Advanced Algorithum Practicals

Madhav Agarwal -- 20221426

Question 1 : Write a program to sort the elements of an array using Randomized Quick Sort (the program should report the number of comparisons)

Solution :

This program sorts the elements of an array using Randomized Quick Sort. Randomized Quick Sort is a variation of Quick Sort where the pivot is chosen randomly to improve performance in the average case. The program sorts the array and reports the number of comparisons made during sorting. The number of comparisons gives insight into the efficiency of the algorithm for a given input.

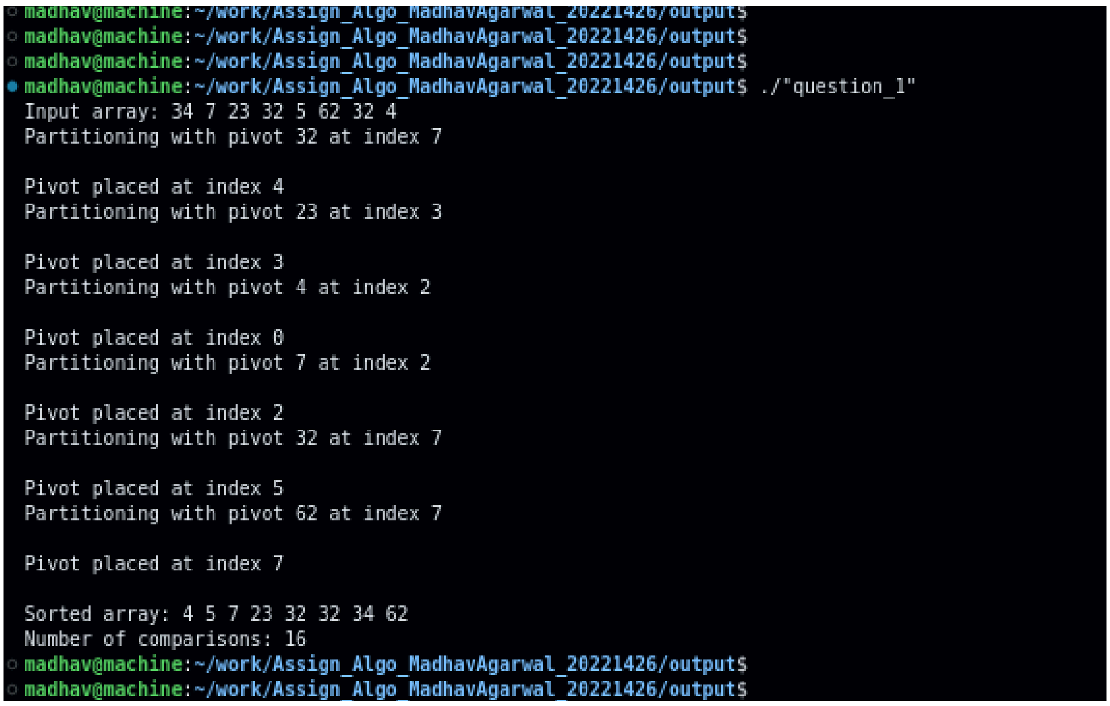
Time Complexity:

1. Best Case:
   - When the pivot divides the array into two equal halves.
   - Complexity: O(n log n)
2. Average Case:
   - On average, the randomized pivot results in balanced partitions.
   - Complexity: O(n log n)
3. Worst Case:
   - The worst case occurs if the partition always chooses the smallest or largest element as the pivot, leading to highly unbalanced partitions.
   - Complexity: O(n^2)

Space Complexity:

- Since this implementation uses recursion, the space complexity is determined by the depth of the recursion stack.
- Space Complexity: O(log n) on average for balanced partitions and O(n) in the worst case due to recursion depth.

