Data Structures Odyssey: Exploring the Foundations of Computing

Ex. No.:07	Implementation of Queue using Array and Linked List Implementation	Date:11/04/2024

Write a C program to implement a Queue using Array and linked List implementation and execute the following operation on stack.

(i) Enqueue

(ii) Dequeue

(iii) Display the elements in a Queue

Algorithm:

1) Start
2) 1. For Queue using Array:
3) Initialize an array of fixed size to store the queue elements.
4) Maintain two pointers, front and rear, to track the front and rear of the queue.
5) Implement the following operations:
- Enqueue: Add an element to the rear of the queue by incrementing the rear pointer. - Dequeue: Remove an element from the front of the queue by incrementing the front pointer.
- isEmpty: Check if the queue is empty by comparing the front and rear pointers.
- isFull: Check if the queue is full by comparing the rear pointer with the array size.

6) For Queue using Linked List:
7) Define a Node structure with data and a pointer to the next Node.
8) Maintain pointers to the front and rear Nodes of the queue.
9) Implement the following operations:
- Enqueue: Create a new Node with the given data and update the rear pointer.
- Dequeue: Remove the front Node and update the front pointer.
- isEmpty: Check if the queue is empty by comparing the front pointer with NULL. - Display: Traverse the linked list to display all elements in the queue. 10) Stop

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>

struct queue
{    int data;    struct
queue *next;
};

struct queue *addq(struct queue *front); struct
queue *delq(struct queue *front);

int main()
{
   struct queue *front = NULL;
int option;

   do
   {
      printf("\n1. Add to Queue");
printf("\n2. Delete from Queue");
printf("\n3. Exit");       printf("\nSelect
an option: ");       scanf("%d",
&option);

      switch(option)
      {
case 1:
            front = addq(front);
printf("\nElement added to the queue.");
```

```
break;            case 2:               front =
delq(front);                break;            case 3:
exit(0);
        }
    } while(1);

    return 0;
}

struct queue *addq(struct queue *front)
{
    struct queue *new_node = (struct queue*)malloc(sizeof(struct queue));
if(new_node == NULL)
    {
        printf("Insufficient memory.");
return front;
    }

    printf("\nEnter data: ");
scanf("%d", &new_node->data);
new_node->next = NULL;

    if(front == NULL)
    {
        front = new_node;
    }
else
    {



            struct queue *temp = front;         while(temp-
>next != NULL)
        {
```

```
            temp = temp->next;
        }
        temp->next = new_node;
    }

    return front;
}

struct queue *delq(struct queue *front)
{
    if(front == NULL)
    {
        printf("Queue is empty.");
return front;
    }

    struct queue *temp = front;
printf("Deleted data: %d", front->data);
front = front->next;     free(temp);

    return front; }
```

OUTPUT:

```
1. Add to Queue
2. Delete from Queue
3. Exit
Select an option: 1

Enter data: 2

Element added to the queue.
1. Add to Queue
2. Delete from Queue
3. Exit
Select an option: 1

Enter data: 3

Element added to the queue.
1. Add to Queue
2. Delete from Queue
3. Exit
Select an option: 1

Enter data: 4

Element added to the queue.
1. Add to Queue
2. Delete from Queue
3. Exit
Select an option: 2
Deleted data: 2
1. Add to Queue
2. Delete from Queue
3. Exit
Select an option: 2
Deleted data: 3
1. Add to Queue
2. Delete from Queue
3. Exit
Select an option: 4

1. Add to Queue
2. Delete from Queue
3. Exit
Select an option: 3
aiml231501179@cselab:~$
```

RESULT: Thus, the program was successfully executed.

Ex. No.:08	Tree Traversal	Date:18/04/2024

Write a C program to implement a Binary tree and perform the following tree traversal operation.

(i) Inorder Traversal

(ii) Preorder Traversal

(iii) Postorder Traversal

Algorithm:

1) Start
2) Define a Node structure with data, left child pointer, and right child pointer. 3) Create functions for the following traversal methods:
4) Inorder traversal:
- Recursively call the function on the left child.
- Print the data of the current Node.
- Recursively call the function on the right child.
5) Preorder traversal:
- Print the data of the current Node.
- Recursively call the function on the left child.
- Recursively call the function on the right child.
6) Postorder traversal:
- Recursively call the function on the left child.
- Recursively call the function on the right child.
- Print the data of the current Node.
7) Initialize the root of the binary tree.
8) Call the traversal functions with the root Node to perform inorder, preorder, and postorder traversal. 9) Stop

PROGRAM;

```
#include <stdio.h>
#include <stdlib.h>

struct node { int
element; struct
node* left; struct
node* right;
};

struct node* createNode(int val)
{
struct node* Node = (struct node*)malloc(sizeof(struct node));
Node->element = val;
Node->left = NULL;
Node->right = NULL;

return (Node);
}

void traversePreorder(struct node* root)
{
if (root == NULL) return;
printf(" %d ", root->element); traversePreorder(root->left);
traversePreorder(root->right);
}


void traverseInorder(struct node* root)
{
```

```
if (root == NULL) return;
traverseInorder(root->left);
printf(" %d ", root->element);
traverseInorder(root->right);
}
void traversePostorder(struct node* root)
{
if (root == NULL) return;
traversePostorder(root->left);
traversePostorder(root->right); printf("
%d ", root->element);
}

int main()
{
struct node* root = createNode(36); root-
>left = createNode(26); root->right =
createNode(46); root->left->left =
createNode(21); root->left->right =
createNode(31); root->left->left->left =
createNode(11); root->left->left->right =
createNode(24); root->right->left =
createNode(41); root->right->right =
createNode(56); root->right->right->left =
createNode(51); root->right->right->right =
createNode(66);

printf("\n The Preorder traversal of given binary tree is -\n"); traversePreorder(root);

printf("\n The Inorder traversal of given binary tree is -\n"); traverseInorder(root);

printf("\n The Postorder traversal of given binary tree is -\n");
traversePostorder(root);
```

```
return 0;
}
```

OUTPUT:

