5, 3, 21, 9, 1, 13, 2, 7, 10, 12

UNIT-5

Pattern matching & Tries : pattern matching algorithms - the Boyer-Moore algorithm, the Knuth-Morris pratt algorithm

Tries :- Definitions & concept of digital search tree Binary tree, patricia, multi-way tree

Boyer-Moore pattern matching algorithm:

For Searching the pattern in the string we can use the Boyer moore pattern Searching algorithm. The intuition of the B-M algorithm is, Very Simple two pointers are aligned at the 0th index of the Text String & the character string. the pattern string is then compared character by character with the current portion of the Text String, beginning with the rightmost character

Now if a character does not match then boyer moore algorithm shifts the characters using two strategies Simultaneously.

- Bad character heuristic
- Good Suffix heuristic

Bad character heuristic

not satisfied

21

Boyer Moore algorithm matches the pattern ss string with the provided text NOW if a match is found then it returns the patterns starting index otherwise there may arise two cases

1. when the mismatched character of input text is present in the pattern.

In such cases we call that character a bad character so when a bad character is found, we will shift the pattern until it gets aligned with the mismatched character of the text.

Ex: A Y R R Q M G R Q R Q

mismatch

R P C R Q

compare from right to left

we found a mismatch between the character R of the text & character C of the pattern so we will shift the pattern string until character R of the pattern matches the character R of the text string

A Y R R Q M G R Q R Q

R P C R Q

2. when the mismatched character of input text is not present in the pattern.

A Y R R Q M G R P C R Q
R P C R Q
↑ mismatch

there is a mismatch between G & Q, but G is not at all present in entire pattern string so we would not compare the entire pattern & we would easily shift the pattern until the character G on the input text as :

A Y R R Q M G R P C R Q
R P C R Q.

→ shift till after 'G'

Example: 2

G C A A T G C C T A T G T G A C C
↕ mismatch
T A T G T G

here mismatch occur in A from text & G in pattern.

here we check A is in pattern or not here A is in pattern here we apply case 1

G C A A T G C C T A T G T G A C C
T A T G T G
↕

1, 13, 2, 3, 10, 12, 4, 8:

nee again mismatch occurs here c is not in pattern so we apply case

G C A A T G C C T A T G T G A C C
T A T G T G

Approach 2:-

Good suffix heuristic

Another approach for searching the pattern using the boyer-moore algorithm is to detect the good suffix & then do the processing. The main idea is to shift more efficiently when a mismatch occurs by aligning the overlapping parts of the pattern & the text string together

let t be substring of text T which is matched with substring of pattern P. Now we shift pattern until:-

1) Another occurrence of t in P matched with t in T
2) A prefix of P, which matches with suffix of t
3) P moves past t

Case 1 :- Another occurrence of t in P matched with t in T

Pattern P might contain few more occurrences of t in such case we will try to shift pattern to align that occurrence with t in text T.

Ex

A B A A B A B A C B A

C A B A B

t

we have got a substring t of text T matched with pattern P before mismatch at index 2. Now we will search for occurrence starting at position 1 so we will right shift the pattern 2 times to align t in P with t in T

A B A A B A B A C B A

C A B A B

Case 2 :

A prefix of P which matches with suffix of t in T

It is not always likely that we will find the occurrence of t in P. Sometimes there is no occurrence at all, in such cases sometimes

we can search for some suffix of
t matching & with some prefix of
P & try to align them by shifting P

ex:

A A B A B A B A C B A
A B B A B

A A B A B A B A C B A
A B B A B

case 3 : P moves past t

Above two cases are not satisfied then
we will shift the pattern past the
t. for example.

A A C A B A B A C B A
C B A A B

A A C A B A B A C B A
C B A A B

Example of Good suffix heuristic

A A B A B A B A C B A C A B B C A B
A A C C A C C A C
t

In above example got matched
with t in T. The mismatching

character c is 'c' at position p[6]
Now we start searching t in p
we will get the first occurence of
t starting at position 4. but this
occurrence is preceded by 'c' which is
equal to c, so we will skip this &
carry on searching. At position 1
we got another occurrence of t. This
occurrence is preceded by 'A' which
is not equivalent to c. So we will
shift pattern p 6 times to align this
occurrence with t in T

A A B A B A B A C B A C A B B C A B
A A C C A C C A E

Knuth - Morris pratt Algorithm :-(KMP)

KMP matching algorithm uses degenerating property (pattern having the same sub patterns appearing more than once in the pattern) of the pattern & improves the worst case complexity to O(n)

The basic idea behind KMP's algorithm is whenever we detect a mismatch, we already know some of the characters in the text of the

next window we take advantage of this information to avoid matching the characters that we know will anyway match

→ KMP algorithm preprocess pattern and constructs an auxiliary LPS of size m which is used to skip characters while matching

→ LPS indicates longest proper prefix which is also a suffix

→ In LPS table we map every character of the pattern to a value

Creating LPS table (prefix table)

Step 1: Define a one dimensional array with size equal to length of the pattern (LPS[size])

Step 2 :- Define variables i & j. Set i=0, j=1 & LPS[0]=0

Step 3 :- Compare the characters at- pattern[i] & pattern[j].

