2.2) Divide and Conquer

Divide and Conquer algorithm works according to the following general plan.

1. A problem's instance is divided into several smaller instances of the same problem, ideally about the same size.

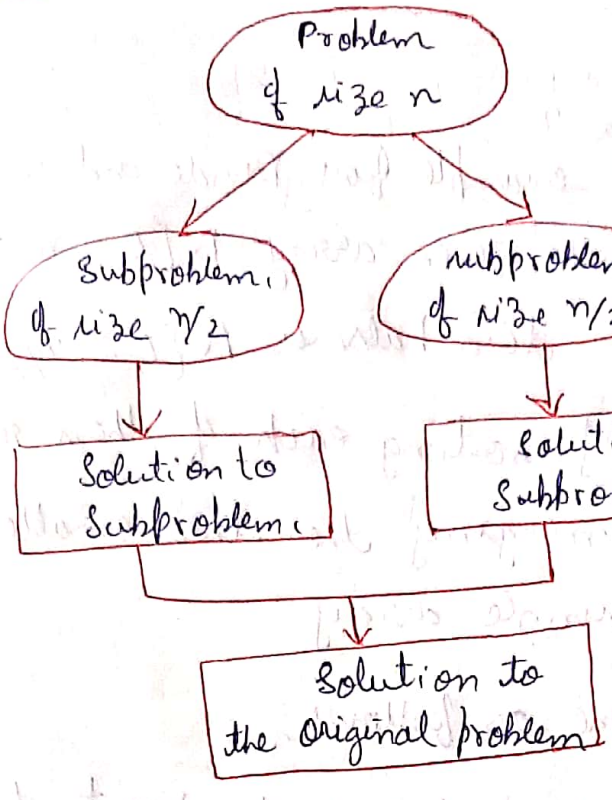
2. The smaller instances are solved

3. If necessary, the solutions obtained for smaller instances are combined to get solution for original problem.

Example: Consider the problem of computing sum of 'n' numbers $a_0, \ldots, a_{n-1}$. if $n > 1$, we can divide the problem into two instances of same problem.
i.e compute the sum of first $\lfloor n/2 \rfloor$ numbers and compute the sum of remaining $\lfloor n/2 \rfloor$ numbers.
Once there two sums is computed, add their values to get the sum

$$a_0 + \ldots + a_{n-1} = (a_0 + \ldots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \ldots + a_{n-1})$$

Divide and conquer technique is ideally suited for parallel computations, in which each subproblem can be solved simultaneously over its own processor.



$f(n)$ = time spent on dividing & combining their solutions

a, b = constants

$T(n/b)$ = Smaller instance

a = Solution to smaller instance

$$T(n) = aT(n/b) + f(n)$$

Master Theorem: If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence eqtn

$a=2, b=3, d=1$
$2 < 3^1 = 2 < 3$
$\Theta n^1 = n$

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

$a=2, b=2, d=1$
$2 = 2^1$ $2=2$
$\Theta n^1 \log n$
$n \log n$

to solve the problem of size n, it is divided into smaller instances n/b. a = Solution to problem of size n. $T(n)$ = Run time

$$T(n) = aT(n/b) + f(n)$$

Eg: Number of additions $A(n)$ made by divide and conquer summation algorithm on inputs of size $n = 2^k$ is

$$A(n) = \underbrace{2A(n/2)}_{\text{Find the sum of first \& second half}} + \underbrace{1}_{\text{Add the solution of I st \& II nd half}}$$

here, $a=2, b=2, d=0$, since $a > b^d$

$$A(n) \in \Theta n^{\log_b a} = \Theta n^{\log_2 2} = \Theta n$$

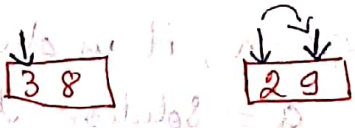
2.2.1) Merge sort

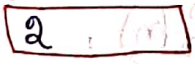
Mergesort is perfect example for Divide and Conquer technique. It sorts the given array $A[0 \ldots n-1]$ by dividing it into two halves $A[0 \ldots n/2-1]$ and $A[\lceil n/2 \rceil \ldots n-1]$ sorting each of them recursively, and then merging the two smaller sorted arrays into single array.

The Merging is done as follows:

1) Two pointers are initialized to point the first elements of two subarrays.

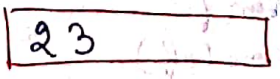
2) The elements pointed are compared with each other and the smallest element is copied to new array and its pointer is incremented to point to next element



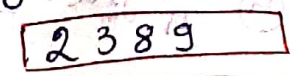


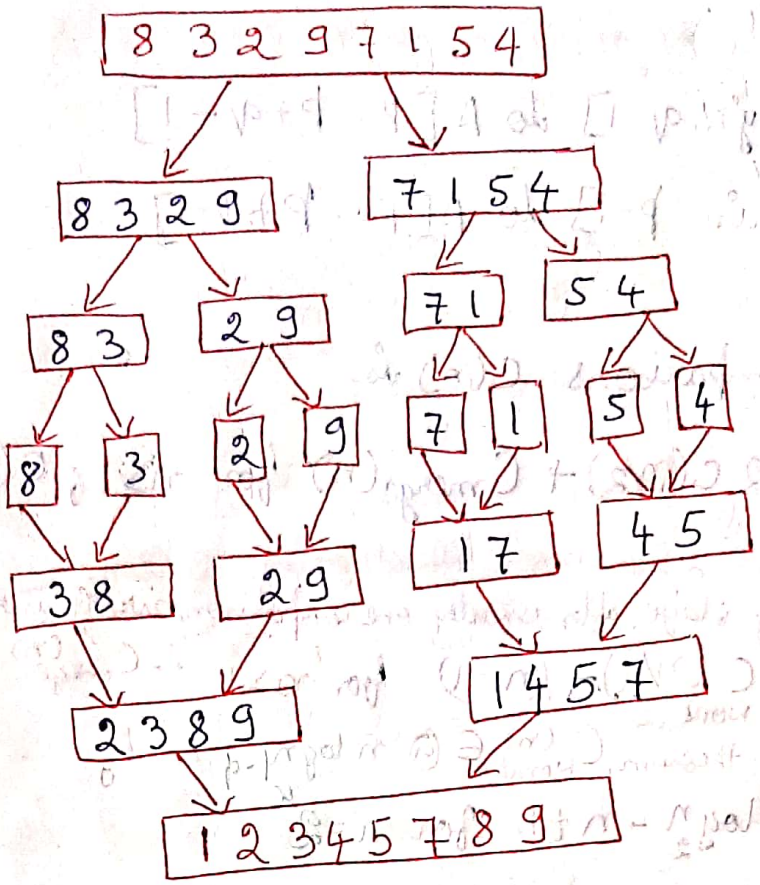
new array

Next 3, and 9 are compared, 3 is smallest so it is copied to new array & its pointer is incremented to next element



Now 8 & 9 are compared, 8 is smallest so its copied to new array





Algorithm: Mergesort(0...n-1)

{ // Sorts the array A[0...n-1]

if n > 1

copy A[0...⌊n/2⌋-1] to B[0...⌊n/2⌋-1]

copy A[⌊n/2⌋...n-1] to C[0...⌈n/2⌉-1]

Mergesort(B[0...⌊n/2⌋-1])

Mergesort(C[0...⌈n/2⌉-1])

Merge(B, C, A)

}

Merge(B[0...p-1], C[0...q-1], A[0...p+q-1])

Algorithm: i ← 0; j ← 0; k ← 0

while i < p and j < q do

if B[i] ≤ C[j]

A[k] ← B[i]; i ← i+1

else A[k] ← C[j]; j ← j+1

k ← k+1

if $i = p$ // when list is exhausted

copy $C[j \ldots q-1]$ to $A[k \ldots p+q-1]$

else copy $B[i \ldots p-1]$ to $A[k \ldots p+q-1]$

No of key comparisons $C(n)$ is

$$C(n) = 2\,C(n/2) + C_{merge}(n) \text{ for } n>1,\ C(1)=0$$

Adding 2 subarrays | Merging

Note: In merging stage, after exactly one comparison, array is reduced by

$$C_{worst}^{(n)} = 2\,C_{worst}^{(n/2)} + (n-1) \text{ for } n>1 \quad \therefore C_{merge}^{(n)} = n-1$$

a worst b

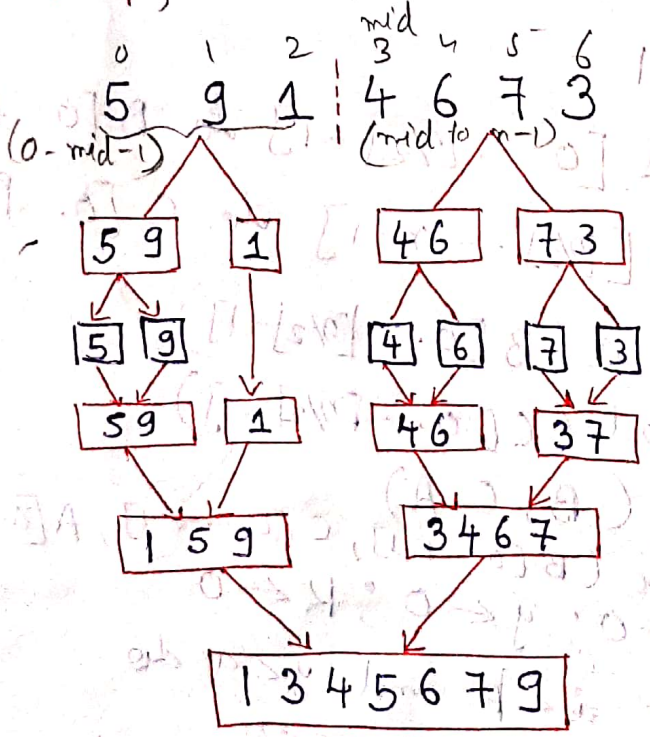
By master's theorem, $C_{worst}^{(n)} \in \Theta(n \log n)$