```
aiml231501167@cselab:~$ ./a.out

 The Preorder traversal of given binary tree is -
 36  26  21  11  24  31  46  41  56  51  66
 The Inorder traversal of given binary tree is -
 11  21  24  26  31  36  41  46  51  56  66
 The Postorder traversal of given binary tree is -
 11  24  21  31  26  41  51  66  56  46  36 aiml231501167@cselab:~$
```

RESULT: Thus, the program was successfully executed.

Ex. No.:09	Implementation of Binary Search tree	Date:25/04/2024

Write a C program to implement a Binary Search Tree and perform the following operations.

(i) Insert

(ii) Delete

(iii) Search

(iv) Display

Algorithm:

1) Start
2) Define a Node structure with data, left child pointer, and right child pointer.
3) Initialize a root pointer to NULL.
4) Create functions for the following operations: a. Insert:
- If the tree is empty, create a new Node and set it as the root.
- Otherwise, traverse the tree starting from the root:
- If the data is less than the current Node's data, move to the left child.
- If the data is greater than the current Node's data, move to the right child. - Repeat until reaching a NULL child pointer, then insert the new Node. b. Search:
- Start from the root and compare the data with each Node:
- If the data matches, return the Node.
- If the data is less than the current Node's data, move to the left child.
- If the data is greater than the current Node's data, move to the right child.
- Repeat until finding the data or reaching a NULL child pointer.

5) Test the operations by inserting elements into the tree and searching for specific values. 6) Stop

PROGRAM;

```
#include <stdio.h>
#include <stdlib.h> struct
BinaryTreeNode { int
key;
struct BinaryTreeNode *left, *right;
};

struct BinaryTreeNode* newNodeCreate(int value)
{
struct BinaryTreeNode* temp =
(struct BinaryTreeNode*)malloc(
sizeof(struct BinaryTreeNode));
temp->key = value; temp->left =
temp->right = NULL; return temp;
}
struct BinaryTreeNode*
searchNode(struct BinaryTreeNode* root, int target)
{
if (root == NULL || root->key == target) { return
root;
}
if (root->key < target) { return
searchNode(root->right, target);
}
return searchNode(root->left, target);
}

struct BinaryTreeNode*
insertNode(struct BinaryTreeNode* node, int value)
```

```
{
if (node == NULL) { return
newNodeCreate(value);
}
if (value < node->key) { node->left =
insertNode(node->left, value);
}
else if (value > node->key) { node->right =
insertNode(node->right, value);
}
return node;
}

void postOrder(struct BinaryTreeNode* root)
{
if (root != NULL) {
postOrder(root->left);
postOrder(root->right); printf("
%d ", root->key);
}
}

void inOrder(struct BinaryTreeNode* root)
{
if (root != NULL) {
inOrder(root->left); printf("
%d ", root->key);
inOrder(root->right);
}
}
void preOrder(struct BinaryTreeNode* root)
{
if (root != NULL) { printf("
%d ", root->key);
```

```
preOrder(root->left);
preOrder(root->right);
}
}

struct BinaryTreeNode* findMin(struct BinaryTreeNode* root)
{
if (root == NULL) { return
NULL;
}
else if (root->left != NULL) { return
findMin(root->left);
}
return root;
}

struct BinaryTreeNode* delete (struct BinaryTreeNode* root, int
x)
{
if (root == NULL) return
NULL;

if (x > root->key) { root->right =
delete (root->right, x);
}
else if (x < root->key) { root->left
= delete (root->left, x);
}
else {
if (root->left == NULL && root->right == NULL) {
free(root); return NULL;
}
```

```
else if (root->left == NULL ||
root->right == NULL) { struct
BinaryTreeNode* temp; if
(root->left == NULL) { temp =
root->right;
} else { temp =
root->left;
} free(root);
return temp;
} else
{
struct BinaryTreeNode* temp = findMin(root->right);
root->key = temp->key; root->right = delete (root-
>right, temp->key);
}
}
return root;
}

int main()
{
struct BinaryTreeNode* root = NULL;

root = insertNode(root, 50);
insertNode(root, 30);
insertNode(root, 20);
insertNode(root, 40);
insertNode(root, 70);
insertNode(root, 60);
insertNode(root, 80); if
(searchNode(root, 60) != NULL) {
printf("60 found");
```

```
} else { printf("60 not
found");
}

printf("\n"); postOrder(root);
printf("\n");

preOrder(root);
printf("\n"); inOrder(root);
printf("\n");

struct BinaryTreeNode* temp = delete (root, 70);
printf("After Delete: \n"); inOrder(root);

return 0;
}
```

OUTPUT:

```
aiml231501167@cselab:~$ ./a.out

 The Preorder traversal of given binary tree is -
 36  26  21  11  24  31  46  41  56  51  66
 The Inorder traversal of given binary tree is -
 11  21  24  26  31  36  41  46  51  56  66
 The Postorder traversal of given binary tree is -
 11  24  21  31  26  41  51  66  56  46  36 aiml231501167@cselab:~$
```

RESULT: Thus, the program was successfully executed.

Ex. No.: 10	Implementation of AVL Tree	Date:02/05/2024

Write a function in C program to insert a new node with a given value into an AVL tree. Ensure that the tree remains balanced after insertion by performing rotations if necessary. Repeat the above operation to delete a node from AVL tree.

Algorithm:

1) Start
2) Define the AVL Node Structure.
3) Implement Rotation Operations (left and right rotations).
4) Insert new nodes into the AVL tree, updating heights and balancing as needed.
5) Delete nodes from the AVL tree, updating heights and balancing as needed.
6) Implement traversal functions (in-order, pre-order, post-order) to navigate through the tree.
7) Implement a search function to find specific elements within the AVL tree.
8) Test the AVL tree implementation with various scenarios.
9) Optionally, optimize the implementation for better performance. 10) Stop

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>

struct node
{  int data;    struct
node* left;    struct
node* right;
 int ht;
};

struct node* root = NULL;

struct node* create(int);   struct node*
insert(struct node*, int);   struct node*
delete(struct node*, int);   struct node*
search(struct node*, int);   struct node*
rotate_left(struct node*);   struct node*
rotate_right(struct node*);   int
balance_factor(struct node*);   int
height(struct node*);   void
inorder(struct node*);   void
preorder(struct node*);   void
postorder(struct node*);

int main()
{
   int user_choice, data;

   char user_continue = 'y';
struct node* result = NULL;
```

```
   while (user_continue == 'y' || user_continue == 'Y')
   {
      printf("\n\n------- AVL TREE --------\n");
printf("\n1. Insert");          printf("\n2.
Delete");          printf("\n3. Search");
printf("\n4. Inorder");          printf("\n5.
Preorder");          printf("\n6. Postorder");
printf("\n7. EXIT");

      printf("\n\nEnter Your Choice: ");
scanf("%d", &user_choice);

      switch(user_choice)
      {
case 1:
            printf("\nEnter data: ");
scanf("%d", &data);                root
= insert(root, data);
break;

         case 2:
            printf("\nEnter data: ");
scanf("%d", &data);                root
= delete(root, data);
break;


            case 3:
            printf("\nEnter data: ");
scanf("%d", &data);                result
= search(root, data);                if
(result == NULL)
            {
```

```
                printf("\nNode not found!");
        }
else
        {
          printf("\n Node found");
        }
break;          case 4:
inorder(root);
break;

      case 5:
preorder(root);
break;

      case 6:
        postorder(root);
break;

      case 7:
        printf("\n\tProgram Terminated\n");
        return 1;

      default:

        printf("\n\tInvalid Choice\n");
    }

    printf("\n\nDo you want to continue? ");
scanf(" %c", &user_continue);
  }

  return 0;
}
```

```
struct node* create(int data)
{
   struct node* new_node = (struct node*) malloc (sizeof(struct node));
if (new_node == NULL)
   {
      printf("\nMemory can&'t be allocated\n");
return NULL;
   }
   new_node->data = data;
new_node->left = NULL;
new_node->right = NULL;      return
new_node;
}
struct node* rotate_left(struct node* root)
{
   struct node* right_child = root->right;
root->right = right_child->left;
right_child->left = root;      root->ht =
height(root);      right_child->ht =
height(right_child);   return right_child;

}
struct node* rotate_right(struct node* root)
{
   struct node* left_child = root->left;      root-
>left = left_child->right;      left_child->right =
root;


   root->ht = height(root);      left_child-
>ht = height(left_child);            return
left_child;
}


int balance_factor(struct node* root)
```

```
{     int lh, rh;     if (root
== NULL)         return 0;
if (root->left == NULL)
      lh = 0;     else         lh
= 1 + root->left->ht;     if
(root->right == NULL)
      rh = 0;     else         rh =
1 + root->right->ht;
return lh - rh;
}
int height(struct node* root)
 {     int lh,
rh;     if
(root ==
NULL)

   {
      return 0;
   }
   if (root->left == NULL)
      lh = 0;     else         lh
= 1 + root->left->ht;     if
(root->right == NULL)
      rh = 0;     else         rh =
1 + root->right->ht;


   if (lh > rh)
return (lh);
return (rh);
}

struct node* insert(struct node* root, int data)
{
   if (root == NULL)
```

```
  {
    struct node* new_node = create(data);
if (new_node == NULL)
    {
      return NULL;
    }
    root = new_node;
  }
  else if (data > root->data)
  {
        root->right = insert(root->right, data);
if (balance_factor(root) == -2)
    {
      if (data > root->right->data)
      {
        root = rotate_left(root);
      }
else
      {
        root->right = rotate_right(root->right);
root = rotate_left(root);
      }
    }
}
else
  {
    root->left = insert(root->left, data);
if (balance_factor(root) == 2)
    {
      if (data < root->left->data)
      {
        root = rotate_right(root);
```

```
        }
else
        {
            root->left = rotate_left(root->left);
root = rotate_right(root);
        }
      }
    }
    root->ht = height(root);
return root;

}


struct node * delete(struct node *root, int x)
{
    struct node * temp = NULL;

    if (root == NULL)
    {
        return NULL;
    }

    if (x > root->data)
    {
        root->right = delete(root->right, x);
if (balance_factor(root) == 2)
      {
          if (balance_factor(root->left) >= 0)
          {
              root = rotate_right(root);
          }
else
          {
```

```
            root->left = rotate_left(root->left);
root = rotate_right(root);
         }
      }
   }
   else if (x < root->data)
   {
      root->left = delete(root->left, x);


            if (balance_factor(root) == -2)
      {
         if (balance_factor(root->right) <= 0)
         {
            root = rotate_left(root);
         }
else
{
root->right = rotate_right(root->right);   root
= rotate_left(root);
}
}
} else
{
if (root->right != NULL)
{
temp = root->right;   while
(temp->left != NULL)
temp = temp->left;

root->data = temp->data;   root->right =
delete(root->right, temp->data);   if
(balance_factor(root) == 2)
```

```
{
if (balance_factor(root->left) >= 0)
{
root = rotate_right(root);
}


else
{
root->left = rotate_left(root->left);   root
= rotate_right(root);
}
}
}
else
{
return (root->left);
}
}
root->ht = height(root);   return
(root);
}
struct node* search(struct node* root, int key)
{
 if(root == NULL)
{
return NULL;
}

if(root->data == key)
{
return root;
}
```

```
if(key > root->data)
{
search(root->right, key);
}

else
{       search(root->left,
key);
}
}
void inorder(struct node* root)
{
if (root == NULL)
{
return;
}
inorder(root->left);   printf("%d
", root->data);   inorder(root-
>right);
}
void preorder(struct node* root)
{
   if(root == NULL)
{
return;
}
printf("%d ", root->data); preorder(root-
>left); preorder(root->right);
}
void postorder(struct node* root)
{
if(root == NULL)
```

```
{
return;
}

postorder(root->left);
postorder(root->right);   printf("%d
", root->data);
}
```

OUTPUT:

```
------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 1

Enter data: 15


Do you want to continue? y


------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 1

Enter data: 20


Do you want to continue? y


------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 1

Enter data: 25


Do you want to continue? y


------- AVL TREE --------

1. Insert
```

```
Enter data: 30

 Node found

Do you want to continue? y


------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 4
15 20 30 30

Do you want to continue? y


------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 5
20 15 30 30

Do you want to continue? y


------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
5. Preorder
6. Postorder
7. EXIT

Enter Your Choice: 6
15 30 30 20

Do you want to continue? y


------- AVL TREE --------

1. Insert
2. Delete
3. Search
4. Inorder
```

RESULT: Thus, the program was successfully executed.