Step 4 :-

If both are matched then set LPS[j]= i+1 & increment both i & j values by one. Go to step 3

ep 5:- If both are not matched then check the value of variable i. If it is 0 then set LPS[j]=0 & increment j value by one. If it is not 0 then set i = LPS[i-1] Go to Step 3

Step 6 :- Repeat above steps until all the values of LPS[] are filled.

Ex. pattern : | A | B | C | D | A | B | D |

LPS[] size is 7 which is equal to length of pattern.

LPS

Define variables i & j i=0, j=1 & LPS[0] = 0

0	1	2	3	4	5	6
0						

i=0 j=1

compare pattern[i] with pattern[j]
i.e A with B

since both are not matching & i=0, we need to set LPS[j]=0 & increment j value by one

0	0					

i=0 & j=2

A	B	C	D	A	B	D
0	1	2	3	4	5	6

compare pattern[i] with pattern[j]

i.e A with C

since both are not matching & also i=0 we need to set LPS[j]=0 & increment j value by one

LPS | 0 | 0 | 0 | | | |

i=0 & j=3

compare pattern[i] with pattern[j]

i.e A with D

since both are not matching & also i=0 we need to set LPS[j]=0 & increment j value by one

LPS | 0 | 0 | 0 | 0 | | |

i=0 j=4

compare pattern[i] with pattern[j]

i.e A with A

since both are matching set LPS[j] = i+1 & increment both i & j value by one

LPS | 0 | 0 | 0 | 0 | 1 | |

i=1 & j=5

compare pattern[i] with pattern[j]

i.e B with B

since both are matching set LPS[j] = i+1 & increment both i, j

LPS | 0 | 0 | 0 | 0 | 1 | 2 |

i = 2 & j = 6

Compare pattern[i] with pattern[j]
i.e C with D

Since both are not matching & i != 0
we need to set i = LPS[i-1]
i.e i = LPS[2-1] = LPS[1] = 0

LPS [0|0|0|0|1|2]

i = 0 & j = 6

compare pattern[i] with pattern[j]
i.e A with D

Since both are not matching & also i = 0 we need to set LPS[j] = 0 & increment j value by one.

[0|0|0|0|1|2|0]

Here LPS is filled with all values so we stop the process

How to use LPS table :-

When a mismatch occurs, check the LPS value of previous character of the mismatched character in the pattern. If it is 0 then start comparing the first character of the pattern with the next character to the mismatched character in the text. If it is not 0 then

start comparing the character which is at an index value equal to the LPS value of the previous character to the mismatched character in pattern with the mismatched character in text.

Ex ABC ABCDAB ABCDABCD ABDE

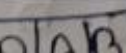
pattern : ABCDABD

LPS | 0 | 0 | 0 | 0 | 1 | 2 | 0 |

| A | B | C | D | A | B | D |

mismatch occur at pattern[6] so we need to consider LPS[2] value since LPS[2] value is 0. we must compare first character in pattern with the next character of mismatch character

| A | B | C | D | A | B | D |

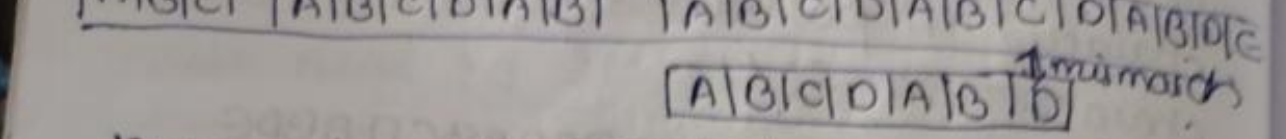


mismatch occur at pattern[6] so we need to consider LPS[5] value since LPS[5] value is 2 we compare pattern[2] value with mismatched character

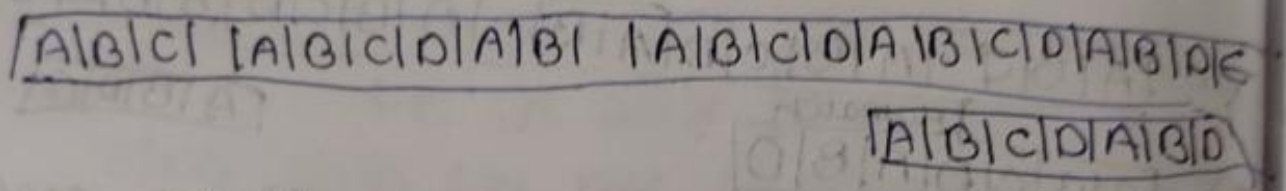
| A | B | C | D | A | B | D |

here mismatch occured at pattern[2], so we need to consider LPS[1] value

Since LPS[1] value is '0' we must compare pattern[2] first character of pattern with next character of mismatched character.



here mismatch occured at pattern[6]
So we need to consider LPS[5] value.
Sina LPS[5] value is 2 we compare pattern[2] character with mismatched character in text



Example: 2

T: b a c b a b a b a b a c a c a

P: a b a b a c a

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

b a c b a b a b a b a c a c a
a b a b a c a

b a c b a b a b a b a c a c a
a b a b a c a
mismatch

b a c b a b a b a b a c a c a
a b a b a c a

b a c b a b a b a b a c a c a
a b a b a c a
mismatch

b a c b a b a b a b a c a c a
a b a b a c a

Tries

Digital Search Tree :-

Trie is a Sorted tree based data structure that stores the set of strings. It has the no. of pointers equal to the no. of characters of the alphabet in each node.