$$C_w(n) = n \log_2 n - n + 1 \text{ for } n = 2^k$$

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

MergeSort if no of elements in arrays is odd value,

Say $n=7$, $mid = 7/2 = 3.5 = 3$



2 3 4 6

2 3 4 5

2 4 6

2 3 4

$T(n) = 2C_{worst}(n/2) + (n-1)$ for $n>1$

$C_{worst}(1) = 0$

$= 2C\frac{2}{2^{k-1}} + (n-1)$

$= 2C2^{k-1} + (n-1)$

$= 2$

$2 < 2^1 = 2$

$= 2 = 2^1 = 2$

$n^1 \log n$

$\Theta(n^d \log n)$

$= \Theta(n^1 \log n)$

$a = 2, b = 2, d = 0$

$a \quad b^d$

$2 \quad 2^0 = 1$

$2 > 1$

$a = 2 = n$

$b = 2$

$f(n) = n-1$

$f(n) = \Theta(n^d)\log n$

$n^0 \log n = \log n$

$f(n) = n = n$

$a = 2, b = 2$

$d = 1$

$a = 2 \quad b^d = 2^1 = 2$

$a = b^d$

$n^1 \log n$

$d = 1$

$2^2 = 4$

$a < b^d$

$a = 2$

$b = 2$

$d = 2$

n^d

n^2

$n-1$

$T(n) = aT(\frac{n}{b}) + f(n)$

$a \geq 1, b > 1$

$n^d \Rightarrow n^{\log_b a}$

$2 = 2, a = 2$

$\Rightarrow n^{\log_2 2}$

$n^{\log_2 2}$

$n^1 \log n$

$a < b^d$

$a = b^d$

$\Theta(n^d \log n)$

$= \Theta(n^1 \log n)$

2.2.2) Binary Tree Traversals & Related properties

In this section we see how divide and conquer technique can be applied to binary trees. Binary Tree is a tree then can have only two child nodes, not more than that or 0 or 1 child can be there.

Binary Tree T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint trees T_L and T_R called left Subtree and Right Subtree.

Height of tree is defined as the length of longest path from root to leaf. Hence it is calculated as the maximum of (heights of root's left & right subtree) + 1.

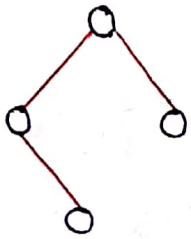
Algorithm: Height(T)

// Computes recursively the height of binary tree

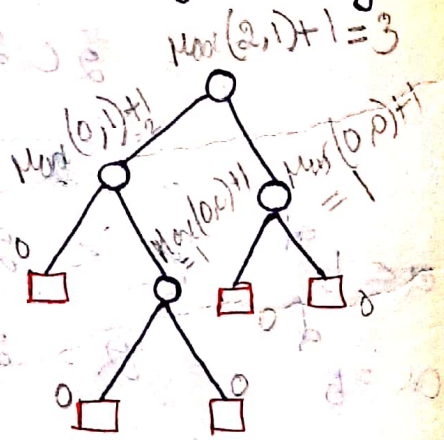
if $T = \emptyset$ return -1

else max{Height(T_L), Height(T_R)} + 1

Example:



Binary Tree



□ → External nodes (x)

○ → Internal nodes (n)

No of Comparisons made to compute max of 2 numbers and No of Additions made by algorithm are same.

$$A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1 \quad \text{for } n(T) > 0$$
$$A(0) = 0$$

Let x → external node and n → internal node

The no of external node in a tree is computed as

$$x = n + 1$$

Let, n_L and x_L = internal & external node of Left Subtree

n_R and x_R = internal & external node of Right Subtree

Since $n > 0$, T has a root hence $n = n_L + n_R + 1$ using the equality assumed to be correct for Left & Right Subtree, we obtain the following.

$$x = x_L + x_R$$
Apply $(x = n+1)$
$$= (n_L + 1) + (n_R + 1)$$
$$= n_L + 1 + n_R + 1$$
$$= (n_L + n_R + 1) + 1$$
Apply $n = n_L + n_R + 1$
$$x = n + 1$$

The no of comparisons to check whether tree is empty is

$$C(n) = n + x$$
Apply $x = n+1$
$$C(n) = n + n + 1 = 2n + 1$$

The number of additions is $A(n) = n$

Traversals of binary tree can be done in 3 ways.

Preorder: Root in visited first then left subtree and then right subtree
Root L R

Inorder: First left subtree is visited then Root and after that Right subtree.
L Root R

Postorder: First left subtree is visited, then right subtree and then Root.
L R Root

2.2.3) Multiplication of Large integers & Strassen's Matrix multiplication.

1) Multiplication of large integers: Let us start with a case of two 2-digit integers 23 & 14.

$23 = 2 \cdot 10^1 + 3 \cdot 10^0$ (20 + 3)

$14 = 1 \cdot 10^1 + 4 \cdot 10^0$ (10 + 4)

Let us multiply them

$$23 * 14 = (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0)$$

$$= (2*1)10^2 + (3*1)10^1 + (2*4)10^1 + (3*4)10^0$$

$$= (2*1)10^2 + (3*1 + 2*4)10^1 + (3*4)10^0$$

$200 + (3+8)10^1 + 12$ (110)

$= 322$

For any pair
$b = b_1 b_0$, The
formula

$C = a *$

$C = C_2 1$

where, $C_2 =$

$C_0 =$

$C_1 =$

Eg: $a = 23$
$a_1 a_0$

$a_1 = 2, a_0 =$

$C_2 = a_1 * b$

$C_0 = a_0 *$

$C_1 = (a_1 + c$

$= (2 + 3$

$= 5 * 5$

$C_1 = 11$

Product, $C =$

$C =$

For any pair of 2 digit numbers $a = a_1 a_0$ and $b = b_1 b_0$, Their product c can be computed by formula

$$c = a * b$$

$$c = c_2 10^2 + c_1 10^1 + c_0$$

where, $c_2 = a_1 * b_1$ is product of their first digits

$c_0 = a_0 * b_0$ is product of their second digits

$$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$$

Eg: $a = 23$, $b = 14$
$a_1\ a_0$ $b_1\ b_0$

$a_1 = 2$, $a_0 = 3$ & $b_1 = 1$, $b_0 = 4$

$c_2 = a_1 * b_1 = 2 \times 1 = 2$

$c_0 = a_0 * b_0 = 3 \times 4 = 12$

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$

$= (2+3) * (1+4) - (2+12)$

$= 5 * 5 - 14$

$c_1 = 11$

Product, $c = c_2 10^2 + c_1 10^1 + c_0$

$= 2(10)^2 + 11(10) + 12$

$= 200 + 110 + 12$

$c = 322$

Now apply the same trick for multiplying 2 n-digit numbers (i.e integers) 'a' and 'b' where n is a +ve even number.

Let us divide the numbers both in middle (divide n conquer). First half of number $a = a_1$

Second half of number $a = a_0$

Illy for b, I^st half $= b_1$

II^nd half $= b_0$

$$a = a_1 a_0 \Rightarrow a = a_1 10^{n/2} + a_0$$

$$b = b_1 b_0 \Rightarrow b = b_1 10^{n/2} + b_0$$

$$c = a * b$$

$$= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$$

$$= (a_1 * b_1) 10^{(n/2)^2} + a_1 10^{n/2} * b_0 + a_0 * b_1 10^{n/2} + a_0 * b_0$$

(Take $10^{n/2}$ common)

$$= (a_1 * b_1) 10^n + (a_1 * b_0 + a_0 * b_1) 10^{n/2} + a_0 * b_0$$

$$C = C_2 10^n + C_1 10^{n/2} + C_0$$

Since multiplication of n-digit numbers requires 3 multiplications. i.e C_2, C_1, C_0 involves 3 multiplication of $n/2$ digit nos

$$M(n) = 3M(n/2) \text{ for } n>1, M(1) = 1$$

for $n = 2^k$

$M(2^k) = 3$

$= 3$

$= 3^2$

$= 3^{k-1}$

$= 3^k$

since $k =$

Strassen's

Consider

say, A & B

-cation of

just now

$C = A$

$\begin{bmatrix} C_{00} & C \\ C_{10} & C \end{bmatrix}$

for $n = 2^k$

$$M(2^k) = 3M(2^{k-1})$$
$$= 3[3M(2^{k-2})]$$
$$= 3^2 M(2^{k-2})$$
$$\vdots$$
$$= 3^{k-1} M(2^{k-(k-1)})$$
$$\vdots$$
$$= 3^k M(2^{k-k}) = 3^k$$

since $k = \log_2 n$, $M(n) = 3^{\log_2 n}$

$$M(n) = n^{\lg_2 3} \approx n^{1.585}$$

Strassen's Matrix Multiplication

Consider we have to multiply two 2×2 matrix say, A & B. Simple brute force method uses 8 multiplication operations. But by using strassen's multiplication just seven multiplications are enough.

$$C = A \times B$$

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where $m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$

$m_2 = (a_{10} + a_{11}) * b_{00}$

$m_3 = a_{00} * (b_{01} - b_{11})$

$m_4 = a_{11} * (b_{10} - b_{00})$

$m_5 = (a_{00} + a_{01}) * b_{11}$

$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$

$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

Example: $A = \begin{matrix} a_{00} & a_{01} \\ \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \\ a_{10} & a_{11} \end{matrix}$ $B = \begin{matrix} b_{00} & b_{01} \\ \begin{bmatrix} 0 & 5 \\ 3 & 1 \end{bmatrix} \\ b_{10} & b_{11} \end{matrix}$