Output :



```
○ madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
○ madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
○ madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
● madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$ ./"question_1"
  Input array: 34 7 23 32 5 62 32 4
  Partitioning with pivot 32 at index 7

  Pivot placed at index 4
  Partitioning with pivot 23 at index 3

  Pivot placed at index 3
  Partitioning with pivot 4 at index 2

  Pivot placed at index 0
  Partitioning with pivot 7 at index 2

  Pivot placed at index 2
  Partitioning with pivot 32 at index 7

  Pivot placed at index 5
  Partitioning with pivot 62 at index 7

  Pivot placed at index 7

  Sorted array: 4 5 7 23 32 32 34 62
  Number of comparisons: 16
○ madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
○ madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
```

Question 2 : Write a program to find the ith smallest element of an array using Randomized Select.

Solution :

This program finds the ith smallest element of an array using the Randomized Select algorithm. Randomized Select is a variation of the Quick Select algorithm where the pivot is chosen randomly to improve the average-case performance. The program takes the value of i as input and finds the ith smallest element in the array

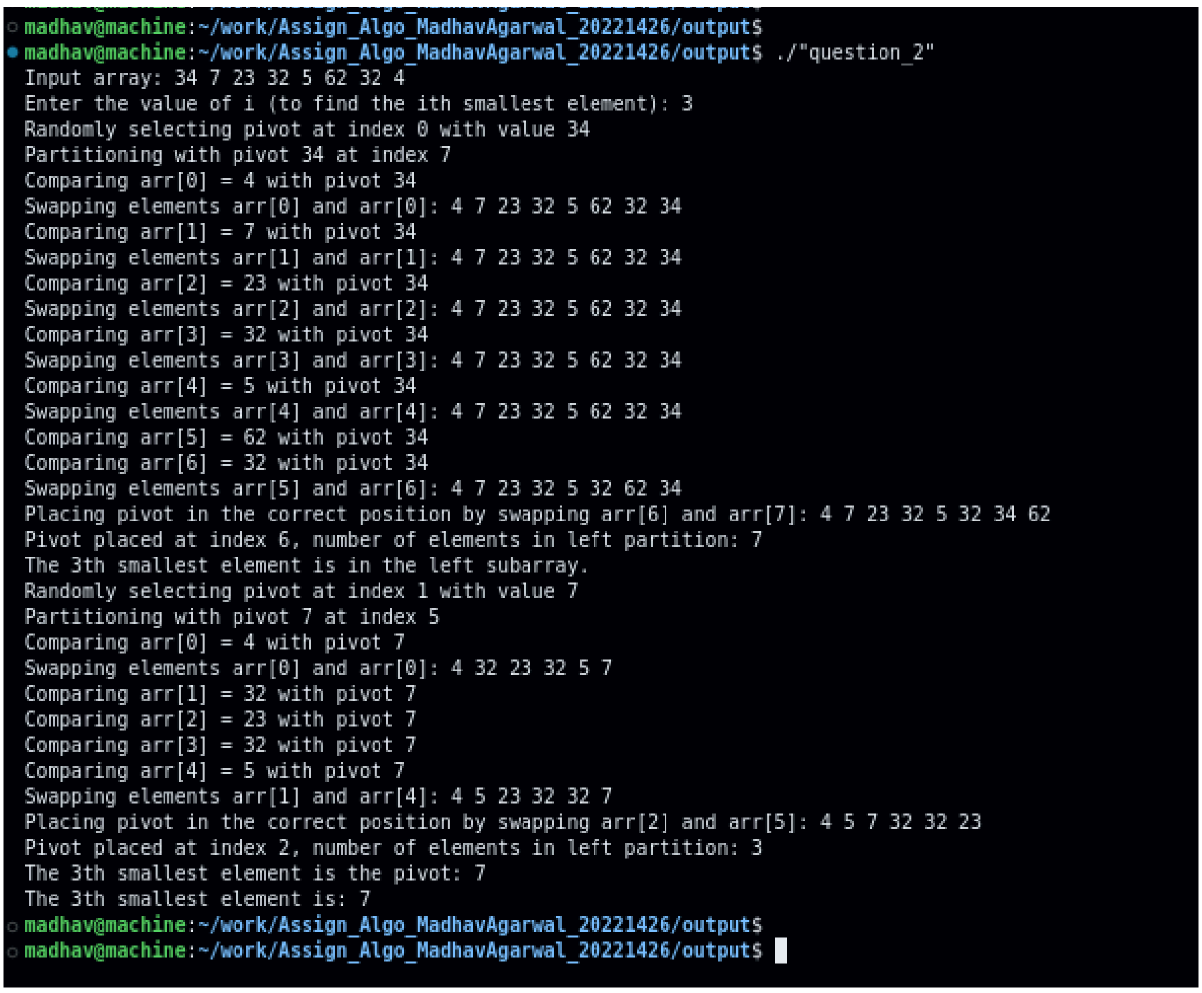
Time Complexity:

1. Best Case:
   - Complexity: O(n), where n is the number of elements in the array.
2. Average Case:
   - Complexity: O(n) on average.
3. Worst Case:
   - Complexity: $O(n^2)$

Space Complexity:

- The algorithm uses recursion, and the space complexity depends on the depth of the recursion stack.
- Space Complexity: O(log n) on average (if partitions are balanced) and O(n) in the worst case due to recursion depth.

Output :



```
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$ ./"question_2"
Input array: 34 7 23 32 5 62 32 4
Enter the value of i (to find the ith smallest element): 3
Randomly selecting pivot at index 0 with value 34
Partitioning with pivot 34 at index 7
Comparing arr[0] = 4 with pivot 34
Swapping elements arr[0] and arr[0]: 4 7 23 32 5 62 32 34
Comparing arr[1] = 7 with pivot 34
Swapping elements arr[1] and arr[1]: 4 7 23 32 5 62 32 34
Comparing arr[2] = 23 with pivot 34
Swapping elements arr[2] and arr[2]: 4 7 23 32 5 62 32 34
Comparing arr[3] = 32 with pivot 34
Swapping elements arr[3] and arr[3]: 4 7 23 32 5 62 32 34
Comparing arr[4] = 5 with pivot 34
Swapping elements arr[4] and arr[4]: 4 7 23 32 5 62 32 34
Comparing arr[5] = 62 with pivot 34
Comparing arr[6] = 32 with pivot 34
Swapping elements arr[5] and arr[6]: 4 7 23 32 5 32 62 34
Placing pivot in the correct position by swapping arr[6] and arr[7]: 4 7 23 32 5 32 34 62
Pivot placed at index 6, number of elements in left partition: 7
The 3th smallest element is in the left subarray.
Randomly selecting pivot at index 1 with value 7
Partitioning with pivot 7 at index 5
Comparing arr[0] = 4 with pivot 7
Swapping elements arr[0] and arr[0]: 4 32 23 32 5 7
Comparing arr[1] = 32 with pivot 7
Comparing arr[2] = 23 with pivot 7
Comparing arr[3] = 32 with pivot 7
Comparing arr[4] = 5 with pivot 7
Swapping elements arr[1] and arr[4]: 4 5 23 32 32 7
Placing pivot in the correct position by swapping arr[2] and arr[5]: 4 5 7 32 32 23
Pivot placed at index 2, number of elements in left partition: 3
The 3th smallest element is the pivot: 7
The 3th smallest element is: 7
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
```

Question 3 : Write a program to determine the minimum spanning tree of a graph using Kruskal’s algorithm

Solution :

This program finds the Minimum Spanning Tree (MST) of a graph using Kruskal's Algorithm. The graph is represented using edges, and the edges are sorted by weight. The algorithm uses a disjoint set (Union-Find) to avoid cycles and ensures that the resultant tree is the minimum spanning one. The program prints the edges of the MST and their weights.