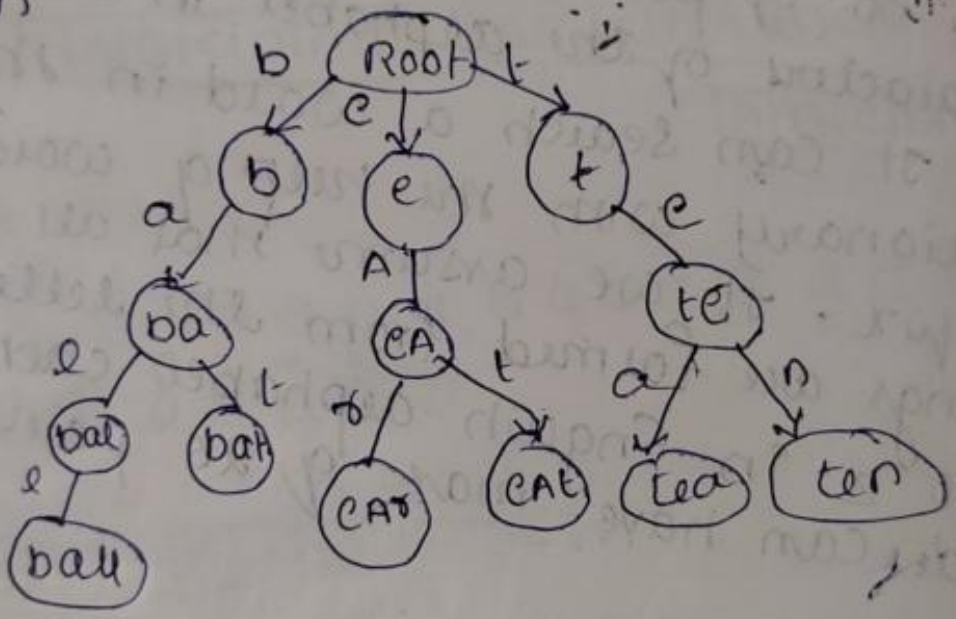
It can search a word in the dictionary with the help of word's prefix. If we assume that all the strings are formed from the letters a to z in English alphabet each trie node can have max of 26 pointers.

properties :-

→ The structure of a trie is like that of a tree
→ Each trie consists of a root node
→ The root node branches in to various child nodes having multiple Edges
→ Each trie node consists of an array of pointers where every index

of the array represents a character.
→ Each node of a trie represents a string and each edge represents a character
→ The root node is an empty string.
→ Each node except the root node is a string having characters along the path from the root to that node.

Ex.. Trie datastructure with input strings as ball, bat, CAR, CAt, tea & ten



Digital Seach tree :- (DST)

A digital Seach Tree is a binary tree conose ordering of nodes is based on the values of bits in the binary representation of a nodes key.

The ordering principle of is very simple at each level of a tree a

different bit of the key is checked if the bit is 0 the search element continues down to the right subtree

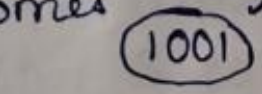
Every node in DST holds a key & links to the left & right child ; unlike ordinary binary search tree.

Creating & Searching DST :-

Let create a DST with elements 1001, 0110, 0000, 1111, 0100, 0101, 1110

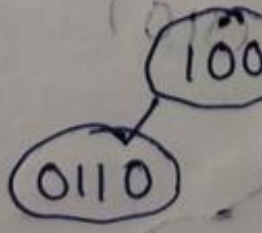
Step 1 : Insert 1001

Since we start with an Empty Tree 1001 becomes root.



Step 2 : Insert 0110

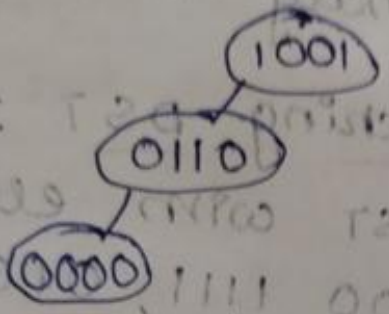
Since tree is not empty. We look at the first bit from the left Since this bit is 0 we take the path down to left subtree & find left pointer is NIL since we create a node.