1) $m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$

$= (1+4) * (0+1)$

$\boxed{m_1 = 5}$

2) $m_2 = (a_{10} + a_{11}) * b_{00}$

$= (3+4) * 0$

$\boxed{m_2 = 0}$

3) $m_3 = a_{00} * (b_{01} - b_{11}) = 1 * (5-1)$

$\boxed{m_3 = 4}$

4) $m_4 = a_{11} * (b_{10} - b_{00}) = 4 * (3-0)$

$\boxed{m_4 = 12}$

5) $m_5 = (a_{00} + a_{01}) * b_{11} = (1+2) * 1$

$\boxed{m_5 = 3}$

6) $m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}) = (3-1) * (0+5)$

$\boxed{m_6 = 10}$

7) $m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}) = (2-4) * (3+1)$

$$C = \begin{bmatrix} m_1+m_4-m_5+m_7 & m_3+m_5 \\ m_2+m_4 & m_1+m_3-m_2+m_6 \end{bmatrix}$$

$$= \begin{bmatrix} 5+1.2-3+(-8) & 4+3 \\ 0+12 & 5+4-0+10 \end{bmatrix}$$

$$C = \begin{bmatrix} 6 & 7 \\ 12 & 19 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 0 & 5 \\ 3 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1\times0+2\times3 & 1\times5+2\times1 \\ 3\times0+4\times3 & 3\times5+4\times1 \end{bmatrix}$$

$$= \begin{bmatrix} 6 & 7 \\ 12 & 19 \end{bmatrix}$$

if $M(n)$ = no of multiplications in multiplying two n-by-n matrices

$M(n) = 7M(n/2)$ for $n \geq 1$, $M(1) = 1$

Since, $n = 2^k$

$$M(2^k) = 7M(2^{k-1})$$

$$= 7[7M(2^{k-2})]$$

$$= 7^2 M(2^{k-2})$$

$$= 7^4 M(2^{k-4})$$

$$= 7^k M(2^{k-k})$$

$$M(2^k) = 7^k$$

Since $k = \log_2 n$

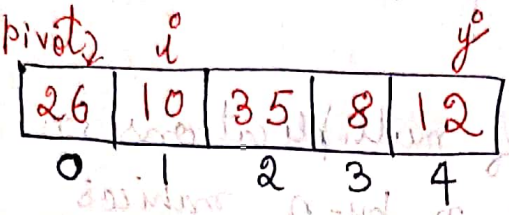
$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807}$$

2.2.4) QuickSort

Quicksort is another algorithm to sort the elements of the given array. Quicksort divides the array into two parts based on pivot element, such that the elements left to pivot element are all smaller than pivot and elements right to pivot element are all larger than pivot.

There are many strategies to select pivot element, Here simply we select the array's first element as pivot element.

Example:



The i^{th} element is compared with the pivot element, if it is smaller than pivot, 'i' is incremented. until we encounter the element greater than pivot element, we keep on incrementing. When the element greater than pivot element is encountered left to right scan is stopped, we begin right-to-left scan. In right to left scan, when element is greater than pivot element, j is decremented. until we encounter the element smaller than pivot element, we keep on decrementing the j. when element smaller than pivot is encountered, Right-to-left scan is stopped.

if $i < j$ exchange, A[i] and A[j] and when we reach the point where i is stopped and j is also stopped. we exchange the pivot element with 'j' element.

Eg:

	i						j
0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	7

		i	j
(26) 10 35 8 12			
0 1 2 3 4			
26 10 35 8 12		1	4
26 10 35 8 12		2 (35 > pivot)	4 steps (because 4 < pivot)
26 10 12 8 35		2	4
26 10 12 8 35		3 (i, j)	4
(26) 10 12 8 35		4 steps (35 > pivot)	3 steps (8 < pivot)
8 10 12 26 35 ✓			

i = 2+1+1
j = 1+1+1+1

Sorted

Best case i/p -> 30, 10, 20, 40, 50

Analysis of Quicksort

Best Case: When does best case occurs, It is when the two segments are exactly equal. if 'i' and 'j' crossover n+1 comparisons are made, if i = j 'n' comparisons

$$C_{BC}^{(n)} = \begin{cases} 0, & n=1 \\ 2C_{BC}^{(n/2)} + n, & \text{if } n>1 \end{cases}$$

assume $n = 2^k$

$$C_{BC}^{(n)} = \Theta(n \log_2 n)$$

Worst Case: Worst case occurs, when all elements are pushed to one side, either left or right. if division did not occur properly i.e equally. for Eg: Entire array contains similar elements, then all will be shown on one side.

Eg: (10) 20 30 40
pivot i

$$C_{WC}(n) = (n+1) + C_{WC}(n-1) \quad \text{for } n > 1$$

$$C_{WC}(1) = 0 \quad , \quad \text{for } n = 1$$

$(n+1)$ because algorithm would make $(n+1)$ comparisons in WC, the remaining elements $(n-1)$ must be still compared recursively.

Eg 5, 7, 8, 6

$$C_{WC}(1) = 0$$

$$C_{WC}(2) = 3 + C_{WC}(1) \rightarrow \text{Apply } C_{WC}(n) = (n+1) + C_{WC}(n-1)$$

$$= 3 + 0$$

$$C_{WC}(2) = 3$$

$$C_{WC}(3) = 4 + C_{WC}(2)$$

$$C_{WC}(3) = 4 + 3$$

$$\vdots$$

$$C_{WC}(n) = (n+1) + n + \ldots + 4 + 3$$

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

Substitute n with $(n+1)$

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

$$= \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2)$$

Average case:

In average case, list may not be equally partitioned into 2 halves, nor pushed one side. Instead pivot element may be placed anywhere in the array.

'p' may range between 0 to n-1, $0 \le p \le n-1$

with probability $1/n$

$$C_{AC}(n) = \frac{1}{n} \sum_{p=0}^{n-1} \left[(n+1) + C_{AC}(p) + C_{AC}(n-1-p) \right] \text{ for } n>1$$

$$C_{AC}(0) = 0 \quad \& \quad C_{AC}(1) = 0$$

$$C_{AC}(n) = 2n \ln n$$

$$\approx 1.38 n \log_2 n.$$

$2n \log_e n = 2n \frac{\log_2 n}{\log_2 e}$

$= 2 \log_2 e \, n \log_2 n$

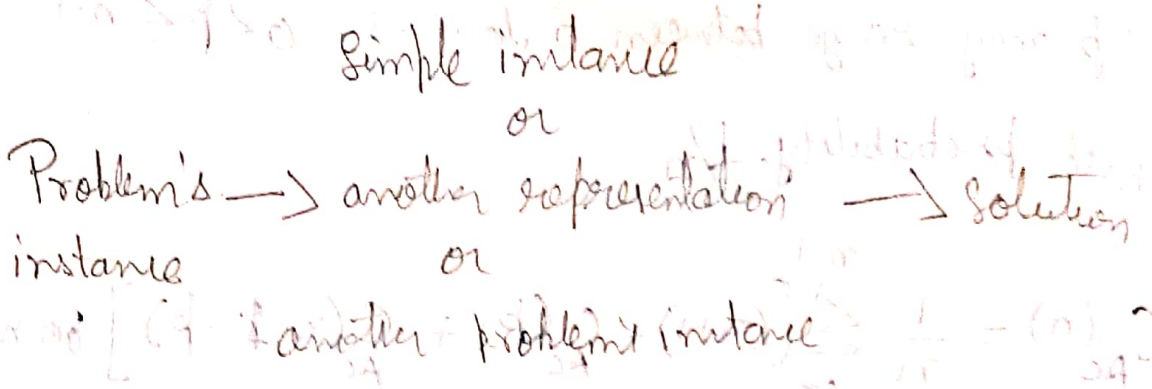
$= 1.38 \, n \log_2 n$

$\log_b a = \frac{\log_e a}{\log_e b}$

Algorithm:

```
p ← A[l]
i ← l; j ← n+1
repeat
    repeat i ← i+1 until A[i] ≥ p
    repeat j ← j-1 until A[j] ≤ p
    Swap (A[i], A[j])
until i ≥ j
Swap (A[i], A[j])
Swap (A[l], A[j])
return j
```

END OF UNIT-2



Transform & Conquer: It works in two-stage procedure,

Stage 1: Transformation

Stage 2: Conquer → Solution

1) Transformation to a simpler or more convenient instance of same - instance simplification

2) Transformation to a different representation of same instance - representation change

3) Transformation to an instance of a different problem for which an algorithm is already available - we call it problem reduction.

UNIT-3 Transform & Conquer

3.1) Balanced Search Trees

Balanced trees will make the algorithms more efficient. Eg Consider binary search tree, searching for a key in binary search tree will reduce the time. because as we know, the property of binary search tree states that the values less than root is on left side of the tree and the values greater than the root are placed on right side of the tree.

so, if we are searching for a key, we dont have to search entire ~~key~~ tree, we will compare the key with root, if key < root, we search on left and if key > root we search on right of the tree.

AVL (Adelson-velsky & Landis)

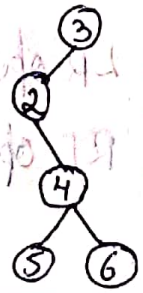
3.1.1) AVL Tree:

AVL tree is a tree where the difference between the height of left subtree and the height of right subtree is not more than 1. It can be either -1, 1 or 0, but not less than 0 and more than or equal to 2. ie difference ranges between -1 to +1.

