2512

Code No.: R25CS58102PC | R25 | H.T.No. | | | 8 | R | | | | | | |

CMR ENGINEERING COLLEGE: : HYDERABAD
UGC AUTONOMOUS
I–M.TECH–I–Semester End Examinations (Regular) - February- 2026
ADVANCED DATA STRUCTURES
(CSE)

[Time: 3 Hours] **[Max. Marks: 60]**

Note: This question paper contains two parts A and B.
Part A is compulsory which carries 10 marks. Answer all questions in Part A.
Part B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b, c as sub questions.

PART-A **(10 Marks)**

1. a) Define a heap data structure with example. [2M]
   b) What is a hash function? [2M]
   c) Define Balance Factor in an AVL tree? [2M]
   d) Difference between Trie and BST. [2M]
   e) What is Boyer–Moore algorithm? [2M]

PART-B **(50 Marks)**

2. Explain Heap structure & construct a Min Heap by inserting the elements 10, 4, 15, 20, 0, 8. Show all intermediate steps. [10M]

OR

3. Explain Leftist Heaps with insertion and deletion algorithms and diagrams. [10M]

4. Construct a hash table using Division Method (h(k) = k mod 10) for keys 23, 43, 13, 27, 53. [10M]

OR

5. Compare Open Addressing and Chaining techniques in hash tables with examples. [10M]

6. Insert the following keys into a Red-Black Tree: 7, 3, 18, 10, 22, 8, 11. [10M]

OR

7. Compare AVL Trees and Red-Black Trees with suitable examples. [10M]

8. Explain Tries and its types and construct a Tries for the words: cat, cap, can. [10M]

OR

9. Compare Standard Tries and Compressed Tries with representations. [10M]

10. Explain Naïve String Matching and apply it to find pattern "aba" in "ababa". [10M]

OR

11. Analyze best, worst, and average-case complexities of all pattern matching algorithms. [10M]

****

I. M.Tech I-Semester End Examination- Feb-26 (R25)
R25CS58102PC-ADVANCED DATA STRUCTURES

PART-A

1)
a) Define a heap data structure with example?

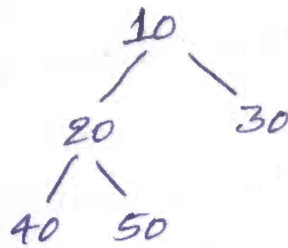
Ans:- Heap Data structure:-

A Heap is a complete binary tree that satisfies the heap property.

→Min Heap:- Parent node ≤ child nodes.

→Max Heap:- Parent node ≥ child nodes.

Example (Min Heap):-



b) What is a hash function?

Ans:- Hash Function:-

A Hash Function is a function that converts a key into an index (hash value) in a hash table.

It is used to store and retrieve data efficiently.

Example:-

If table size = 10

h(key) = key mod 10

h(25) = 25 mod 10 = 5

c) Define Balance Factor in an AVL tree?

Ans:- Balance Factor in AVL Tree:-

Ans:- In an AVL tree, the Balance Factor (BF) of node is:-

BF = Height of Left subtree - Height of Right subtree.

For AVL trees, BF must be -1, 0, or +1.

d) Difference between Trie and BST?

Trie	BST
Used to store strings	Used to store numbers or keys
stores characters level by level	stores keys using left < root < right rule.
search time depends on word length	search time depends on tree height.
Example:- Dictionary	Example:- Binary search Tree.

e) What is Boyer-Moore algorithm?

Ans:- The Boyer-Moore string-search algorithm is a string searching algorithm.

It searches for a pattern in text by comparing characters from right to left and skips sections using heuristics.

It is more efficient than naive string matching in

PART-B

2) Explain Heap structure and construct a Min Heap by inserting the elements 10, 4, 15, 20, 0, 8. show all intermediate steps.

Ans:- 1. Heap structure:-

A Heap is a complete binary tree that satisfies the heap property.

In a Min Heap:-

→ Every parent node is less than or equal to its children.

→ The minimum element is always at the root.

→ It is usually represented using an array.