Ex. No.: 11	Topological Sorting	Date:09/05/2024

Write a C program to create a graph and display the ordering of vertices.

Algorithm:

1) Start
2) Initialize an empty stack to store the topologically sorted elements.
3) Initialize a set to track visited nodes.
4) Start a depth-first search (DFS) from any unvisited node in the graph.
5) During DFS traversal: a. Mark the current node as visited.
b. Recursively visit all adjacent nodes that are not visited yet.
6) Once a node has no unvisited adjacent nodes, push it onto the stack.
7) Repeat steps 3-5 until all nodes are visited.
Pop elements from the stack to get the topologically sorted order. 8)
Stop

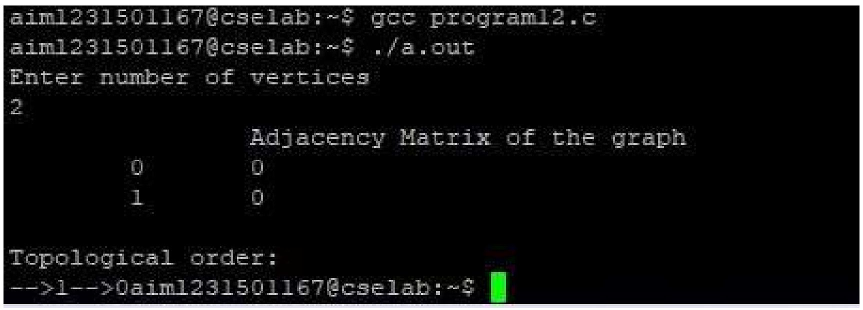
PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>

int s[100], j, res[100; void
AdjacencyMatrix(int a[][100], int n) {
 int i, j; for (i = 0; i < n;
i++) { for (j = 0; j <=
n; j++) { a[i][j] = 0;
} } for (i = 1; i < n;
i++) { for (j = 0; j < i;
j++) { a[i][j] = rand()
% 2; a[j][i] = 0;
}
}
}

void dfs(int u, int n, int a[][100]) {
int v; s[u] = 1; for (v = 0; v < n - 1;
v++) { if (a[u][v] == 1 && s[v] ==
0) { dfs(v, n, a);

}
} j +=
1;
res[j]
= u;
}

void topological_order(int n, int a[][100]) {
```

```
int i, u; for (i = 0; i < n;
i++) { s[i] = 0; } j = 0;
for (u = 0; u < n; u++) {
if (s[u] == 0) { dfs(u, n,
a);
}
}
return;
}
int main() { int
a[100][100], n, i, j;

printf("Enter number of vertices\n");
scanf("%d", &n);

AdjacencyMatrix(a, n);  printf("\t\tAdjacency Matrix of the graph\n"); /* PRINT
ADJACENCY MATRIX */

for (i = 0; i < n; i++) {
for (j = 0; j < n; j++) {
printf("\t%d", a[i][j]);
}
printf("\n");
}


printf("\nTopological order:\n");

topological_order(n, a);

for (i = n; i >= 1; i--) { printf("-->%d",
res[i]);
}
```

```
return 0;

}
```

OUTPUT:



```
aiml231501167@cselab:~$ gcc program12.c
aiml231501167@cselab:~$ ./a.out
Enter number of vertices
2
                Adjacency Matrix of the graph
        0       0
        1       0

Topological order:
-->1-->0aiml231501167@cselab:~$
```

RESULT: Thus, the program was successfully executed.

Ex. No.:12	Graph Traversal	Date:09/05/2024

Write a C program to create a graph and perform a Breadth First Search and Depth First Search.

Algorithm:

DFS

1)Start with an empty stack and a set to track visited nodes.
2) Push the starting node onto the stack and mark it as visited.
3) While the stack is not empty, do the following:
4) If the stack is not empty, pop a node from the stack.
Process the node.
5) For each unvisited neighbor of the node, push the neighbor onto the stack and mark it as visited.
6) Repeat steps 3-6 until the stack is empty.
7) Stop

BFS

1) Start with an empty queue and a set to track visited nodes.
2)Enqueue the starting node into the queue and mark it as visited.
3) While the queue is not empty, do the following:
4) If the queue is not empty, dequeue a node from the queue.
Process the node.
5) For each unvisited neighbor of the node, enqueue the neighbor into the queue and mark it as visited.
6) Repeat steps 3-6 until the queue is empty.
7) Stop

PROGRAM: BFS

```
#include <stdio.h>
#include <stdlib.h>

struct node { int
vertex; struct
node* next;

}; struct adj_list {
struct node* head;
};

struct graph { int
num_vertices; struct
adj_list* adj_lists; int*
visited;
};
struct node* new_node(int vertex) { struct node* new_node =
(struct node*)malloc(sizeof(struct node)); new_node->vertex =
vertex; new_node->next = NULL; return new_node;
}
struct graph* create_graph(int n) { struct graph* graph = (struct
graph*)malloc(sizeof(struct graph)); graph->num_vertices = n;
graph->adj_lists = (struct adj_list*)malloc(n * sizeof(struct adj_list));
graph->visited = (int*)malloc(n * sizeof(int));


int i; for (i = 0; i< n; i++) { graph-
>adj_lists[i].head = NULL;
graph->visited[i] = 0;
}
```

```
return graph;
}
void add_edge(struct graph* graph, int src, int dest) {
struct node* new_node1 = new_node(dest);
new_node1->next = graph->adj_lists[src].head;
graph->adj_lists[src].head = new_node1; struct
node* new_node2 = new_node(src); new_node2-
>next = graph->adj_lists[dest].head; graph-
>adj_lists[dest].head = new_node2;
}
void bfs(struct graph* graph, int v) { int queue[1000]; int
front = -1; int rear = -1; graph->visited[v] = 1;
queue[++rear] = v; while (front != rear) { int
current_vertex = queue[++front]; printf("%d ",
current_vertex); struct node* temp = graph-
>adj_lists[current_vertex].head; while (temp != NULL) { int
adj_vertex = temp->vertex;

if (graph->visited[adj_vertex] == 0) { graph-
>visited[adj_vertex] = 1; queue[++rear] =
adj_vertex;
}
temp = temp->next;
}
}
}

int main() { struct graph* graph =
create_graph(6); add_edge(graph, 0, 1);
add_edge(graph, 0, 2); add_edge(graph, 1, 3);
```