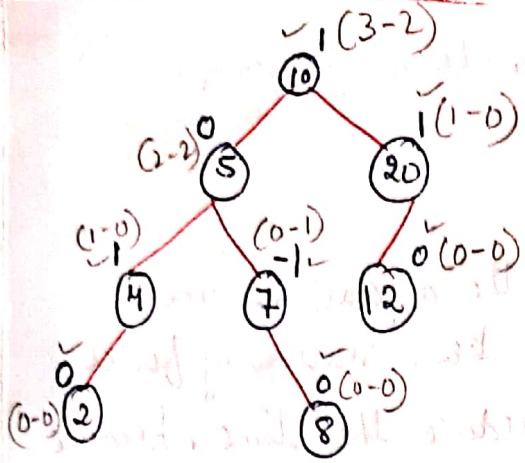
What is height of a tree?

It is the length of the longest path to reach the leaf node from root.

Eg:

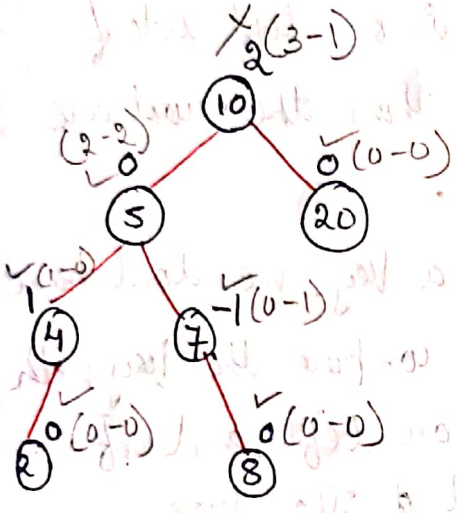


height = 3



(a)

This is an AVL tree, because the difference between the heights of left subtree and right subtree is (0, 1, -1) in this range at all the levels.



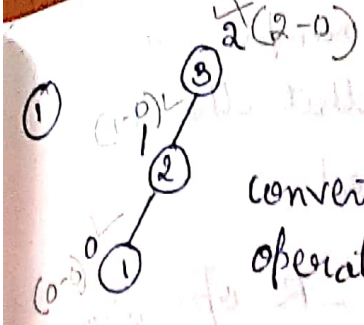
(b)

This tree is not an AVL tree, because the difference between the left and right subtree is 2 at node (10) i.e root node

When a tree is unbalanced, like fig (b) we can convert such trees into balanced tree by performing operations on it. 'Rotation' operation is used to transform a unbalanced tree into balanced tree.

There are 4 types of Rotation operation.

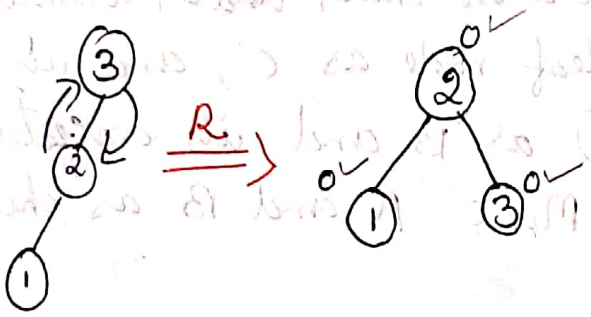
1. L, Rotate Left operation
2. R, Rotate Right operation
3. LR, Left Right operation (Double LR operation)
4. RL, Right Left operation (Double RL operation)



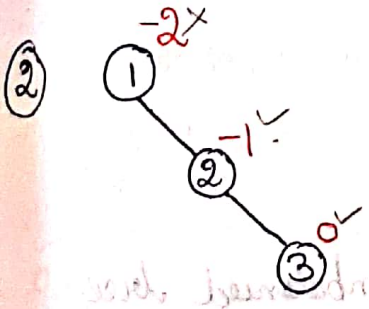
In this example, at node ③ the difference value is 2. so, it is unbalanced. To convert it into balanced tree we apply Rotate operation.

In this case, the node is inserted at left ~~red~~ side, and its parent is on left. The root is having the difference value 2.

node ① is balanced, its parent ② is balanced but its parent ③ is unbalanced. so apply Rotate Right opera-tion. i.e exchange ③ and ②.

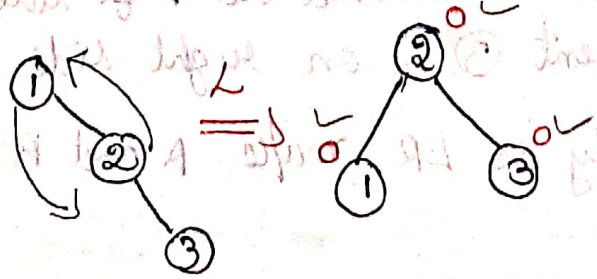


Now it is balanced tree. ③ is placed on right side because it is a binary search tree.

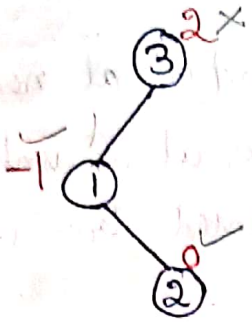


In this case also, the tree is unbalanced. because the root nodes difference value is -2.

Here the leaf node is inserted to right side of tree and all its ancestors are on right side. so, Apply Left Rotate operation to balance this tree.



(3)

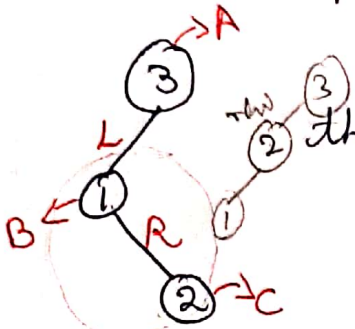


In this example, again we can see that the tree is unbalanced.

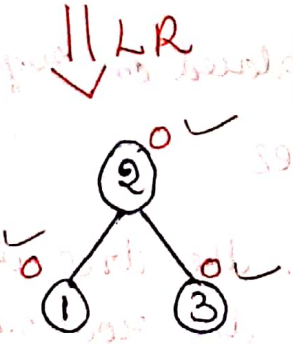
By applying LR operation i.e Left Right operation, we can balance this tree.

How does LR operation works?

In this above tree, last leaf node is on right side and its parent is on Left side. so, apply LR rotate operation.

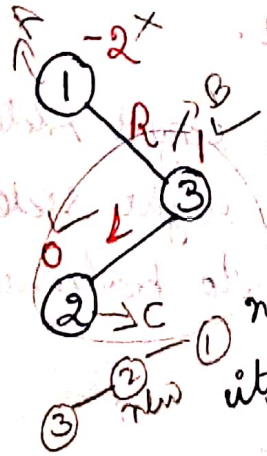


LR operation states that, consider the last leaf node as 'c', and its parent (1) as B and its ancestor as A. Make A and B as children of C.



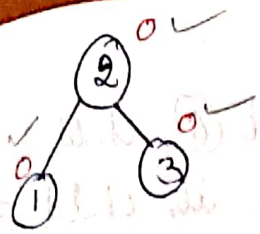
LR is Double Rotation: It applies Rotate left on left subtree, and then for new root, Right Rotate. So it is mix of both Left Rotate & Right.

(4)



This is again an unbalanced tree. To transform it into balanced tree, we apply RL operation. Why RL?, because node (2) is inserted to left side and its parent (3) is on right side

RL works similarly as LR, make A and B as children of C. Right Rotate on Right Subtree, then but new node Rotate Left

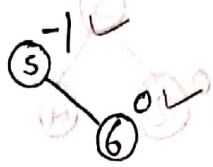


3.1.2) Construction of AVL tree for given list.

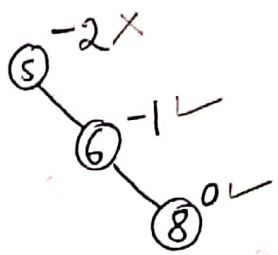
Assume you have given a list 5, 6, 8, 3, 2, 4, 7, how do construct AVL tree for this list?

1) insert 5

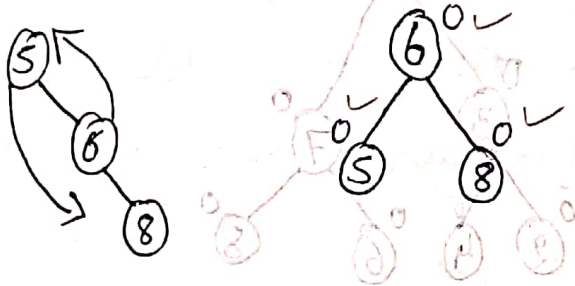
(2) insert 6, 6>5



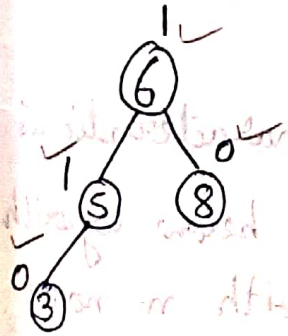
(3) insert 8, 8>6



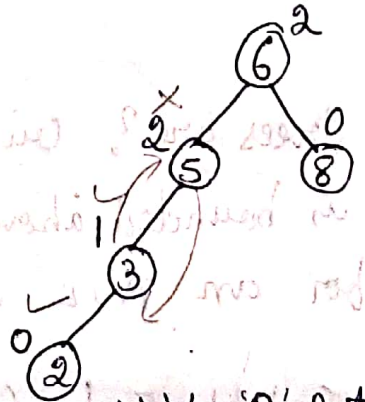
Now tree is unbalanced at step 3. Apply Rotate L operation