Properties:-

1. complete Binary Tree (filled left to right)

2. Follows Min-Heap order property.

3. Insertion uses heapify-up (bubble-up).

2. construction of Min Heap:-

Insert elements one by one:-

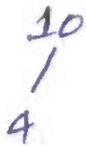
Elements:- 10, 4, 15, 20, 0, 8

step 1:- Insert 10

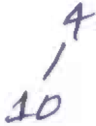
10

step 2:- Insert 4

Insert at next position

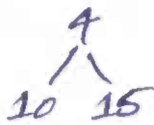


Since 4 < 10 → swap



Array: [4, 10]

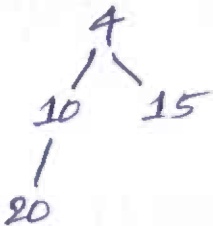
Step 3:- Insert 15



15 > 4 → no swap

Array: [4, 10, 15]

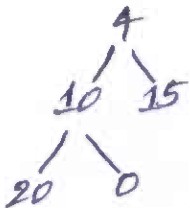
Step 4:- Insert 20



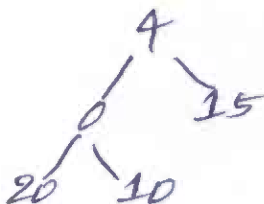
20 > 10 → no swap

Array: [4, 10, 15, 20]

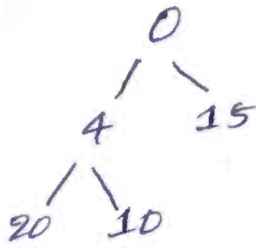
Step 5:- Insert 10



0 < 10 → swap

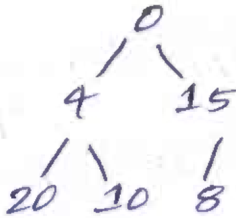


0 < 4 → swap again

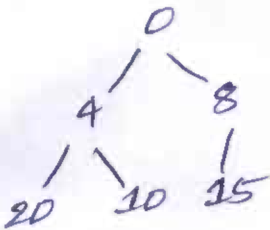


Array: [0, 4, 15, 20, 10]

step 6:- Insert 8

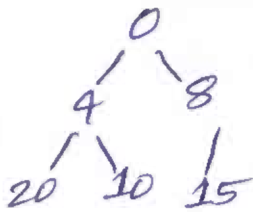


8 < 15 → swap



8 > 0 → stop

Final Min Heap:-



Final Array Representation:-
[0, 4, 8, 20, 10, 15]

3) Explain Leftist Heaps with insertion and deletion algorithms and diagrams. Leftist Heaps?

Ans:- 1. Definition:-

A Leftist tree (Leftist Heap) is a special type of Min Heap used mainly to implement Priority

queues efficiently.

It satisfies:-

1. Heap Property:-

Parent node ≤ children nodes.

2. Leftist Property:-

For every node:-

NPL (Left) ≥ NPL (Right) NPL (Left) \ge NPL (Right) NPL (Left) ≥ NPL (Right)

Where NPL (Null path Length) = Length of shortest path from a node to a null child.

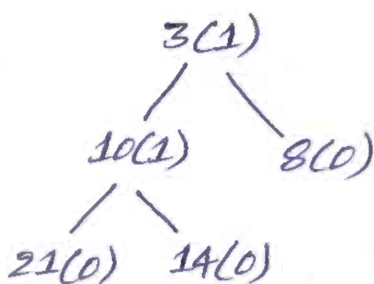
2. Null path Length (NPL)

→ NPL (NULL) = -1

→ NPL (Leaf Node) = 0

→ NPL (Node) = 1 + min (NPL (left), NPL (right))

3. Structure Diagram Example:-



(value (NPL))

Here:-

→ Heap Property is satisfied

→ Leftist Property is satisfied (Left NPL ≥ Right NPL).

4. Insertion in Leftist Heap:-

Idea:-

Insertion is done by merging:-

→ create a new single-node heap.

→ Merge it with existing heap.

Merge Algorithm (Main operation)

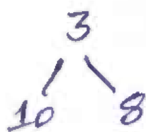
Merge(H1, H2):-

1. If H1 is NULL → return H2
2. If H2 is NULL → return H1
3. If H1. root > H2. root → swap H1 and H2
4. Recursively merge H1. right and H2
5. If NPL(left) < NPL(right) → swap children.
6. Update NPL
7. Return H1.

Time complexity: O(logn)

Example:- Insert 5 into Heap

step 1:- original Heap



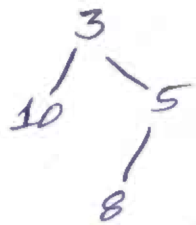
step 2:- create New Node (5)

5

step 3:- Merge

compare 3 and 5 → 3 is smaller → merge right subtree of 3 with 5

After merging:-



check Leftist Property:-

If violated → swap children.

