x ≠ T.Nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.Nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.Nil
z.right = T.Nil
z.color = red
TBInsertFix(T,z)
```

```
TBInsertFIX(T,z):
while z.p.color = RED
    if z.p = z.p.p.left
        y = z.p.p.right
        if y.color = Red
            z.p.color = black
            y.color = black
            z.p.p.color = Red
            z = z.p.p
        elseif z == z.p.right
            z = z.p
            leftrotate(T,z)
        z.p.color = black
        z.p.p.color = red
        rightrotate(T,z.p.p)
else same as "then" clause ↑↑ with right/left exchanged
```

// Case 1

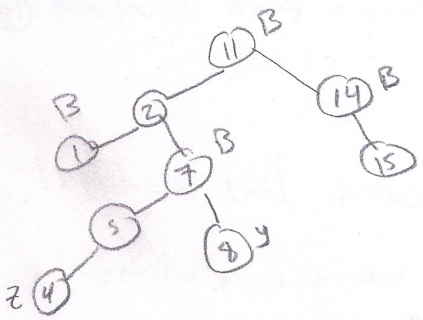
// Case 2

// Case 3

Check back side

T.root color = black // outside while

Note: Insertion may break property 2,4



After 8 insertion, node 5 has red child [Property 4 X]

(18)

This is just useless :/

Chek the reference.