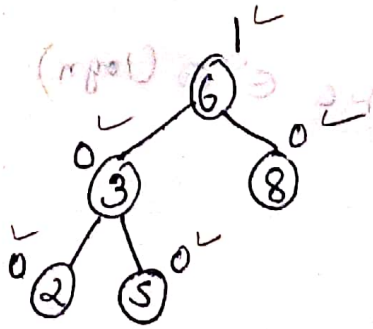
(4) insert 3, 3<5



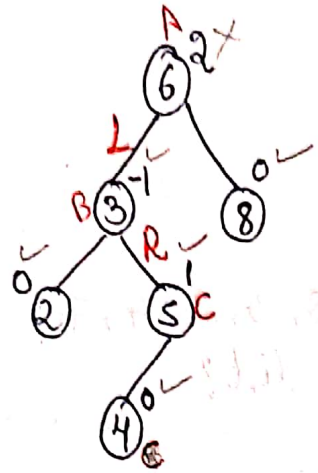
(5) insert 2, 2<3



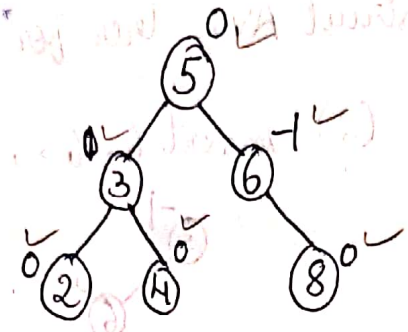
Apply 'R' Rotate operation at (5)



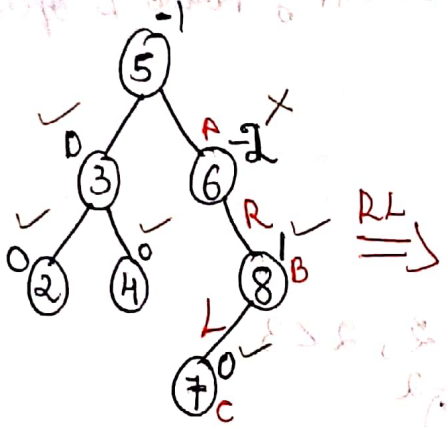
(6) insert 4, 4<5



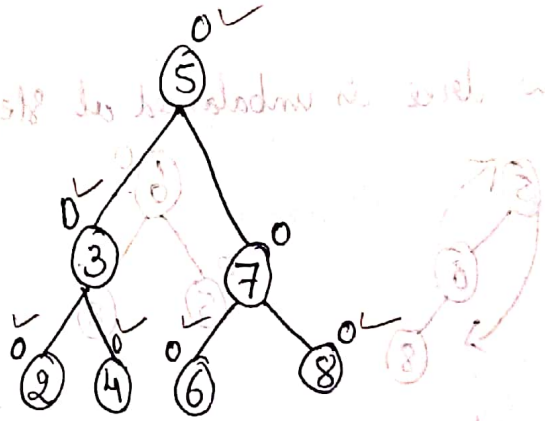
Tree is unbalanced at (6), child of (6) is (3) i.e on left and it's child (5) is on right side. So apply LR rotation operation. Make A & B as children of C



(7) insert 7, 7<8

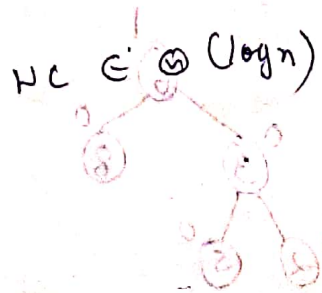


RL ⟹



How efficient AVL trees are? Critical characteristic is tree's height. It is bounded above and below logarithmic function. For an AVL tree with 'n' nodes

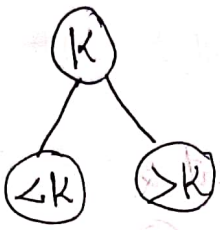
$$\lfloor \log_2 n \rfloor \leq h < 1.4405 \log_2(n+2) - 1.3277$$



3.1.2) 2-3 Trees

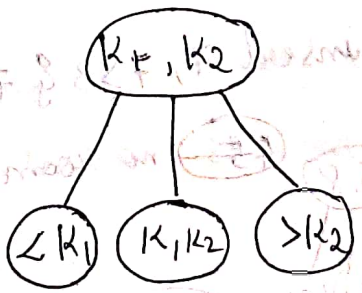
A 2-3 tree is a tree that can have nodes of 2 type
1) 2-node (2) 3-node.

A 2-node contains single key 'K' and two child nodes such that left child < key and right child > key

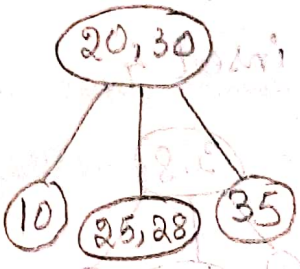


2-node

A 3-node contains two keys i.e K_1, K_2 and it has 3 child nodes. left most child $< K_1$ and rightmost child $> K_2$ and middle child is between K_1 and K_2



Eg:



3-Node

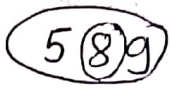
2-3 tree is always height balanced, i.e all of its leaf must be on same level.

Next is how do construct 2-3 tree for a given list?
Say, the list is as follows.

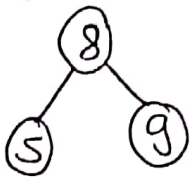
9, 5, 8, 3, 2, 4, 7

given list: 9, 5, 8, 3, 2, 4, 7

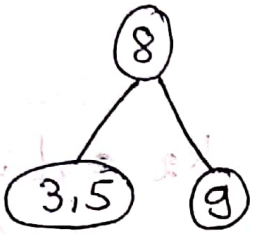
1) insert 9 (2) insert 5, 5<9 (3) insert 8,



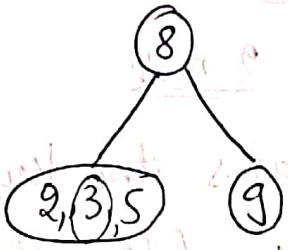
no room
push middle element & split



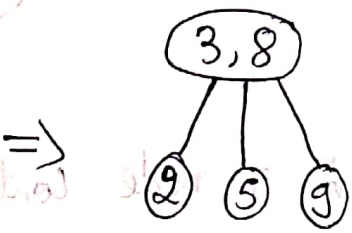
(4) insert, 3



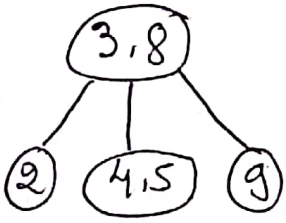
(5) insert 2



no room
push middle element & split



(6) insert 4,



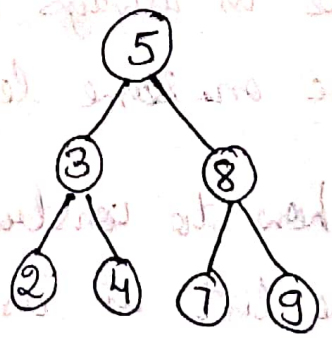
(7) insert 7, 7>3 & 7<8

~~4 5 8~~ no room 4 5 7

push 5 & split

3 5 8 no room

again push and split.



A 2-3 tree of height 'h' with smallest no of keys is a full tree of 2-nodes. ∴ for any 2-3 tree with height 'h' and 'n' nodes. The efficiency of dictionary operation in dependent on height of the tree.

$n \geq 1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1$

$1.2^0 + 1.2^1 + \cdots + 1.2^h$

for 1 node to a key, 2 child

$1(1+2+\cdots 2^h)$

Apply ①

hence $h \leq \log_2(n+1) - 1$ ↪ $\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = 2^{n+1} - 1$ ↪ ①

Any 2-3 tree, for 2 keys and 3 child

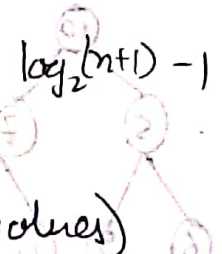
$n \leq 2.1 + 2.3 + \cdots + 2.3^h$

$2.3^0 \quad 2.3^1$

$= 2(1 + 3 + \cdots + 3^h)$

$= 3^{h+1} - 1$

$h \geq \log_3(n+1) - 1 \quad \in \Omega(\log n)$



$\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1$

3.1.3) Heaps and Heapsort

Heap is defined as a binary tree with keys (values) assigned to its node provided the following two conditions are met:

1) Binary tree is a complete tree, i.e all its level are full except possibly the last level, where only some rightmost leaves may be missing.

2) Parental dominance requirement - The value of each node must be greater than its child node.

For Eg:

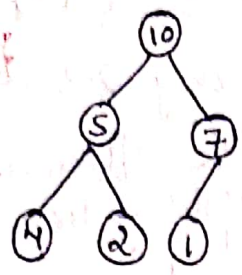
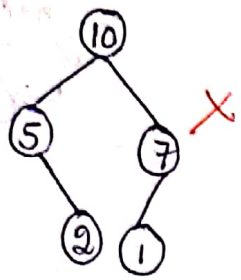


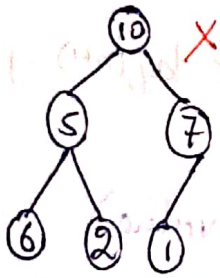
Fig 1:

This is a ~~heap~~ heap, because each node value is ~~less~~ greater than its child. and all levels are complete, In last level only right child is missing. so it satisfies the 2 conditions of heap.



This is not a heap, because after root, first level is incomplete and for node (5) left child is missing. It violates rule.