Final heap satisfies both properties.

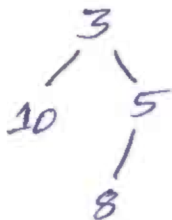
5. Deletion in Leftist Heap:-

Delete Min operation:-

1. Remove the root (Minimum element).
2. Merge left subtree and right subtree.
3. Resulting heap is new heap.

Delete Example:-

original Heap:-



step 1 : Remove Root (3)

Now merge:-

step 2:- Merge 10 and 5

5 < 10 → 5 becomes root

After merging:-

Heap property satisfied.

Time complexity :- O(log n)

6. Advantages of Leftist Heap:-

→ Efficient merging (better than binary heap)

→ Insert and delete in O(logn)

→ Used in Priority queues.

Conclusion:-

A Leftist Heap is a Min Heap that maintains the Leftist property using NPL, making merge, insert, and delete operations efficient (O(logn))

---

4) Construct a hash table using Division Method (h(k) = k mod 10) for keys 23, 43, 13, 27, 53.

Construction of Hash Table using Division Method.

Ans:- Definitions of Terms:-

1. Hash Table:-

A Hash Table is a data structure used to store keys using a hash function for fast insertion and searching.

2. Hash Function:-

A Hash Function converts a given key into an index (address) in the hash table.

Here,

$h(k) = k \bmod 10$

3. Division Method:-

In the Division Method, the hash value is obtained by dividing the key by the table size and taking the

Formula:-

$h(k) = k \bmod m$

where m = table size(10)

4. collision:-

When two or more keys to the same index, it is called a collision.

Given Data:-

Hash Function:-

$h(k) = k \bmod 10$

keys: 23, 43, 13, 27, 53

Table size = 10 (0 to 9)

step 1:- calculate Hash values:-

key	h(k) = k mod 10	Index.
23	23 mod 10 = 3	3
43	43 mod 10 = 3	3
13	13 mod 10 = 3	3
27	27 mod 10 = 7	7
53	53 mod 10 = 3	3

step-2:- construct Hash Table.

since multiple keys map to index 3, collision occurs. we use separate chaining to handle collisions.

Final Hash Table:-

Index	Elements
0	-
1	-
2	-
3	23 → 43 → 13 → 53
4	-
5	-
6	-
7	27

Using the Division Method $(h(k) = k \mod (10))$:-

→ Index 3 → 23 → 43 → 13 → 53

→ Index 7 → 27

→ All other indices are empty.

Collision is handled using chaining method.

5. Compare open Addressing and chaining techniques in hash tables with examples.

Ans:- Introduction:-

When a collision occurs in a Hash Table, it must be resolved using a collision resolution technique.

Two common techniques are:-

1. open Addressing
2. chaining (Separate chaining)

1. open Addressing:-

In open Addressing, all elements are stored inside the hash table itself.

If collision occurs, we search for another empty slot using a probing method.

Types of probing:-

→ Linear probing

→ Quadratic probing

→ Double Hashing.

Example:-

Hash Function:-

$h(k) = k \bmod 10$

keys: 23, 43

→ $h(23) = 3$ → Insert at index 3

→ $h(43) = 3$ → collision

Using Linear Probing.

Next free slot = 4

Final Table:-

Index	Element
3	23
4	43

2. chaining (separate chaining)

In chaining, each index stores a linked list of elements. If collision occurs, elements are stored in the list at that index.

Example:-

Hash Function:-

$h(k) = k \bmod 10$

keys: 23, 43

→ $h(23) = 3$

→ $h(43) = 3$ (collision)

Using chaining

Index	Element
3	23 → 43

comparison Table:-

Feature	open Addressing	chaining
storage	Inside table only	uses linked list
collision Handling	Probing	Linked list
Extra Memory	No extra memory	Requires extra memory.
Performance	slower when table is full	Better when many collisions.
Deletion	Difficult	Easy.

Advantages:-

open Addressing:-

→ No extra memory required

→ simple implementation.

chaining:-

→ Easy deletion

→ Handles high load factor better.

conclusion:-

→ open Addressing stores all keys inside the table and uses Probing.

→ chaining stores collided keys in linked lists.

Both techniques are used to resolve collisions in hash tables.