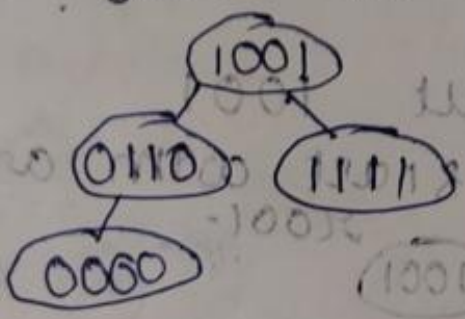
Step 3 : Insert 0000

Since root is not nil we look at first bit since bit is 0

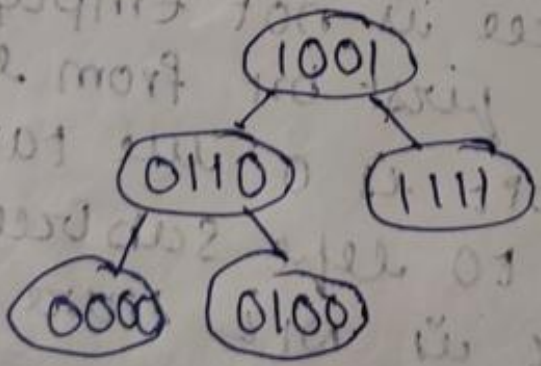
we take the path down to left subtree since the next node is not NIL since we look at the next bit i.e 0 so we take the path down to left subtree & it is empty we place the node



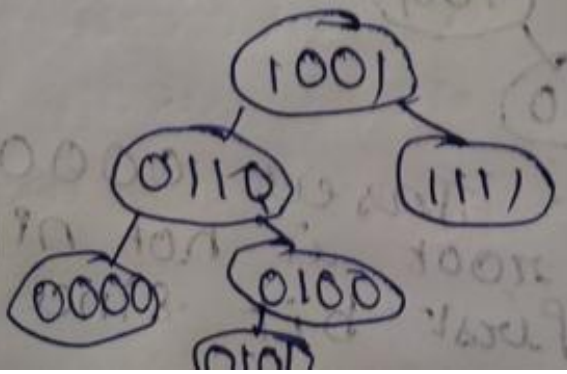
Step 4 : Insert 1111



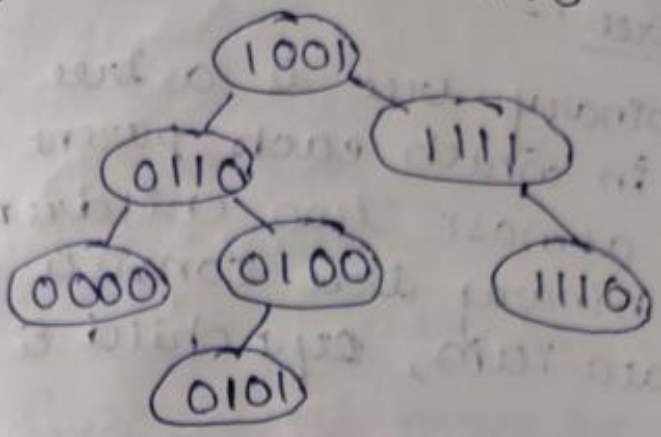
Step 5 : Insert 0100



Step 6 : Insert 0101



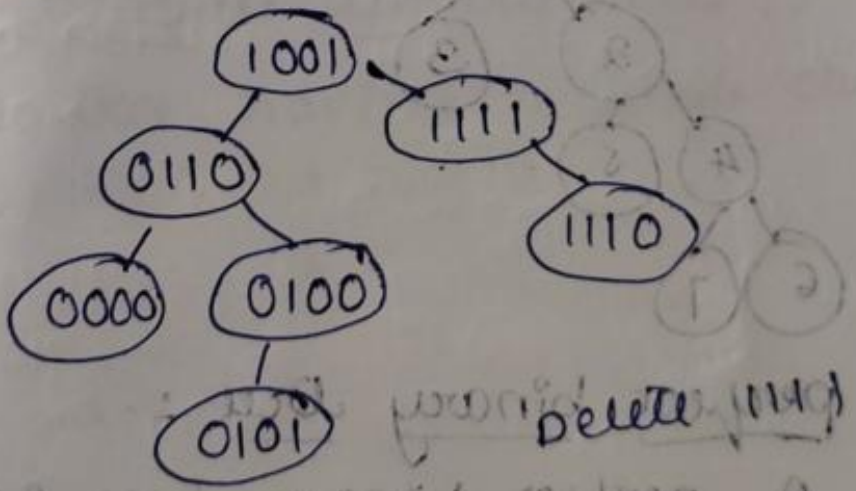
step 7 : Insert 1110

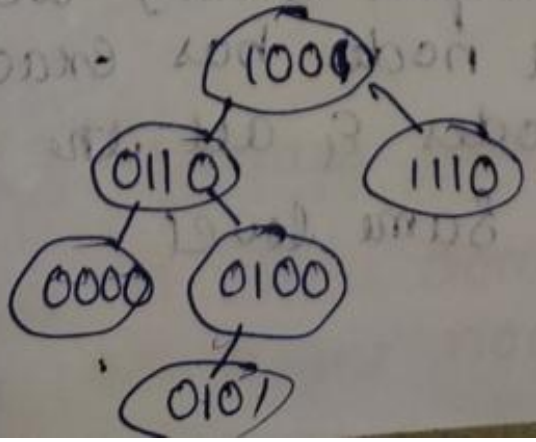


Deleting Nodes from DST's

Deletion is same as BST. If it is a leaf node we will directly delete the node.