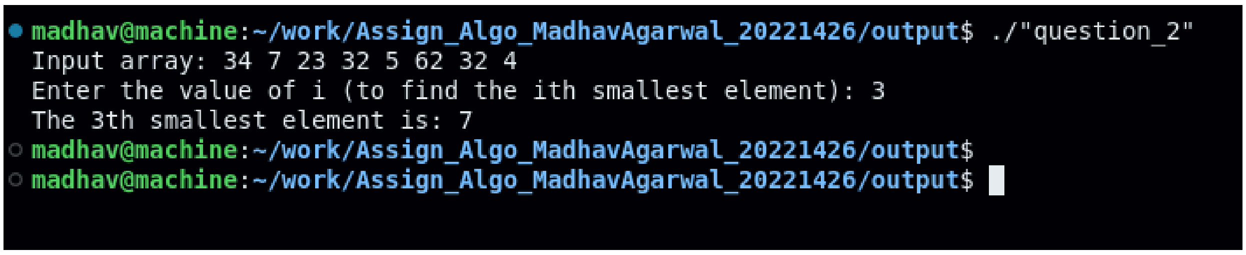
Time Complexity:

- The overall time complexity is dominated by sorting the edges, which is O(E log E).
- Therefore, the total time complexity is O(E log E + E α(V)), which simplifies to O(E log E), as α(V) is very small.

Space Complexity:

- Disjoint Set Data Structure: Space for storing parent and rank arrays, which is O(V).
- Result Storage: Space for storing the MST edges, which is O(V).
- Graph Representation: Space for storing the edges of the graph, which is O(E).

Output :



```
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$ ./"question_2"
Input array: 34 7 23 32 5 62 32 4
Enter the value of i (to find the ith smallest element): 3
The 3th smallest element is: 7
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$ 
```

Question 4 : Write a program to implement the Bellman-Ford algorithm to find the shortest paths from a given source node to all other nodes in a graph

Solution :

This program implements the Bellman-Ford algorithm to find the shortest paths from a given source node to all other nodes in a graph. The algorithm iterates through all edges and relaxes them to find the minimum distance. It can handle graphs with negative-weight edges and detects negative-weight cycles. The program takes the number of vertices, edges, and the source node as input and prints the shortest distances from the source node.

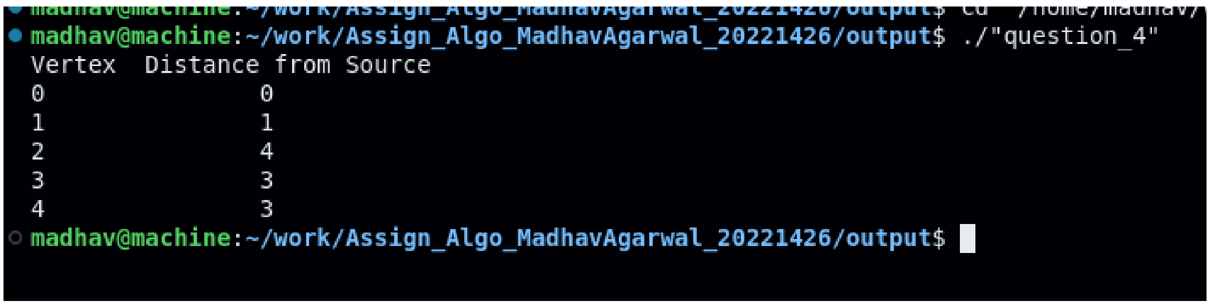
Time Complexity:

- O(V * E), where V is the number of vertices and E is the number of edges.

Space Complexity:

- O(V + E)

Output :



```
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$ cd "/home/madhav/
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$ ./"question_4"
Vertex  Distance from Source
0               0
1               1
2               4
3               3
4               3
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$ 
```