6. Insert the following keys into a Red-Black Tree:- 7, 3, 18, 10, 22, 8, 11. Insertion in Red-Black Tree.

Ans:- Definition:-

A Red-Black tree is a self-balancing Binary search Tree with these Properties.

1. Every node is either Red or Black.

2. Root is always Black.

3. No two consecutive Red nodes.

4. Every path from root to NULL has the same number of Black nodes (Black Height).

Insert keys:-

keys:- 7, 3, 18, 10, 22, 8, 11.

(New node is always inserted as Red)

Step 1:- Insert 7

7 becomes root and is colored Black.

7(B)

Step 2:- Insert 3.

3<7 → Insert left of (Red).

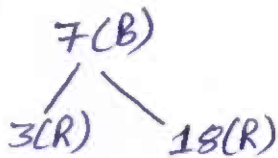
Parent is Black → No Violation.

```
    7(B)
   /
 3(R)
```

Step 3:- Insert 18

18 > 7 → Insert right of 7(Red).

Parent is Black → No Violation



step 4:- Insert 10

10 > 7 → go right

10 < 18 → insert left of 18 (Red)

Now:-

→ Parent (18) is Red

→ uncle (3) is Red

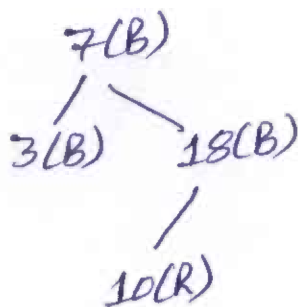
case 1:- Recoloring.

18 → Black

3 → Black

7 → Red

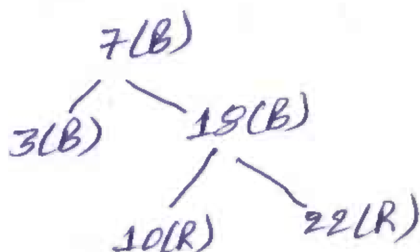
Root must be Black → 7 becomes Black.



step-5:- Insert 22

22 > 18 → insert right of 18(Red).

Parent is Black → No violation.



step-6:- Insert 8

$8 > 7 \rightarrow$ right

$8 < 18 \rightarrow$ left

$8 < 10 \rightarrow$ left of 10(Red)

Now:-

$\rightarrow$ Parent (10) is Red

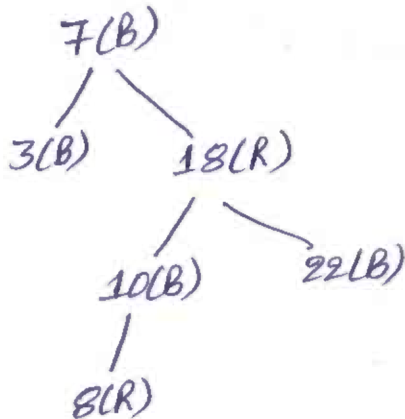
$\rightarrow$ uncle (22) is Red.

case 1:- Recoloring:-

10 $\rightarrow$ Black

22 $\rightarrow$ Black

18 $\rightarrow$ Red.



step 7:- Insert 11

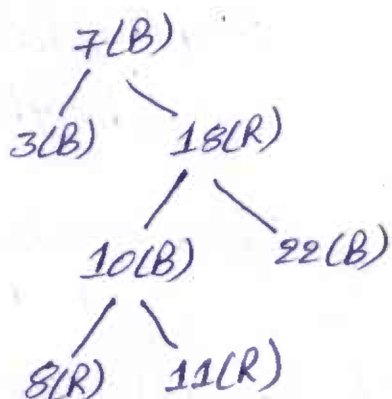
$11 > 7 \rightarrow$ right

$11 < 18 \rightarrow$ left

$11 > 10 \rightarrow$ right of 10(Red)

Parent (10) is Black $\rightarrow$ No violation.

Final Red-Black Tree



Final Answer:-
After inserting 7, 3, 18, 10, 22, 8, 11, the final Red-Black Tree is:-
→ Root = 7 (Black)
→ Balanced according to Red-Black Properties.
→ No two consecutive Red nodes.
→ Equal Black height maintained.

7. Compare AVL Trees and Red-Black Trees with suitable examples. comparison between AVL Tree and Red-Black Tree.

Ans:-

Feature	AVL Tree	Red-Black Tree
Type	self-balancing Binary search Tree	self-balancing Binary search Tree
Balance condition	Balance Factor = -1, 0, +1	Follows Red-Black color Properties.
strictness of Balance	strictly balanced	Less strictly balanced.