→ If it is non leaf node we will replace it inorder successor or predecessor & will delete it.





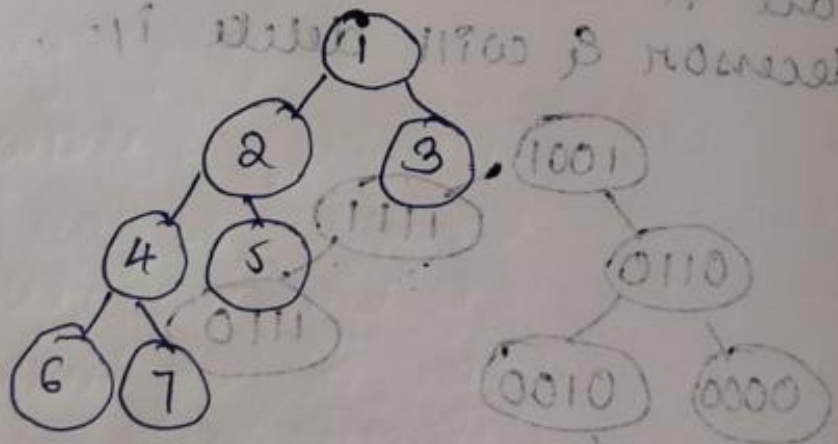
Binary tree :-

A binary tree is a Tree data structure in which each parent node can have atmost two children. Each node of binary tree consists of three items; data item, left child & right child

Types of binary tree :-

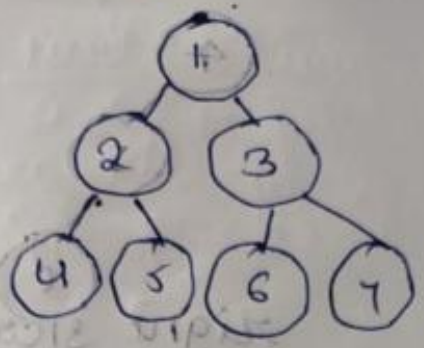
1. Full binary tree :-

It is special type of binary tree in which every parent has either two or no children.



2. perfect binary tree :-

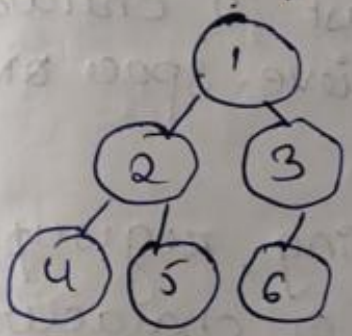
In A perfect binary Tree every internal nodes has exactly two child nodes & all the leaf nodes are at same level.



3. Complete binary tree :-

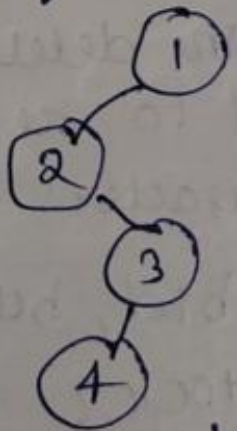
↳ Every level must be completely filled

↳ All the leaf elements must lean towards left



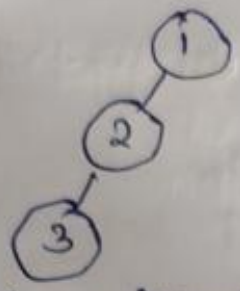
4. Degenerate binary tree :-

A tree having a single child either left or right

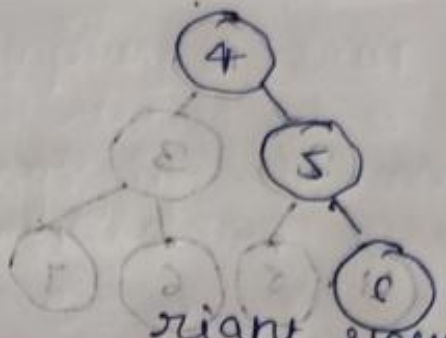


5. Skewed binary tree :-

The tree is either dominated by left nodes or right nodes.



left skewed



right skewed.

Patricia :-

It is also known as Compressed Trie

↳ A Compressed Trie is an advanced version of Standard Trie.

↳ Each node has atleast 2 children

↳ It is used to achieve space optimization.

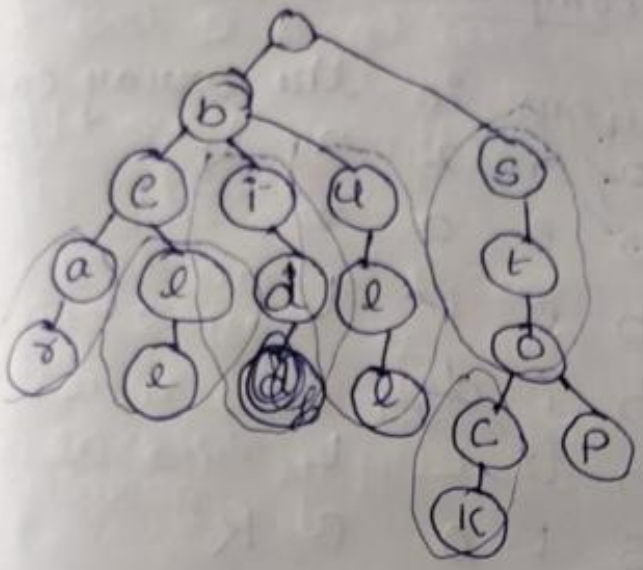
↳ It consists of grouping, regrouping & ungrouping of keys of characters