Fig 2



This is not a heap, though all its level are complete except last level. but observe the child value of node (5) i.e 6 > 5. again it violates rule.

Fig 3

List of properties of heap.

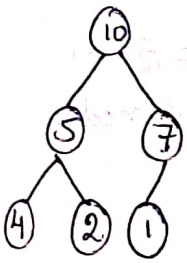
1) There exists exactly one essentially complete binary tree with n nodes.
2) Root of a heap always contains its largest element.
3) A node of a heap considered with all its descendants is also a heap.

4) A heap can be implemented as an array by recording its element in top-down, left to right fashion.
It is convenient to store the heap's elements in positions 1 to n of such an array, leaving H[0] either unused or putting there a sentinel whose value is greater than every element in heap. In such rep

1) The parental node keys will be in first $(n/2)$ positions of array, while the leaf will occupy the last $(n/2)$ position.

2) The children of a key in array's parental position i will be in position $2i$ and $2i+1$.

Heaps and its array representation

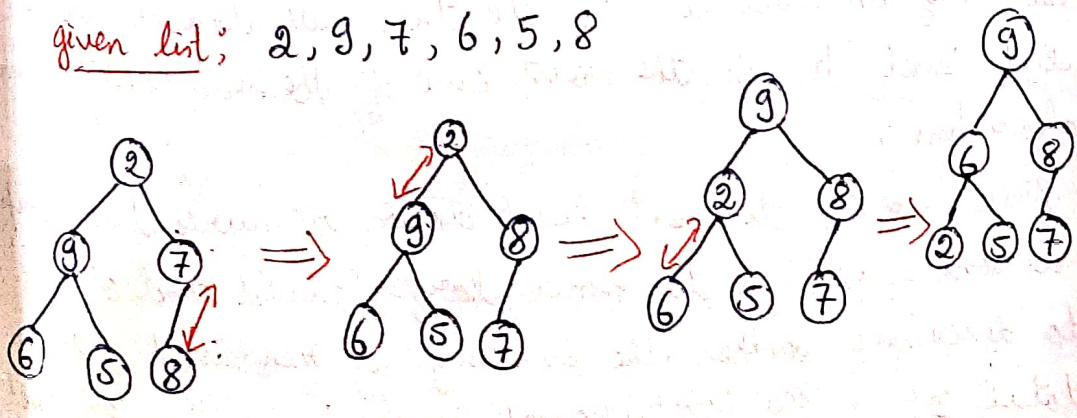


0	1	2	3	4	5	6
	10	5	7	4	2	1

H[0] unused — Parents — leaves

Construction of a heap for the given list (Bottom up)

given list: 2, 9, 7, 6, 5, 8



Algorithm: Heap bottom up $H[1..n]$

```
for i ← n/2 downto 1 do
{ k ← i; v ← H[k]
  heap ← false
  while not heap and 2*k ≤ n do
  { j ← 2*k
    if j < n // there are 2 children
    { if H[j] < H[j+1]
        j ← j+1 }
    if v ≥ H[j]
      heap ← true
    else { H[k] ← H[j]; k ← j }
    H[k] ← v
  }
}
```

How efficient is this algorithm? Assume, $n = 2^k - 1$, so that heap's tree is full. i.e largest no of nodes occur on each level.

Let h = height of the tree

$h = \log_2 n$.

Each key on level 'i' of the tree will travel to the leaf level 'h' in the worst case of the heap construction algorithm.

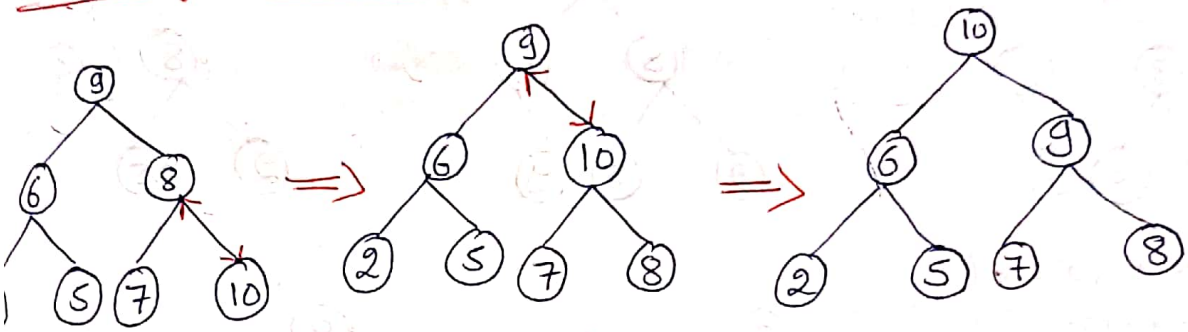
Since moving to next level down requires two comparisons - one is to find larger child and other to determine whether the exchange is required - the total no of key comparisons involving a key on level 'i' will be $2(h-i)$.

$$C_{worst}^{(n)} = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i)$$

$$= \sum_{i=0}^{h-1} 2(h-i)\,2^{i}$$

$$= 2(n - \log_2(n+1))$$

Inserting a key into the heap constructed.



We can insert a new key K into heap by using the up-down heap construction. First attach the new node with key 'K' into the last leaf of the existing heap. Then compare the 'K' with its parent's key. if the 'K' is greater than its parent's key, then swap these two and compare with its new parent. This continues until K is not greater than its parent or it reaches the root.

Since the height of a heap with 'n' nodes is about $\log_2 n$, the time efficiency of insertion is $O(\log n)$.

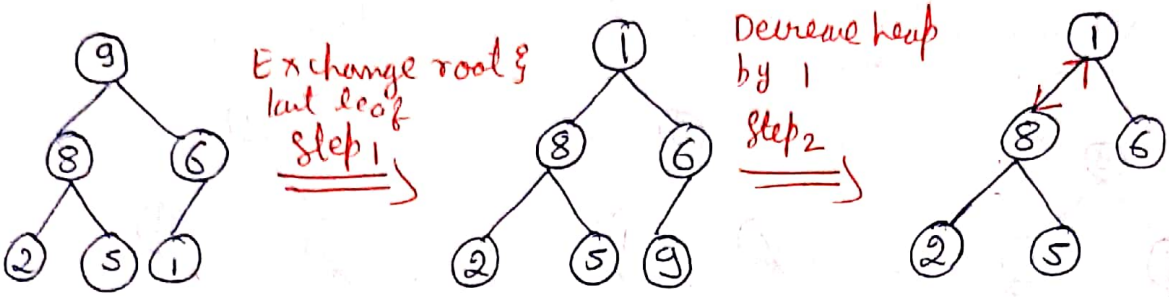
How to delete an item from a heap

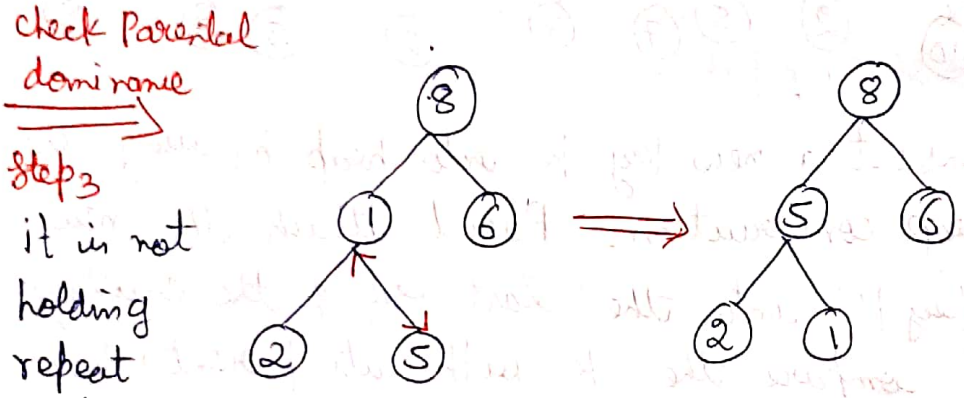
Here we discuss only the most important case of deleting the root's key. It can be done by following the below algorithm.

Algorithm: Step1: Exchange the root's key with the last key K of the heap.

Step2: Decrease the heap's size by 1.

Step3: Heapify the smaller tree by shifting K down in the same way as of bottom-up approach.
Check parental dominance, if it holds we are done





Heapsort: It is an inserting sorting algorithm discovered by J.W.J Williams. It is a 2 stage algorithm.

Step1: Construct a heap for given array.

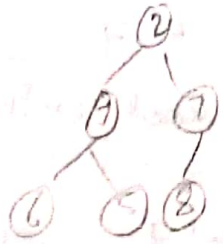
Step2: Apply the root-deletion operation (n-1) times to remaining heap.

Consider the given array is 2, 9, 7, 6, 5, 8.
Now to sort this list we use heapsort algorithm.

Stage 1 :

2 9 7 6 5 8

Parents childs



1) Apply Bottom up approach, i.e compare the last leaf with its parent. so, 8 & 7 is compared. 8>7, swap.