Feature	AVL Tree	Red-Black Tree
Height	smaller height	slightly larger height.
Rotations	More rotations Required	Fewer rotations Required
search operation	Faster (due to strict balance)	slightly slower.
Insertion	More complex	Easier compared to AVL
Deletion	More complex	Easier compared to AVL
Rebalancing Method	single & Double rotations	Recoloring + Rotations
Time complexity	o(logn)	o(logn)
Applications	suitable for search-intensive applications	used in maps, sets, system libraries.

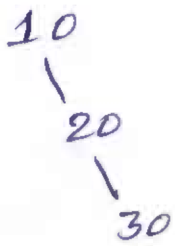
1. AVL Tree Example:-

consider inserting: 10, 20, 30 into an AVL tree

After inserting 10 and 20, tree is balanced.

After inserting 30, tree becomes unbalanced (Right - Right case).

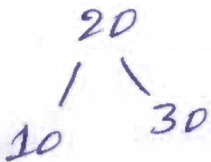
Unbalanced Tree:-



Balance Factor of $10 = -2$ → not allowed.

→ Perform Left Rotation

Final AVL Tree



→ Balanced

→ Height difference ≤ 1.

2. Red-Black Tree Example

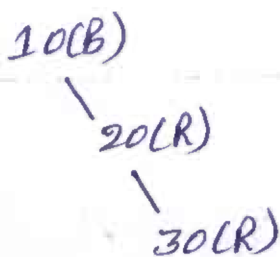
Insert: 10, 20, 30 into a Red-Black tree

Step 1:- 10 → Root (Black)

Step 2:- 20 → Red

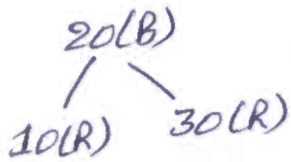
Step 3:- 30 Red (Violation: two consecutive Reds)

Before Fixing:-



Perform Rotation + Recoloring.

Final Red-Black Tree:-



→ No two consective Red nodes.

→ Black height maintained.

8. Explain Tries and its types and construct a Tries for the words: cat, cap, can.

Ans:- Definition:-

A Trie (also called Prefix Tree) is a tree-based data structure used to store strings efficiently. It is mainly used for dictionary searching, autocomplete, and spell checking.

In a Trie:-

→ Each node represents a character.

→ Words are formed from root to leaf.

→ common prefixes are stored only once.

Properties of Trie:-

1. Root node is empty.
2. Each edge represents a character.
3. Nodes share common prefixes.
4. End of word is marked specially (✓ or *).

Types of Tries:-

1. standard Trie:-

→ stores characters one by one.

→ Each node has multiple children (based on alphabet).

2. compressed Trie:-

→ combines single-child nodes into one edge.

→ saves memory.

3. suffix Trie:-

→ stores all suffixes of a string.

→ used in pattern matching.

construction of Trie:-

Words:- cat, cap, can

step 1:- Insert "cat"

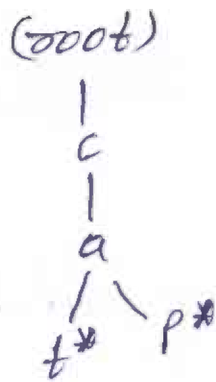
```
(root)
  |
  c
  |
  a
  |
  t*
```

(* = end of word)

step 2:- Insert "cap"

→ c and a already exist.

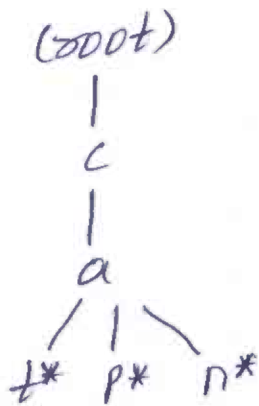
→ Add p branch.



Step 3:- Insert "can"

→ c and a already exist.

→ Add n branch.



Final Trie structure:-

→ All words share common prefix "ca".

→ Branches split at last character.

→ Efficient storage of similar words.

conclusion:-

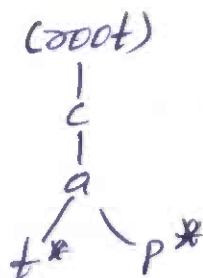
A Trie is useful for fast searching of strings.
Time complexity for search and insertion = O(L)
(where L = Length of words).