↳ while performing insertion operation it may be required to ungroup the already grouped character

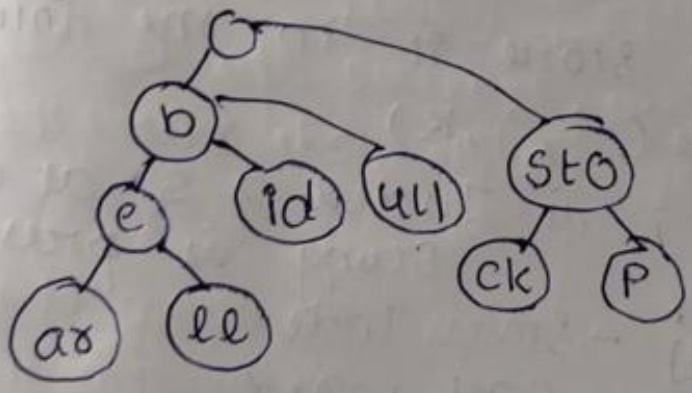
↳ while performing the deletion operation it may be required to regroup the already grouped character

Ex: { bear, bell, bid, bull, stop, stock }

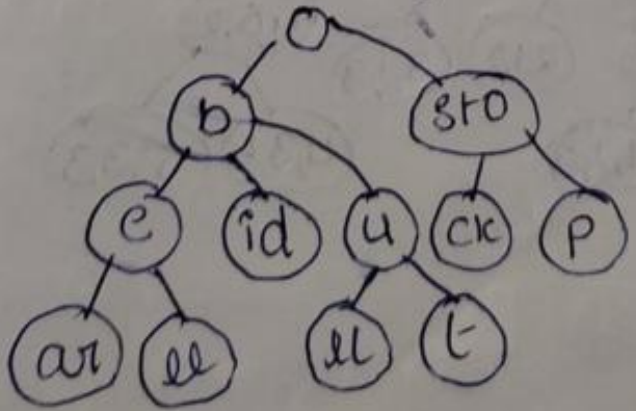
Standard Trie



Compressed Trie



Inserting "bat" requires ungrouping of keys as follows



Representing compressed tries

⇒ storing strings in the array called 's' or

	0	1	2	3	4
S[0] :	b	e	a	r	
S[1] :	b	e	l	l	
S[2] :	b	i	d		
S[3] :	b	u	l	l	
S[4] :	s	t	o	c	k
S[5] :	s	t	o	p	

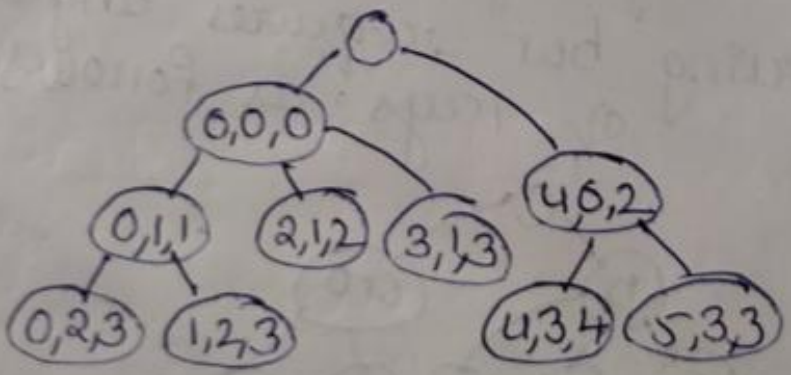
we can store it in the form of

Node (i, j, k)

where i is Index of S at which string is present

j - Start Index

k - End Index



Multi way trees :- (M- way trees)

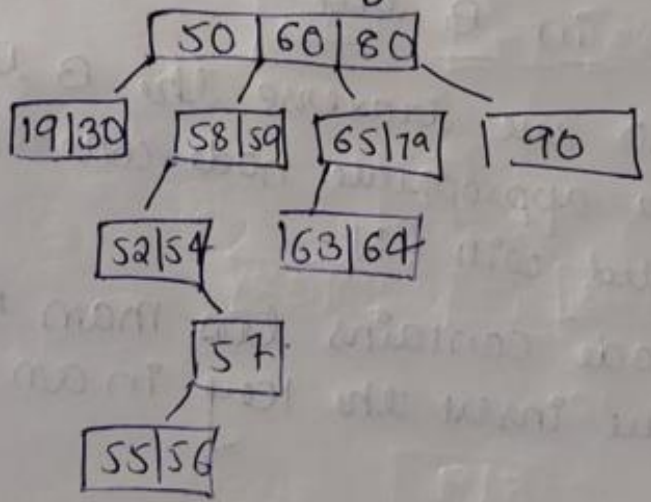
→ Each node in the tree can have at most m children

→ Nodes in the tree have at most (m-1) key fields & m pointers (children)

Ex: 4 way tree

(m) children → can have 4 child

(m-1) keys → can have 3 keys



→ The keys in any node of the tree are arranged in sorted order.