Question 5 : Write a program to implement a B-Tree

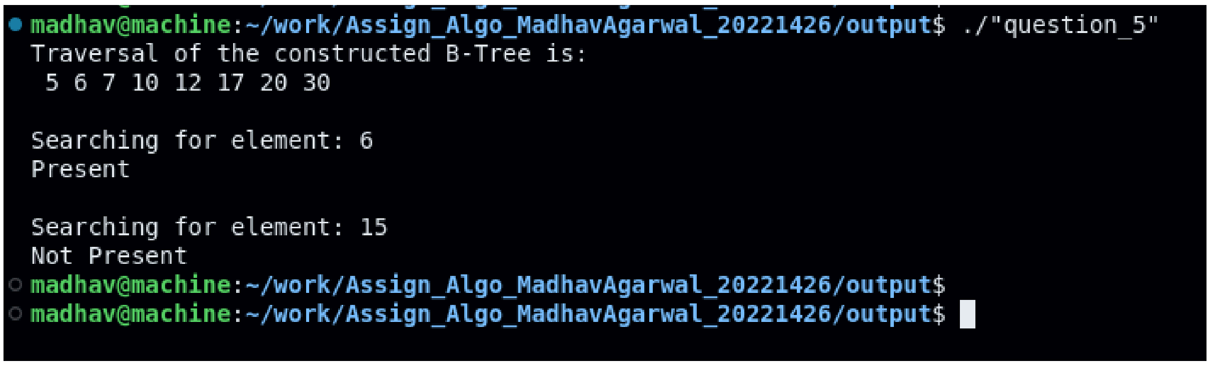
Solution :

This program implements a B-Tree with basic insertion and traversal functionalities. A B-Tree is a self-balancing tree data structure that maintains sorted data and allows search, insertion, and deletion operations in logarithmic time. The B-Tree implemented here supports inserting new keys and traversing the tree, displaying the keys in sorted order.

Time Complexity:
- Insertion and Search: O(log N)
- Traversal: O(N)

Space Complexity:
- O(N)

Output :



```
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$ ./"question_5"
Traversal of the constructed B-Tree is:
 5 6 7 10 12 17 20 30

Searching for element: 6
Present

Searching for element: 15
Not Present
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
```

Question 6 : Write a program to implement the Tree Data structure, which supports the following operations:
a. Insert
b. Search

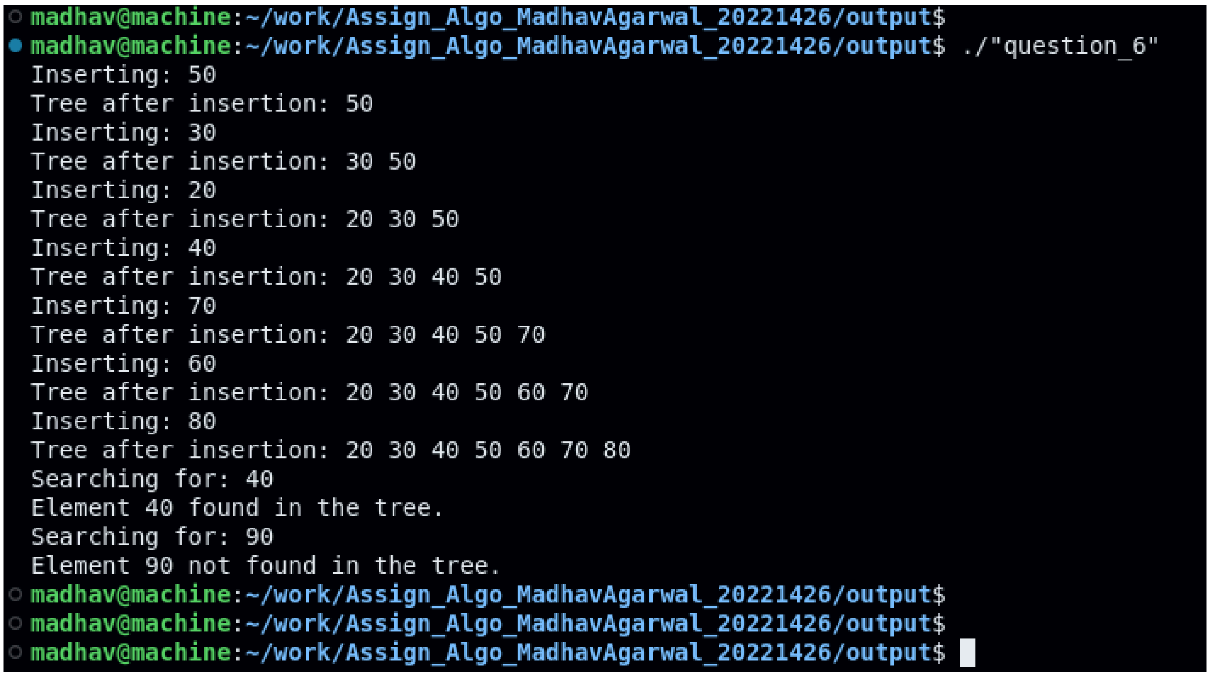
Solution :