2 9 8 6 5 7

2) Next child i.e 5 is compared with its parent, i.e 5 & 9 is compared. 9>5 No swap. Then next child 6 is compared with 9, 9>6 No swap

2 9 8 6 5 7

3) Next 8, 9 are children of 2 i.e 8, 9 > 2. but 9 is greater of all. so, swap 2 and 9

9 2 8 6 5 7

4) Next 6, 5 are children of 2, 6, 5 > 2 so swap 6 and 2

9 6 8 2 5 7

Now, Parent of 7 is 8 i.e 8 > 7 ✓

Parent of 2, 5 is 6 i.e 6 > 2, 5 ✓

Parent of 6, 8 is 9 i.e 9 > 6, 8 ✓

It is a ~~sorted~~ heap

Stage 2: Maximum deletions

9 6 8 2 5 7

= Exchange root with last leaf i.e 9 & 7

1) 7 6 8 2 5 | 9

Delete the last leaf. after deletion we get

7 6 8 2 5

Next, compare parental dominance. i.e 8 has no child. 2,5 parent is 6 i.e 6 > 2,5. and 6,8 parent is 7 but 8 > 7 no swap

8 6 7 2 5

=

2) 5 6 7 2 | 8 delete 8

5 6 7 2 -> 6,7 children of 5 but 7 > 5

so swap

7 6 5 2

=

3) 2 6 5 | 7 delete 7

2 6 5 , 6,5 children of 2, 6 > 2 & swap

6 2 5

=

$C(n) \leq 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \ldots + 2\lfloor \log_2 1 \rfloor \leq 2 \sum_{i=1}^{n-1} \log_2 i$ if $i = (n-1)$

$\leq 2 \sum_{i=1}^{n-1} \log_2(n-1) = 2 \log_2(n-1) \sum_{i=1}^{n-1} 1$

$= 2(n-1) \log_2(n-1)$

4) 5 2 | 6 delete 6

5 2

=

5) 2 | 5 delete 5

6) 2

∴ The array elements are deleted in decreasing order (9, 8, 7, 6, 5, 2). array representation of heap, element being deleted is placed last.

i.e Bottom up Approach

1.4) Space and Time Tradeoffs

Space and Time Tradeoff algorithm design are a well known issue for both practical and theoritical analysis.

Eg; $f(n) = 2(n)+1$

for Example:- Consider problem of computing values of a function, we can compute the function's value and store them in a table. This is exactly what humans do before invention of computer. Though such idea is vanished these days, we can make use of this concept in developing algorithm. This approach is known as input enhancement. we discuss the algorithm which uses this technique.

1) Counting methods for sorting.

2) Horspool algorithm.

Another approach is to make use of extra space to facilitate faster / more flexible access of data. This is known as prestructuring.

i.e Some procesing is done before a problem in question is solved. we illustrate this approach in 1) Hashing

2) Indexing with B-Trees

$2n\log_2 n$

3.1.4.1) Sorting by Counting

In this method we count for each element of a list, the total number of elements smaller than this element and record the results in table. This numbers will indicate the position of the elements in the sorted list.

Eg: 62 31 84 96 19 47

0 1 2 3 4 5

col[0] col[1] col[2] col[3] col[4] col[5]

columns = j

62	31	84	96	19	47	
0	0	0	0	0	0	→ initially
3	0	1	1	0	0	
	1	2	2	0	1	rows = i
		4	3	0	1	
			5	0	1	
				0	2	
3	1	4	5	0	2	

No of elements smaller than 62 ←

19, 31, 47, 62, 84, 96

Note: once we fill the no of elements smaller than 62, next we see each element in the remaining list. if that element is smaller we retain '0' as it is and if the element is greater, we increment its value by 1.

i.e: for 31, 31 < 62 so retain 0 but 84 > 62 so we increment by 1. same thing continues for rest of the rows

Last and final row indicates the indices where the elements should be in the sorted array.

19, 31, 47, 62, 84, 96

Algorithm: for $i \leftarrow 0$ to $n-1$ do count[i] $\leftarrow 0$

```
for i ← 0 to n-2 do
{ for j ← i+1 to n-1 do
  { if A[i] < A[j]
      count[j] ← count[j] + 1
    else
      count[i] ← count[i] + 1
  }
}
for i ← 0 to n-1 do S[count[i]] ← A[i]
return S
```

S[3] = A[0], S[1] = A[1]
31
62 S[4] = 84

No of times basic operation i.e comparisons is executed is equal to the sum we encountered.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$\sum_{i=l}^{u} 1 = u - l + 1$$

$$\sum_{j=i+1}^{n-1} 1 = (n-1) - (i+1) + 1$$

$$= n - 1 - i + 1$$

$$= n - i - 1$$

$$\sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}$$

for $1+2+3+\ldots(n-1)$

Apply std formula $\frac{n(n+1)}{2}$

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

$$\sum_{i=0}^{n-2} (n-1-i)$$

for $i=0 \Rightarrow n-1$

$i=1 \Rightarrow n-2$

$i=2 \Rightarrow n-3$

$i=n \Rightarrow n-1-n$

$i=n-2 \Rightarrow (n-1)-(n-2)$

$= n-1-n+2$

$i = n-2 = 1$, write sumation

$(n-1)+(n-2)+(n-3)+\ldots+1$

Distribution Counting

Let us consider a more realistic situation of sorting a list of items with other information associated with their keys so we cant overwrite the list elements. Then we can copy elements into new array S[0..n-1]. The elements of A whose values are equal to lowest possible value l are copied into first F[0] elements of S, i.e position 0 to F(0)-1, the elements of value l+1 are copied to positions from F(0) to (F(0) + F(1)) - 1 and so on. Since such accumulated sums of frequencies are called a distribution in statistics and method known as distributed counting.

Eg: consider sorting a array

0	1	2	3	4	5
13	11	12	13	12	12

l = 11, u = 13, F = {11, 12, 13}

These values come from the set {11, 12, 13}. and should not be overwritten in process of sorting. The frequency and distribution arrays are as follows:

Note: frequency:- no of times a particular element has occured.

Distribution values:- scan the array from right to left and note down the index at which the element 11 i.e the first element of set, has appeared.

0	1	2	3	4	5
13	11	12	13	12	12

distribution value of 11 = 1

next element i.e 12, we add the previous element's distribution value + the frequency of the element 12.

1 2 3 4 5 6
11, 12, 12, 12, 13, 13

Array values	11	12	13
Frequency	1	3	2
Distribution value	1 (1+0)	4	6

A[5] = 12 ✓ D
A[4] = 12
A[3] = 13
A[2] = 12
A[1] = 11
A[0] = 13

0	1	2
1	4	6
1	3	6
1	2	6
1	2	5
1	1	5
0	1	5

0 1 4

0	1	2	3	4	5
			12		
		12			.
					13
	12				
11					
				13	

11, 12, 12, 12, 13, 13

Note that the distribution values indicate the proper positions for the last occurences of their elements in the final sorted array. If we index array positions from 0 to n-1, distribution values must be reduced by 1 to get corresponding element positions.

It is more convenient to process the input array right to left. For Eg: last value is 12, and i.e. its distribution value is 4, we place 12 in position 4-1 = 3 of array B that will hold the sorted list. Then we decrease the 12's distribution value by 1 and proceed to next (from right) element in given array.

Algorithm: I/p :- Array A[0...n-1] of integer bt l and
O/p :- Array S[0...n-1] of A's element in ascending order.

Note:- None of the for loop is nested (13-11)

→ for j ← 0 to u - l (2, 13, 12) do D[j] ← 0

→ for i ← 0 to n-1 do D[A[i]-l] ← D[A[i]-l] + 1 // Cal DV

→ for j ← 1 to u-l (13-11) do D[j] ← D[j-1] + D[j] // Cal DV

→ for i ← n-1 down to 0 do // Rearr DV
{
j ← A[i] - l
S[D[j] - 1] ← A[i]
D[j] ← D[j] - 1
}
return S

3.1.4.2) Input enhancement in String Matching

In brute force string matching, consider 'm' characters are there in pattern string and 'n' characters in Text string. This method matches corresponding pairs of characters in pattern and text string from left to right, If mismatch occurs shift the entire pattern by one position to right for next trial. Since the largest no of such trials is n-m+1 and in worst case 'm' comparisons is m(n-m+1) ∈ O(nm).

Nevertheless, several better algorithms have been discovered. They exploit the input enhancement idea preprocess the pattern to get some information and store this information in a table, then

use this information in string matching.

Horspool's Algorithm

Consider we want to search for the pattern BARBER in some text string S[0...n-1].

Let 'c' be the some character of the text that is alliongned with last character of pattern string

S_0 ..... c ....S_{n-1}

BARBER

Starting with the last 'R' of the pattern and moving right to left, we compare the corresponding pairs of characters in the pattern and text. If all the pattern characters match succesfully, a matching substring is found.

If we encounter a mismatch, we need to shift the pattern to the right. Horspool's algorithm determines the size of shift by looking at the character 'c' of the text that was aligned against the last character of pattern. Following 4 possibilities may occur.

Case 1: if 'c' is not matching with last character of pattern, shift the entire pattern to right side. Eg: Consider we have S as 'c'

S_0 ..... S ..... S_{n-1}

BARBER

BARBER

R ≠ S, so shift entire pattern

& also 'S' is not present in the pattern string.

Case 2 : If 'C' is not matching with pattern's last character but 'C' is present in pattern means, shift the pattern by the no of bits = index of the occurs of 'C' in pattern

S_0 ..... B ..... S_{n-1}

BARBER
BARBER

$R \neq B$, but B is present in the pattern. Position of the B's first occurence from right to left in pattern string is 2. so shift the pattern by 2 bits towards Right

Case 3 : S_0 ..... MER ..... S_{n-1}

LEADER
LEADER

here, R & E are matching but $M \neq D$, and also 'M' is not present in pattern string, so shift the entire string as we did in Case 1

Case 4 : S_0 ..... OR ..... S_{n-1}

REORDER
REORDER

here $R = R$, but $O \neq E$ and 'O' is present in the pattern string. leave R, and count the index of 'O' by scanning from ~~left to~~ right-to-left

i.e 'O' index will be 3, so shift the pattern by 3 bits towards right

Algorithm: ShiftTable(P[0...m-1]) // Fills the Shift Table by Horspool's method

Input: Pattern P[0..m-1]

output: Table[0...size-1] indexed by alphabets char & shift size

initialize all elements of Table with m

for j ← 0 to m-2 do Table[P[j]] ← m-1-j

return Table

Horspool's Algorithm

step1: For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table as described above

step2: Align the pattern against the begining of text

step3: Repeat the following until either a match is found or the pattern reaches beyond the last character of the text.