→ B trees are extension of an M- way search tree

B- trees :-

A B tree is an extension of M- way search tree Besides having all of the properties M way search tree &

the properties of its own these mainly are
→ All the leaf nodes in a B tree are at the same level
→ All internal nodes must have M/2 children
→ If the root node is a non leaf node, then it must have at least 2 children
→ All nodes except root node must have atleast [M/2]-1 keys and at most M-1 keys

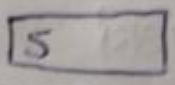
Insertion in B tree

1) First we traverse the B tree to find the appropriate node where the element be inserted will fit.
2) If node contains less than M-1 keys then we insert the key in an increasing order
3) If the node contains exactly M-1 keys then we have two cases insert the new element in increasing order split the nodes in to two nodes through the median push the median element up to its parent node & finally if the parent node also contains M-1 keys then we need to repeat these steps

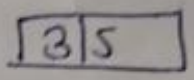
Ex: construct a B tree of order 4
5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8

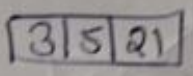
m = 4
keys: M-1 = 3
pointers = 4

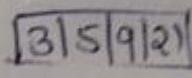
Insert 5



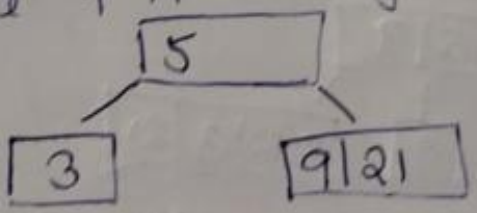
Insert 3



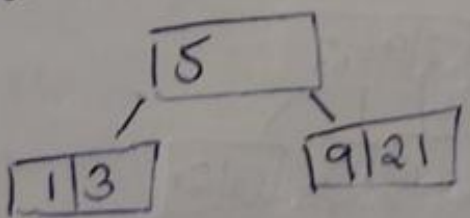
Insert 21 

Insert 9 

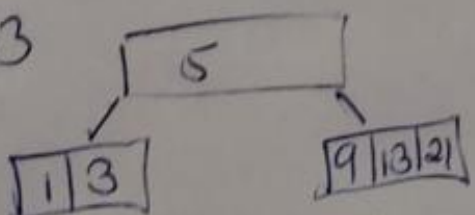
properly violated median is promoted as parent here both 5 & 9 are median. If we promote 5 it is called as left biased if we raising promote 9 it is right biased