9. compare standard Tries and compressed Tries with representations.

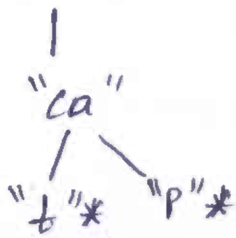
Feature	Trie(standard Trie	compressed Trie.
Node storage	stores one character per node	stores substring (group of characters) per node.
Memory Usage	uses more memory	uses less memory
Height of Tree	More(longer paths)	Less(shorter paths)
space Efficiency	Low(many single-child nodes)	High(merges single-child nodes).
Implementation	simple	slightly complex.
Traversal speed	slower(more nodes to visit)	Faster (fewer nodes)
Example Representation (cat, cap)	root → c → a → t/p	root → "ca" → "t"/"p"

<u>Example</u> <u>Representation</u>:-

→standard Trie:-



→ compressed Trie.



conclusion:-

→ standard Trie stores characters separately.

→ compressed Trie merges nodes to reduce height and save memory.

---

10. Explain Naive string Matching and apply it to find pattern. "aba" in "ababa".

Ans:- Naive string Matching Algorithm:-

Definition:-

The Naive string Matching Algorithm is the simplest Pattern searching method.

It checks the pattern at every possible position in the text.

If all characters match → pattern found.

If mismatch occurs → shift Pattern by one Position and continue.

Algorithm steps:-

1. start comparing pattern with text from leftmost position.

2. compare characters one by one.

3. If mismatch → shift pattern by 1 position.

4. Repeat until end of text.

Time complexity:-

→ Worst case:- $O(nm)$
(n = length of text, m = length of pattern).

Apply to Example:-

Text(T) = "ababa"

Pattern(P) = "aba"

Length of Text (n) = 5

Length of pattern (m) = 3

Possible shifts = $n-m+1 = 5-3+1 = 3$

step 1:- shift 0

compare:-

T: a b a b a

P: a b a

✓ a = a

✓ b = b

✓ a = a

⇒ match found at position 0.

step 2:- shift 1

T: a b a b a

P: a b a

× a ≠ b (Mismatch at first character)

shift again.

step 3:- shift 2

T:- a b a b a

P: a b a

✓ a = a

✓ b = b

✓ a = a

⇒ Match found at Position 2

Final Answer:-

Pattern "aba" occurs in "ababa" at positions:

- 0
- 2

conclusion:-

Naive string matching compares the pattern at every shift. for the given example, the pattern appears twice in the text.

11. Analyze best, worst, and average-case complexities of all pattern matching algorithms.

Ans:- Time complexity Analysis of Pattern Matching Algorithms:-

Let

⇒ n = Length of Text

⇒ m = Length of Pattern.

1. Naive string matching:-

The Naive Algorithm compares pattern at every shift.

case	Time complexity.
Best case	$O(n)$
Average case	$O(nm)$
Worst case	$O(nm)$

Explanation:-

→ Best case:- Mismatch at first character each time.

→ Worst case:- pattern like "aaa" in "aaaaaa" (many comparisons).

2. Knuth – Morris – Pratt algorithm (KMP):-

Uses LPS (Longest prefix suffix) table.

case	Time complexity
Best case	o(n)
Average case	o(n)
Worst case	o(n)

Explanation:-

→Never rechecks characters.

→Efficient for repeated patterns.

3. Rabin-karp algorithm:-

uses hashing technique.

case	Time complexity
Best case	o(n+m)
Average case	o(n+m)
Worst case	o(nm)

Explanation:-

→Worst case occurs when many hash collisions happen.

4. Boyer-Moore string-search Algorithm:-

compares from right to left and skips characters.

case	Time complexity
Best case	o(n/m)
Average case	o(n)
Worst case	o(nm)

Explanation:-

→ very fast in practice.

→ Worst case occurs in repetitive text.

summary Table:-

Algorithm	Best case	Average case	Worst case
Naive	o(n)	o(nm)	o(nm)
KMP	o(n)	o(n)	o(n)
Rabin-karp	o(n+m)	o(n+m)	o(nm)
Boyer-Moore	o(n/m)	o(n)	o(nm)

<u>conclusion:-</u>

KMP:- gives guaranteed linear time o(n)

Rabin-karp :- Is efficient with good hash Function

Boyer-Moore :- Is fastest in Practice

Naive :- Is simple but inefficient in Worst case.