This program implements a simple Binary Search Tree (BST) that supports insertion and searching operations. The program allows inserting new keys into the tree, searching for existing keys, and printing the tree in in-order traversal after every insertion. This gives a sorted view of the keys in the tree, making it easier to visualize the current structure of the tree.

Time Complexity:
- Insertion and Search: O(log N) (balanced) / O(N) (skewed)
- Traversal: O(N)

Space Complexity:
- O(N) (due to recursion stack)

Output :



```
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$ ./"question_6"
Inserting: 50
Tree after insertion: 50
Inserting: 30
Tree after insertion: 30 50
Inserting: 20
Tree after insertion: 20 30 50
Inserting: 40
Tree after insertion: 20 30 40 50
Inserting: 70
Tree after insertion: 20 30 40 50 70
Inserting: 60
Tree after insertion: 20 30 40 50 60 70
Inserting: 80
Tree after insertion: 20 30 40 50 60 70 80
Searching for: 40
Element 40 found in the tree.
Searching for: 90
Element 90 not found in the tree.
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
```

Question 7 : Write a program to search a pattern in a given text using the KMP algorithm

Solution :

This program implements the Knuth-Morris-Pratt (KMP) pattern matching algorithm to search for a given pattern in a given text. The KMP algorithm uses the LPS (Longest Prefix Suffix) array to avoid unnecessary comparisons, improving efficiency. The program prints the starting index of each occurrence of the pattern in the text.

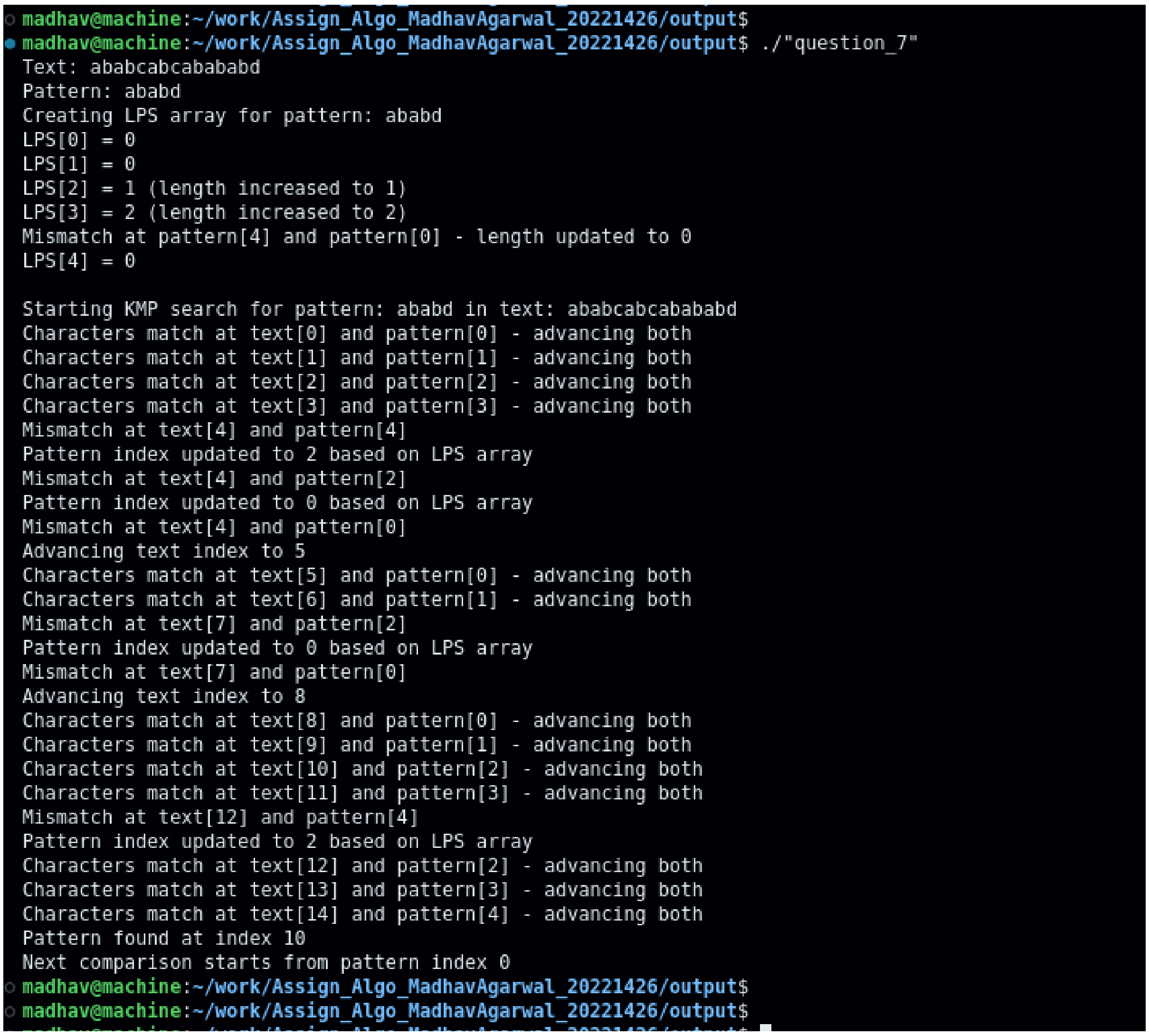
Time Complexity:

- Preprocessing (LPS array creation): O(m), where m is the length of the pattern.
- Pattern Search: O(n), where n is the length of the text.
- Overall Time Complexity: O(n + m)

Space Complexity:

- Space for LPS Array: O(m)

Output :



```
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$ ./"question_7"
Text: ababcabcabababd
Pattern: ababd
Creating LPS array for pattern: ababd
LPS[0] = 0
LPS[1] = 0
LPS[2] = 1 (length increased to 1)
LPS[3] = 2 (length increased to 2)
Mismatch at pattern[4] and pattern[0] - length updated to 0
LPS[4] = 0

Starting KMP search for pattern: ababd in text: ababcabcabababd
Characters match at text[0] and pattern[0] - advancing both
Characters match at text[1] and pattern[1] - advancing both
Characters match at text[2] and pattern[2] - advancing both
Characters match at text[3] and pattern[3] - advancing both
Mismatch at text[4] and pattern[4]
Pattern index updated to 2 based on LPS array
Mismatch at text[4] and pattern[2]
Pattern index updated to 0 based on LPS array
Mismatch at text[4] and pattern[0]
Advancing text index to 5
Characters match at text[5] and pattern[0] - advancing both
Characters match at text[6] and pattern[1] - advancing both
Mismatch at text[7] and pattern[2]
Pattern index updated to 0 based on LPS array
Mismatch at text[7] and pattern[0]
Advancing text index to 8
Characters match at text[8] and pattern[0] - advancing both
Characters match at text[9] and pattern[1] - advancing both
Characters match at text[10] and pattern[2] - advancing both
Characters match at text[11] and pattern[3] - advancing both
Mismatch at text[12] and pattern[4]
Pattern index updated to 2 based on LPS array
Characters match at text[12] and pattern[2] - advancing both
Characters match at text[13] and pattern[3] - advancing both
Characters match at text[14] and pattern[4] - advancing both
Pattern found at index 10
Next comparison starts from pattern index 0
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
```

Question 8 : Write a program to implement a Suffix tree

Solution :

This program builds a suffix tree for a given string using Ukkonen's algorithm. The suffix tree is printed at relevant steps, showing the insertion of new leaves and splits that occur during the construction. The suffix tree is useful for various string operations such as pattern matching, substring search, and more.