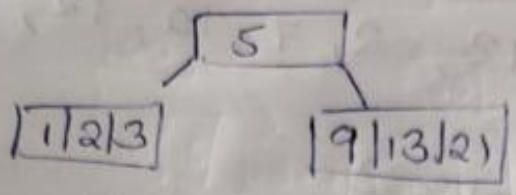
Insert 1



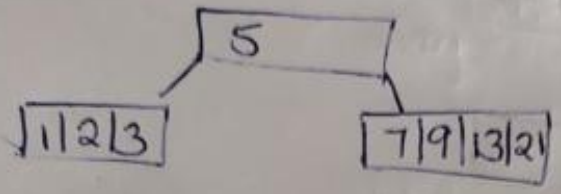
Insert 13



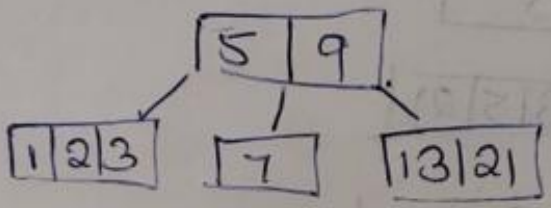
Insert 2



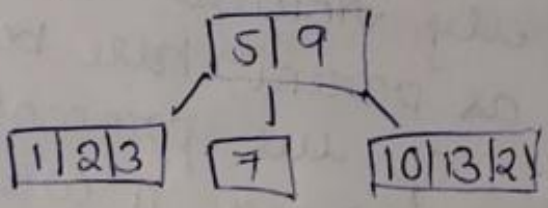
Insert 7



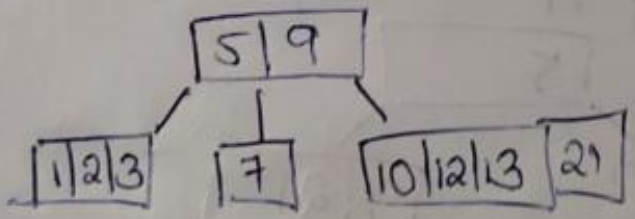
property violated 9 is promoted



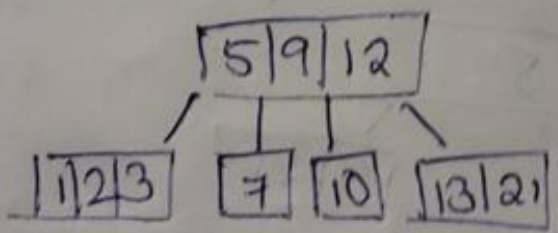
Insert 10



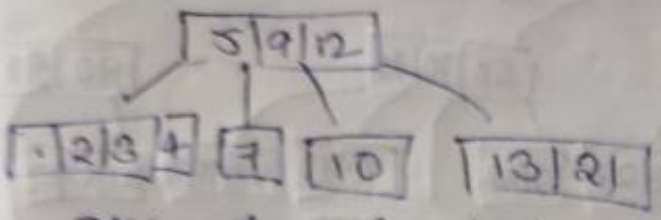
Insert 12



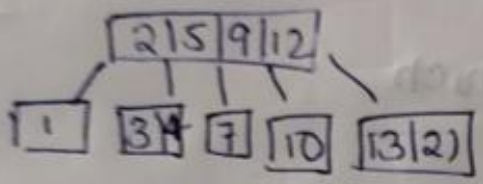
property violated promote 12



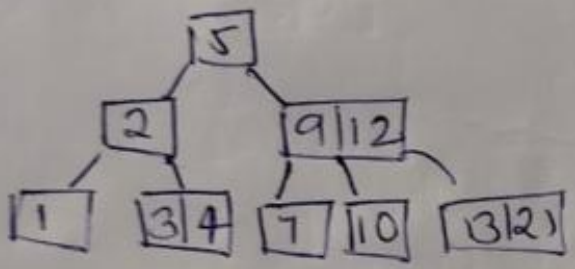
Insert 4



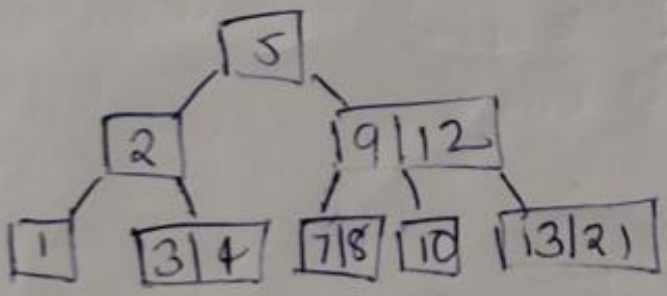
property violated 2 is promoted



property violated promote 5



Insert 8



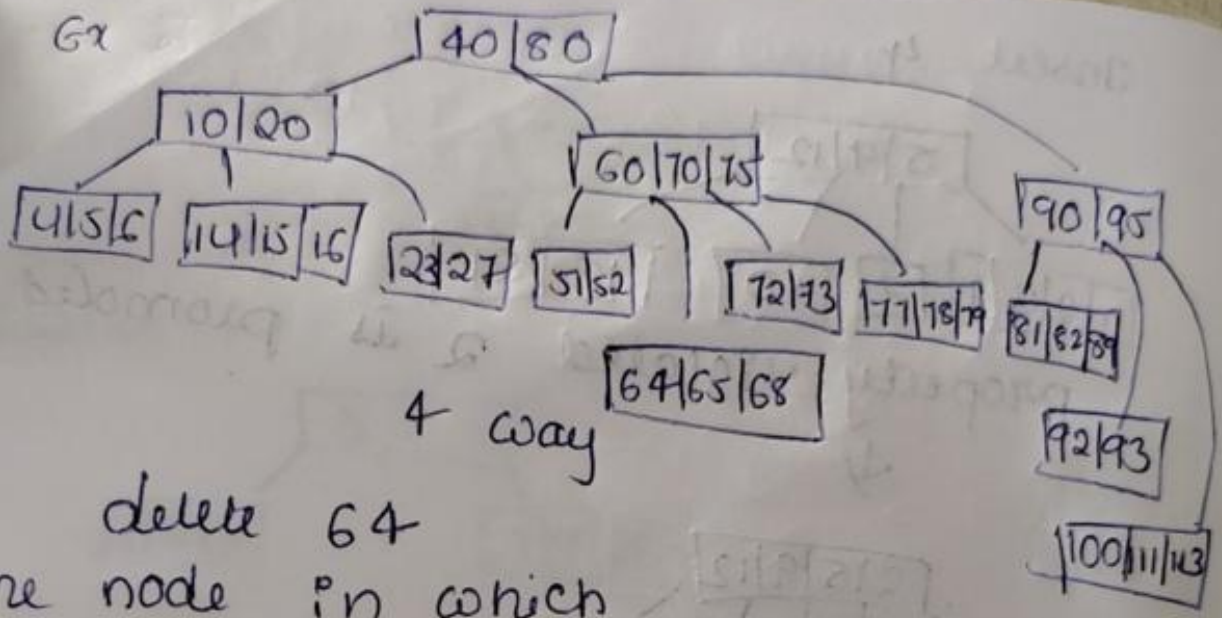
Deletion in a B Tree :-

↳ Deletion of a key from leaf node
↳ Deletion of a key from internal node

If we want to delete a key in leaf node we have 2 cases

case

Ex



4 way

delete 64

The node in which