```
add_edge(graph, 1, 4); add_edge(graph, 2, 4);
add_edge(graph, 3, 4); add_edge(graph, 3, 5);
add_edge(graph, 4,5); printf("BFS traversal
starting from vertex 0: "); bfs(graph, 0);

return 0;
}
```

DFS:

```
#include <stdio.h>
#include <stdlib.h>

// Globally declared visited array int
vis[100];


struct Graph {
   int V;
int E;
int** Adj;
};

struct Graph* adjMatrix()
{
   struct Graph* G = (struct Graph*)
malloc(sizeof(struct Graph));     if
(!G) {        printf("Memory Error\n");
return NULL;
   }
   G->V = 7;
   G->E = 7;
```

```
    G->Adj = (int**)malloc((G->V) * sizeof(int*));
for (int k = 0; k < G->V; k++) {
        G->Adj[k] = (int*)malloc((G->V) * sizeof(int));
    }

    for (int u = 0; u < G->V; u++) {
for (int v = 0; v < G->V; v++) {
            G->Adj[u][v] = 0;
        }
    }
    G->Adj[0][1] = G->Adj[1][0] = 1;
    G->Adj[0][2] = G->Adj[2][0] = 1;
G->Adj[1][3] = G->Adj[3][1] = 1;
G->Adj[1][4] = G->Adj[4][1] = 1;
G->Adj[1][5] = G->Adj[5][1] = 1;
     G->Adj[1][6] = G->Adj[6][1] = 1;     G-
         >Adj[6][2] = G->Adj[2][6] = 1;

    return G;
}
 void DFS(struct Graph* G, int u)

{     vis[u] = 1;     printf("%d ", u);
for (int v = 0; v < G->V; v++) {
if (!vis[v] && G->Adj[u][v]) {
            DFS(G, v);
        }
    }
}

void DFStraversal(struct Graph* G)
{     for (int i = 0; i < 100; i++)
{        vis[i] = 0;
```

```
    }
    for (int i = 0; i < G->V; i++) {
if (!vis[i]) {
            DFS(G, i);
        }
    }
}
 void main()
{


 struct Graph* G;
    G = adjMatrix();
    DFStraversal(G);
}
```

OUTPUT:

```
BFS traversal starting from vertex 0: 0 2 1 4 3 5
```

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

RESULT: Thus, the program was successfully executed.

Data Structures Odyssey: Exploring the Foundations of Computing

Ex. No.:13	Graph Traversal	Date:16/05/2014

Write a C program to create a graph and find a minimum spanning tree using prim's algorithm.

Algorithm:

1) Start
2) Initialize an empty set to store the minimum spanning tree (MST) and a priority queue to store edges and their weights.
3) Choose a starting node and add it to the MST set.
4) For each edge connected to the starting node, add the edge to the priority queue.
5) While the priority queue is not empty:

a. Extract the edge with the smallest weight from the priority queue.
b. If adding the edge to the MST set does not create a cycle, add the edge to the MST set.
c. For each neighbour of the newly added node in the MST set:
i. If the neighbour is not already in the MST set, add the edge connecting the neighbor to the priority queue.

6) Repeat step 4 until all nodes are included in the MST set.
7) The edges in the MST set form the minimum spanning tree of the graph. 8) Stop

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>

struct node { int

vertex; struct

node* next;


}; struct adj_list {

struct node* head;

};

struct graph { int

num_vertices; struct

adj_list* adj_lists; int*

visited;

};

struct node* new_node(int vertex) { struct node* new_node =

(struct node*)malloc(sizeof(struct node)); new_node->vertex =

vertex; new_node->next = NULL; return new_node;

}

struct graph* create_graph(int n) { struct graph* graph = (struct

graph*)malloc(sizeof(struct graph)); graph->num_vertices = n;

graph->adj_lists = (struct adj_list*)malloc(n * sizeof(struct adj_list));

graph->visited = (int*)malloc(n * sizeof(int));


int i; for (i = 0; i< n;

i++) { graph-

>adj_lists[i].head =

NULL; graph-

>visited[i] = 0;

}


return graph;
```

```
}

void add_edge(struct graph* graph, int src, int dest) {
struct node* new_node1 = new_node(dest);
new_node1->next = graph->adj_lists[src].head;
graph->adj_lists[src].head = new_node1; struct
node* new_node2 = new_node(src); new_node2-
>next = graph->adj_lists[dest].head; graph-
>adj_lists[dest].head = new_node2;
}
void bfs(struct graph* graph, int v) { int queue[1000]; int
front = -1; int rear = -1; graph->visited[v] = 1;
queue[++rear] = v; while (front != rear) { int
current_vertex = queue[++front]; printf("%d ",
current_vertex); struct node* temp = graph-
>adj_lists[current_vertex].head; while (temp != NULL) { int
adj_vertex = temp->vertex; if (graph->visited[adj_vertex]
== 0) { graph->visited[adj_vertex] = 1; queue[++rear] =
adj_vertex;
}

temp = temp->next;
}
}
}

int main() { struct graph* graph =
create_graph(6); add_edge(graph, 0, 1);
add_edge(graph, 0, 2); add_edge(graph, 1, 3);
add_edge(graph, 1, 4); add_edge(graph, 2, 4);
add_edge(graph, 3, 4); add_edge(graph, 3, 5);
add_edge(graph, 4,5); printf("BFS traversal
starting from vertex 0: "); bfs(graph, 0);
```

return 0;

}

OUTPUT:

```
Input the number of vertices: 5
Input the adjacency matrix for the graph:
4
1
2
3
5
9
8
6
7
0
11
12
12
14
15
16
17
18
19
20
21
22
23
24
25
Edge    Weight
0 - 1   9
0 - 2   11
0 - 3   16
0 - 4   21
```

RESULT: Thus, the program was successfully executed.