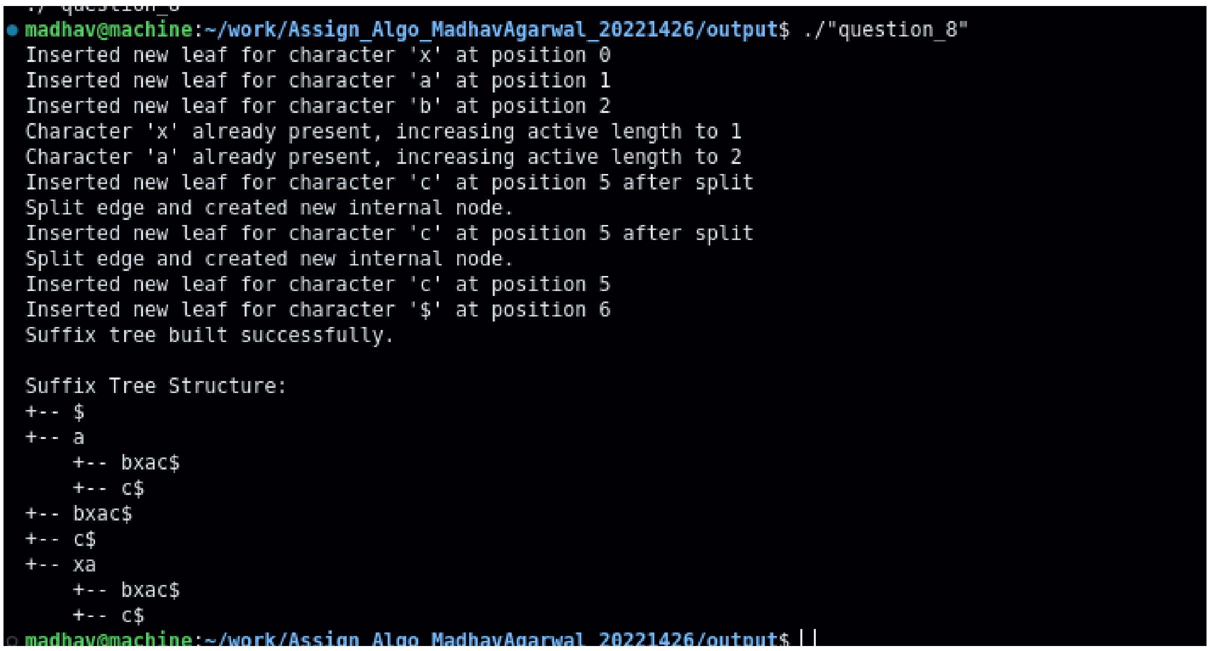
Time Complexity:

- O(N), where N is the length of the text. This is achieved due to Ukkonen’s algorithm, which ensures each character is processed only a constant number of times.

Space Complexity:

- O(N), where N is the length of the text. This is because the tree stores all suffixes of the given string, including the unique end character.

Output :



```
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$ ./"question_8"
Inserted new leaf for character 'x' at position 0
Inserted new leaf for character 'a' at position 1
Inserted new leaf for character 'b' at position 2
Character 'x' already present, increasing active length to 1
Character 'a' already present, increasing active length to 2
Inserted new leaf for character 'c' at position 5 after split
Split edge and created new internal node.
Inserted new leaf for character 'c' at position 5 after split
Split edge and created new internal node.
Inserted new leaf for character 'c' at position 5
Inserted new leaf for character '$' at position 6
Suffix tree built successfully.

Suffix Tree Structure:
+-- $
+-- a
    +-- bxac$
    +-- c$
+-- bxac$
+-- c$
+-- xa
    +-- bxac$
    +-- c$
madhav@machine:~/work/Assign_Algo_MadhavAgarwal_20221426/output$
```