Algorithm: Input: P[0..m-1] & Text T[0..n-1]

Horspool's Alg O/P: index of the left end of the first matching substring or -1 if no match

ShiftTable(P[0...m-1])

i ← m-1 // Position of pattern's right end

while i ≤ n-1 do

K ← 0

while K ≤ m-1 and P[m-1-K] == T[i-K]

K ← K+1

if K = m

return i-m+1

else i ← i + Table[T[i]]

return -1

Character c	A	E	B	...	R	...	Z
shift	4	1	2		3		6

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

JIM_SAW_ME_IN_A_BARBERSHOP → Text string

1) BARBER
   4 3 2 1 0
2) BARBER
3) BARBER

BARBER
BARBER
BARBER

Case 1: R ≠ A, But 'A' is present in pattern, so shift by the index of A. A index = 4 (right to left scan) so shift by 4 bits.

Case 2: R ≠ E, But 'E' is present in pattern, so shift by index of E. E index = 1. shift by 1 bit.

Case 3: R ≠ _ i.e space and '_' is not present in pattern string. so, shift the entire pattern by one bit towards right

Case 4: R ≠ B but 'B' is present in pattern string. index of B = 2, so shift by 2 bits

Case 5: R = R, it moves to next element E ≠ A but again 'A' is present in pattern. index of A = 4 decrement by 1 4 - 1 = 3, because we leave index of R as R = R. so shift by 3.

∴ All characters match, so stop.

3.1.4.3) Hashing

In this section, we consider a efficient way to implement dictionaries. Dictionary is an abstract type namely a set of operations of searching, insertion, deletion defined on its element. Elements of this set can be arbitrary nature : numbers, characters of alphabet etc.

In practice most important case is that of records. Record comprise several fields, each responsible for keeping a particular type of information about the entity the record represents.

Eg: Student record has Student name, USN, Gender, Address, Branch etc.

Among record fields there is usually at least one key.

Hashing is based on the idea of distributing keys among one-dimensional array H[0...m-1] called hash table. Distribution is done by computing for each of the keys, the value of hash function.

For Eg; if key is a +ve integer, a hash function can be of form $h(k) = k \bmod m$

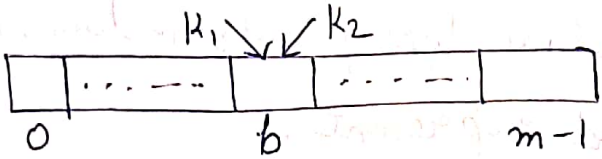
Hash function needs to satisfy two requirements

1) A hash function needs to distribute keys among the cells of hash table as evenly as possible.

2) A hash function has to be easy to compute.

If we choose a hash table's size 'm' to be smaller than no of keys 'n', it leads to collision.

Collision - phenomenon of two or more keys being hashed into the same cells.



Collision of 2 keys in hashing

To resolve such collision, two mechanisms are introduced. 1) Open Hashing (Separate chaining)

2) Closed Hashing (Open Addressing)

Open Hashing

In open hashing, keys are stored in linked lists attached to the cells of hash table. What in hash table? hash table contains the hash value for the elements of given.

Eg: A, FOOL, AND, HIS, MONEY, ARE, SOON, PART

A hash function, we will use the simple function for strings mentioned above. i.e we will add the positions of a word's letter in the alphabet and compute the sum's remainder after dividing by 13.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

1) $h(A) = 1 \bmod 13 = 1$

2) $FOOL = (6+15+15+12) \bmod 13$
$= 48 \bmod 13 = 9$

3) $h(AND) = (1+14+4) \bmod 13$
$= 19 \bmod 13 = 6$

4) $h(HIS) = (8+9+19) \bmod 13$
$= 36 \bmod 13$
$= 10$

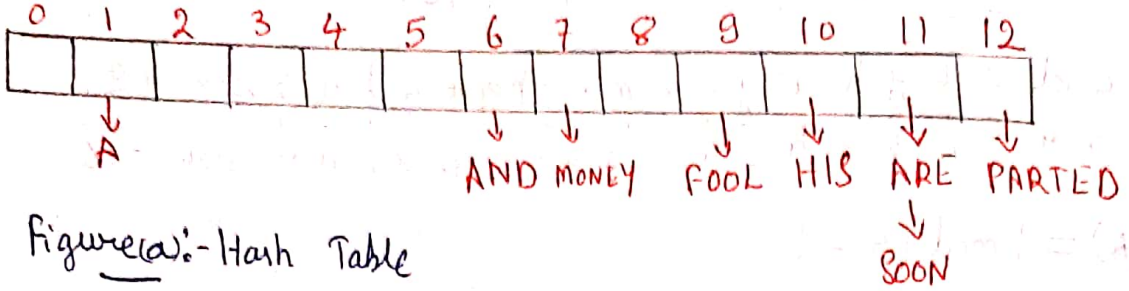
5) $h(MONEY) = (13+15+14+5+25) \bmod 13$
$= 72 \bmod 13$
$= 7$

6) $h(ARE) = (1+18+5) \bmod 13$
$= (24) \bmod 13$
$= 11$

7) $h(SOON) = (19+15+15+14) \bmod 13$
$= 63 \bmod 13$
$= 11$

8) $h(PARTED) = (16+1+18+20+5+4) \bmod 13 = 64 \bmod 13$
$= 12$

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash address	1	9	6	10	7	11	11	12



Figure(a):- Hash Table

How do we search for a key in KID in this hash table

First we compute $h(KID) = (11+9+4) \bmod 13$

$= 24 \bmod 13$

$= 11$

hash value is 11, so we go to cell that has value 11, in cell 11, we have 'ARE' and 'SOON' already mapped so we compare 'KID' with 'ARE' and then with SOON we end up with unsuccessful search.

Efficiency of searching depends on the length of the list, which in turn depend on dictionary & table sizes and quality of hash function.

If the hash function distributes 'n' keys among 'm' cells of the hash ~~function~~ table, each list will be about n/m keys long. The ratio $\alpha = n/m$ calls load factor.

$$S \approx 1 + \frac{\alpha}{2} \quad \text{(Successful Search)}$$

$$U = \alpha \quad \text{(unsuccessful search)}$$

Closed Hashing

The drawback of open hashing is that, as we use linked list we keep on putting the elements to the same cell. but some of the unused cells i.e like 0, 2, 3, 4, 5 in the figure (a) is wasted. To avoid this closed hashing was introduced. In closed hashing we dont use linked list. Here we make use of concept called linear probing. Linear probing means, suppose the hash value is '5' for some function. we look into the cell with index 5, if it is empty we will insert element into it or else we will look to its next cell. If next cell is free we will insert it there.

Example: Given list 200, 111, 23, 83, 105, 56, 55

$h(x) = x \bmod 10$

Calculate the hash values

1) $h(200) = 200 \bmod 10 = 0$
2) $h(111) = 111 \bmod 10 = 1$
3) $h(23) = 23 \bmod 10 = 3$
4) $h(83) = 83 \bmod 10 = 3$
5) $h(105) = 105 \bmod 10 = 5$
6) $h(56) = 56 \bmod 10 = 6$
7) $h(55) = 55 \bmod 10 = 5$

key	200	111	23	83	105	56	55
hash address	0	1	3	3	5	6	5

0	1	2	3	4	5	6	7	8	9
200	111		23	83	105	56	55		

Figure (b):- hash Table

Here we can observe, 23 and 83 has same hash value. 23 is already inserted into cell 3. We check whether the cell next to cell 3 is free or not. if it is empty insert 83 in that empty cell.

i.e how we calculate the index of next cell to be inserted.

$$h'(k) = (h(k)+1) \bmod 10$$

$$= (h(83)+1) \bmod 10$$

$$h'(k) = (3+1) \bmod 10 = 4 \bmod 10 = 4$$

∴ It is inserted at 4.

Next, $h(56) = 56 \bmod 10 = 6$, so we checked cell it was empty so we insert it at 6th cell. after that we take 55.

$$h(55) = 55 \bmod 10$$

$$h(55) = 5$$

But in cell 5, we have already inserted 105, so we check the next cell i.e cell 6. but 6 is also occupied.

	55
	5

g

e hash value.
check whether
if it is empty,

t cell to be

0 = 4

we checked cells 5,
6th cell. after

inserted 105,
ll 6. but 6 is

$h'(55) = (h(55) + 1) \bmod 10$

$h'(55) = 6 \bmod 10 = 6$ but 6 is also occupied, so we calculate $h''(55)$

$h''(55) = (h'(55) + 1) \bmod 10$

$= (6+1) \bmod 10 = 7 \bmod 10 = 7$, so we check if 7 is empty? yes 7th cell is empty so we insert it at 7. This is called as double hashing.

The simplified versions of these results state that, the average number of times the algorithm must access hash table with load factor α in successful and unsuccessful searches are

$$S \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \text{ & } U \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

3.1.4.4) B-Trees

B-Trees extends the idea of 2-3 tree by permitting more than a single key in the same node of a search tree.

In B-Trees, all data records (record keys) are stored at the leaves, in increasing order of the keys. The parental nodes are used for indexing.

A B-Tree of order 'm' must satisfy the following properties. what is 'order' of tree? order of a tree indicates the number of children that each node can have.