Data Structures Odyssey: Exploring the Foundations of Computing

Ex. No.:14	Graph Traversal	Date: 16/05/2024

Write a C program to create a graph and find the shortest path using Dijkstra's Algorithm.

Algorithm:

1) Start
2) Initialize the distance from the start node to all other nodes as infinity, except for the start node itself which is 0.
3) Create a priority queue to store nodes and their distances from the start node.
4) Add the start node to the priority queue with a distance of 0.
5) While the priority queue is not empty:

a. Extract the node with the smallest distance from the priority queue.
b. For each neighbor of the extracted node:
i. Calculate the distance from the start node to the neighbor through the extracted node.
ii. If this distance is smaller than the current distance stored for the neighbor, update the distance. iii. Add the neighbor to the priority queue with the updated distance.

6) Repeat step 4 until all nodes have been processed.
7) The distances stored for each node after the algorithm completes represent the shortest path from the start node to that node. 8) Stop

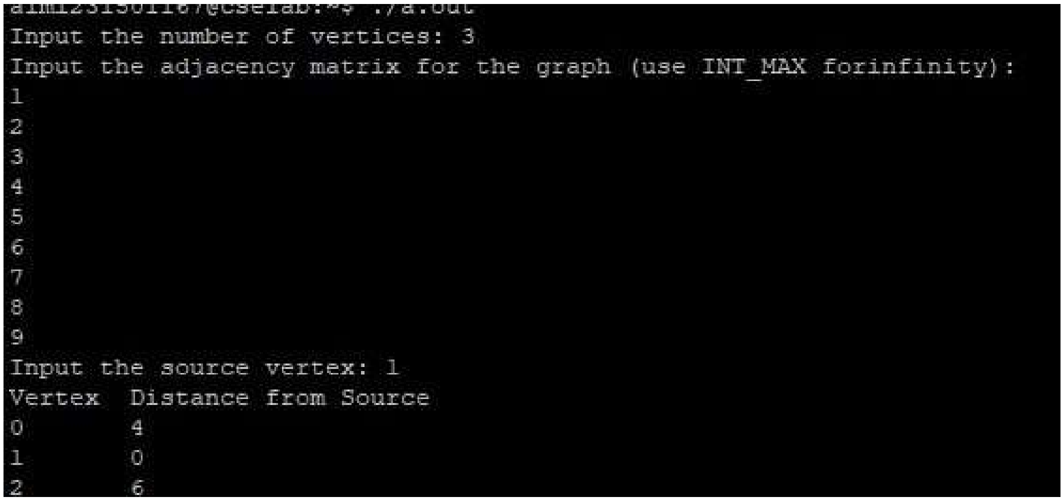
PROGRAM;

```
#include <stdio.h>

#include <limits.h>
```

```
#define MAX_VERTICES 100
int minDistance(int dist[], int sptSet[], int vertices) {
int min = INT_MAX, minIndex; for (int v = 0; v <
vertices; v++) { if (!sptSet[v] && dist[v] < min) { min
= dist[v]; minIndex = v;
}
}
return minIndex;
}
void printSolution(int dist[], int vertices) {
printf("Vertex \tDistance from Source\n");
for (int i = 0; i < vertices; i++) { printf("%d
\t%d\n", i, dist[i]);
}
}
void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int src, int
vertices) { int dist[MAX_VERTICES]; int sptSet[MAX_VERTICES];
for (int i = 0; i < vertices; i++) {
dist[i] = INT_MAX; sptSet[i] =
0;
} dist[src] =
0; for (int
count = 0;
count <
vertices - 1;
count++) {
int u =
minDistance
(dist, sptSet,
vertices);
sptSet[u] =
1;
```

```
for (int v = 0; v < vertices; v++) { if (!sptSet[v] &&
graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v]) { dist[v] = dist[u] + graph[u][v];
}
}
}
printSolution(dist, vertices);
}
int main() { int vertices; printf("Input the
number of vertices: "); scanf("%d", &vertices); if
(vertices <= 0 || vertices > MAX_VERTICES) {
printf("Invalid number of vertices. Exiting...\n");
return 1;
}
int graph[MAX_VERTICES][MAX_VERTICES]; printf("Input the adjacency matrix
for the graph (use INT_MAX forinfinity):\n"); for (int i = 0; i < vertices; i++) {

for (int j = 0; j < vertices; j++) { scanf("%d",
&graph[i][j]);
}
}
int source;
printf("Input the source vertex: ");
scanf("%d", &source); if (source < 0 ||
source >= vertices) { printf("Invalid source
vertex. Exiting...\n"); return 1;
}
dijkstra(graph, source, vertices); return
0;
}
```

OUTPUT:



```
Input the number of vertices: 3
Input the adjacency matrix for the graph (use INT_MAX forinfinity):
1
2
3
4
5
6
7
8
9
Input the source vertex: 1
Vertex  Distance from Source
0       4
1       0
2       6
```

RESULT: Thus, the program was successfully executed.

Data Structures Odyssey: Exploring the Foundations of Computing

Ex. No.:15	**Sorting**	**Date:23/05/2024**

Write a C program to take n numbers and sort the numbers in ascending order. Try to implement the same using the following sorting techniques.

1. **Quick Sort**
2. **Merge Sort**

Algorithm:

1) Start
2) If the array has fewer than two elements, return it as it is already sorted.
3) Divide the array into two halves.
4) Recursively sort the two halves using Merge Sort.
5) Merge the sorted halves into a single sorted array.
6) Choose a sorting algorithm (e.g., Bubble Sort, Merge Sort, Quick Sort).
7) Implement the selected sorting algorithm.
8) Pass the unsorted array to the sorting algorithm.
9) Execute the sorting algorithm to sort the array. Obtain the sorted array as output. 10) stop

PROGRAM:

QUICK SORT: #include

```
<stdio.h>            void
swap(int* a, int* b)
{
int temp = *a; *a
= *b;
*b = temp;
}
int partition(int arr[], int low, int high)
{ int pivot =
arr[low]; int i =
low; int j = high;

while (i < j) { while (arr[i] <= pivot &&
i<= high - 1) { i++;
}
while (arr[j] > pivot && j >= low + 1) {
j--; } if (i < j) {
swap(&arr[i], &arr[j]);
}
}
swap(&arr[low], &arr[j]); return
j;
}

void quickSort(int arr[], int low, int high)
{ if (low < high)
{
```

```
int partitionIndex = partition(arr, low, high);

quickSort(arr, low, partitionIndex - 1); quickSort(arr,

partitionIndex + 1, high);

}

}

int main()

{

int arr[] = { 19, 17, 15, 12, 16, 18, 4, 11, 13 };

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array:'); for (int i = 0; i<n; i++)

{ printf("&%d", arr[i]);

}

quickSort(arr, 0, n 0;

printf("\nSorted array:"); for

(int i = 0; i &lt; n; i++) {

printf("%d", arr[i]);

}


return 0;
}
```

MERGE SORT

```
#include <stdio.h> #include

<stdlib.h> void merge(int arr[], int l,

int m, int r)

{ int i, j,

k;

int n1 = m - l + 1; int n2

= r - m; int L[n1], R[n2];
```

```
for (i = 0; i < n1; i++) L[i]
= arr[l + i]; for (j = 0; j <
n2; j++) R[j] = arr[m + 1
+ j]; i = 0; j = 0; k = l;
while (i < n1 && j < n2) {
if (L[i] <= R[j]) { arr[k] =
L[i]; i++;
} else {
arr[k] = R[j];
j++;
}
k++;
} while (i < n1)
{ arr[k] = L[i];
i++; k++;
}

while (j < n2) {
arr[k] = R[j]; j++;

k++;
}
}
void mergeSort(int arr[], int l, int r)
{ if (l < r) { int m = l + (r -
l) / 2; mergeSort(arr, l,
m); mergeSort(arr, m +
1, r); merge(arr, l, m, r);
}
}
void printArray(int A[], int size)
```

```
{ int i; for (i = 0;
i<size; i++)
printf("%d ", A[i]);

printf("\n");
}
int main()
{
int arr[] = { 12, 11, 13, 5, 6, 7 }; int
arr_size = sizeof(arr) / sizeof(arr[0]);
printf("Given array is \n");
printArray(arr, arr_size);
mergeSort(arr, 0, arr_size - 1);
printf("\nSorted array is \n");
printArray(arr, arr_size); return 0;
}
```

OUTPUT:

RESULT: Thus, the program was successfully executed.

Data Structures Odyssey: Exploring the Foundations of Computing

Ex. No.: 16	Hashing	Date:30/05/2024

Write a C program to create a hash table and perform collision resolution using the following techniques.

(i) Open addressing

(ii) Closed Addressing

(iii) Rehashing

Algorithm:

1. Open Addressing:

1. Compute the hash of the key to be inserted.
2. Check the computed hash index in the hash table.
3. If the slot is empty, insert the key.
4. If the slot is occupied, then it means a collision has occurred. In this case, move to the next slot in the hash table.
5. Repeat the process until an empty slot is found.

2. Closed Addressing (Separate Chaining):

1. Compute the hash of the key to be inserted.
2. Check the computed hash index in the hash table.
3. If the slot is empty, insert the key.
4. If the slot is occupied, then it means a collision has occurred. In this case, add the new key to the linked list at that slot.

3. Rehashing:

1. When the load factor of the hash table reaches a certain threshold (typically > 0.7), create a new hash table of larger size.
2. Compute the hash of each key in the old table.
3. Insert each key into the new table.
4. Delete the old table.

PROGRAM:

A. OPEN ADDRESSING:

```
#include <stdio.h>

#define max 10

int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 }; int
b[10];

void merging(int low, int mid, int high) { int
l1, l2, i;

for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
if(a[l1] <= a[l2]) b[i] = a[l1++]; else
b[i] = a[l2++]; }

while(l1 <= mid) b[i++]
= a[l1++];

while(l2 <= high) b[i++]
= a[l2++];

for(i = low; i <= high; i++)
a[i] = b[i]; }

void sort(int low, int high) { int
mid;

if(low < high) { mid =
(low + high) / 2;
sort(low, mid);
sort(mid+1, high);
merging(low, mid, high);
} else { return;
}
}

int main() { int
i;

printf("List before sorting\n");

for(i = 0; i <= max; i++)
printf("%d ", a[i]);

sort(0, max);
```

```
printf("\nList after sorting\n");

for(i = 0; i <= max; i++)
printf("%d ", a[i]);
}
```

B. CLOSED ADDRESSING;

```
#include <stdio.h>

#define max 10

int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 }; int
b[10];

void merging(int low, int mid, int high) {
int l1, l2, i;

for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
if(a[l1] <= a[l2]) b[i] = a[l1++]; else b[i] = a[l2++];
}
while(l1 <= mid) b[i++]
= a[l1++]; while(l2 <=
high) b[i++] = a[l2++];

for(i = low; i <= high; i++)
a[i] = b[i];
}

void sort(int low, int high) { int
mid;

if(low < high) { mid =
(low + high) / 2;
sort(low, mid);
sort(mid+1, high);
merging(low, mid, high);
} else { return;
}
}

int main() { int
i;

printf("List before sorting\n");

for(i = 0; i <= max; i++)
printf("%d ", a[i]);

sort(0, max);

printf("\nList after sorting\n");
```

```
for(i = 0; i <= max; i++) printf("%d
", a[i]);
}
```

C. REHASHING:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
int key; int value;
struct Node* next;
} Node;

typedef struct HashTable {
int size; int count; Node**
table;
} HashTable;

Node* createNode(int key, int value) { Node*
newNode = (Node*)malloc(sizeof(Node));
newNode->key = key; newNode->value = value;
newNode->next = NULL; return newNode;
}

HashTable* createTable(int size) {
HashTable* newTable = (HashTable*)malloc(sizeof(HashTable));
newTable->size = size; newTable->count = 0;
newTable->table = (Node**)malloc(sizeof(Node*) * size); for
(int i = 0; i < size; i++) {

newTable->table[i] = NULL;
}
return newTable;
}

int hashFunction(int key, int size) { return
key % size;
}

void insert(HashTable* hashTable, int key, int value);

void rehash(HashTable* hashTable) {
int oldSize = hashTable->size; Node**
oldTable = hashTable->table;

// New size is typically a prime number or double the old size int
newSize = oldSize * 2;
hashTable->table = (Node**)malloc(sizeof(Node*) * newSize);
hashTable->size = newSize; hashTable->count = 0;
for (int i = 0; i < newSize; i++) {
hashTable->table[i] = NULL;
}
```

```
for (int i = 0; i < oldSize; i++) {
Node* current = oldTable[i]; while
(current != NULL) {
insert(hashTable, current->key, current->value);
Node* temp = current;

current = current->next; free(temp);
}
}

free(oldTable);
}

void insert(HashTable* hashTable, int key, int value) { if
((float)hashTable->count / hashTable->size >= 0.75) {
rehash(hashTable);
}

int hashIndex = hashFunction(key, hashTable->size);
Node* newNode = createNode(key, value); newNode->next
= hashTable->table[hashIndex]; hashTable-
>table[hashIndex] = newNode; hashTable->count++;
}

int search(HashTable* hashTable, int key) { int
hashIndex = hashFunction(key, hashTable->size);
Node* current = hashTable->table[hashIndex];
while (current != NULL) { if
(current->key == key) {
return current->value;
}
current = current->next;
}

return -1;
}

void delete(HashTable* hashTable, int key) { int
hashIndex = hashFunction(key, hashTable->size);
Node* current = hashTable->table[hashIndex]; Node*
prev = NULL;
while (current != NULL && current->key != key) { prev
= current;
current = current->next;
}
if (current == NULL) { return;
}
if (prev == NULL) {
hashTable->table[hashIndex] = current->next;
} else {
prev->next = current->next;

}
free(current);
```

```
hashTable->count--;
}

void freeTable(HashTable* hashTable) {
for (int i = 0; i < hashTable->size; i++) {
Node* current = hashTable->table[i];
while (current != NULL) { Node* temp =
current; current = current->next;

free(temp);
}
}
free(hashTable->table); free(hashTable);
}

int main() {
HashTable* hashTable = createTable(5);

insert(hashTable, 1, 10); insert(hashTable,
2, 20); insert(hashTable, 3, 30);
insert(hashTable, 4, 40); insert(hashTable,
5, 50);
insert(hashTable, 6, 60); // This should trigger rehashing

printf("Value for key 1: %d\n", search(hashTable, 1)); printf("Value
for key 2: %d\n", search(hashTable, 2)); printf("Value for key 3:
%d\n", search(hashTable, 3)); printf("Value for key 4: %d\n",
search(hashTable, 4)); printf("Value for key 5: %d\n",
search(hashTable, 5)); printf("Value for key 6: %d\n",
search(hashTable, 6));

delete(hashTable, 3);
printf("Value for key 3 after deletion: %d\n", search(hashTable, 3));

freeTable(hashTable);
return 0;

}
```

OUTPUT:

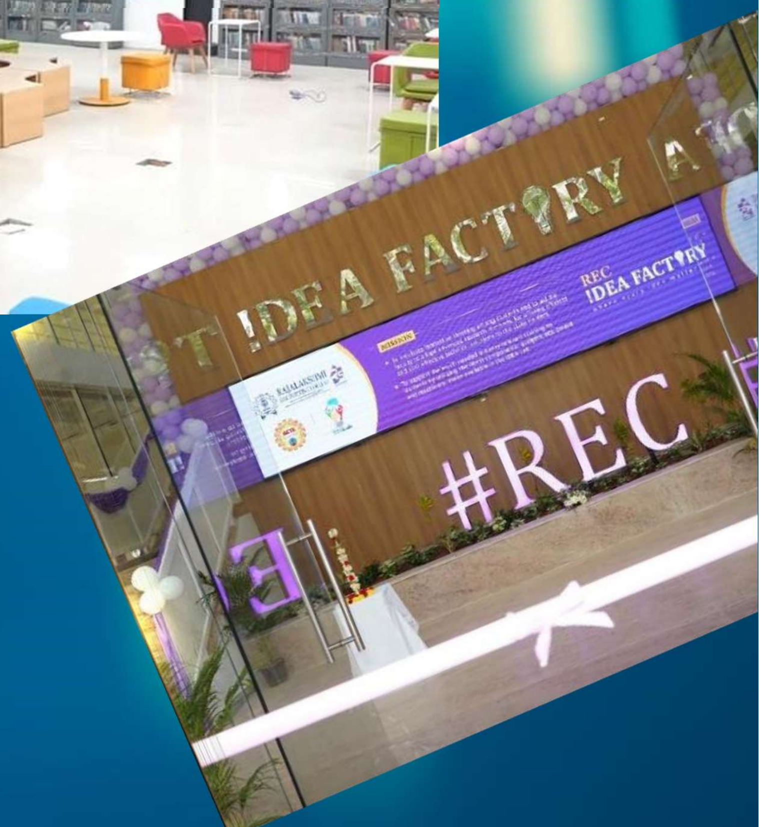
```
aiml231501167@cselab:~$ gcc program16C.c
aiml231501167@cselab:~$ ./a.out
Value for key 1: 10
Value for key 2: 20
Value for key 3: 30
Value for key 4: 40
Value for key 5: 50
Value for key 6: 60
Value for key 3 after deletion: -1
aiml231501167@cselab:~$
```

```
aiml231501167@cselab:~$ ./a.out
Value for key 1: 10
Value for key 2: 20
Value for key 12: 30
Value for key 3: -1
Value for key 2 after deletion: -1
aiml231501167@cselab:~$
```

```
aiml231501167@cselab:~$ gcc program16A.c
aiml231501167@cselab:~$ ./a.out
List before sorting
10 14 19 26 27 31 33 35 42 44 0
List after sorting
0 10 14 19 26 27 31 33 35 42 44 aiml231501167@cselab:~$
```

RESULT: Thus, the program was successfully executed.

Rajalakshmi Institutions



Rajalakshmi Engineering College
Rajalakshmi Nagar Thandalam, Chennai - 602 105.
Phone : +91 -44 -67181111, 67181112
Website : www.rajalakshmi.or g