SHIV NADAR
UNIVERSITY
CHENNAI

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SCHOOL OF ENGINEERING

LABORATORY RECORD

B.TECH
(YEAR : 2023 - 2024)

NAME : SADANAND. VENKATARAMAN

REG. NO. : 23011501011

DEPT. : CSE **SEM. :** II **CLASS & SEC :** M.Tech

www.snuchennai.edu.in

SHIV NADAR
UNIVERSITY
CHENNAI

BONAFIDE CERTIFICATE

Certified that this is the bonafide record of the practical work done in the

[CS5702] ADVANCED DATA STRUCTURES & ALGORITHMS Laboratory by

Name SADANAND . VENKATARAMAN

Register Number 23011501011

Semester II Class & Sec. M.TECH

Branch ARTIFICIAL INTELLIGENCE & DATA SCIENCE

SHIV NADAR UNIVERSITY Chennai

During the Academic year 2023-24.

Faculty **Head of the Department**

Submitted for the END - SEMESTER Practical Examination held at SNU CHENNAI on 08-05-2024.

Internal Examiner **External Examiner**

INDEX

Name : Sadanand. Venkataraman Reg. No. 23011501011

Sem : II Class & Sec : M.Tech

Ex. No.	Date of Expt.	Title of the Experiment	Page No.	Signature of the Faculty	Remarks
1.	31-01-2024.	STACKS.	2.		
2.	31-09-2024.	QUEUES.	4.		
3.	14-02-2024.	Circular QUEUES.	5.		
4.	21-02-2024.	(a) Singly Linked List. (b) Doubly Linked List.	7. 8.		
5.	28-02-2024.	BINARY SEARCH TREES.	10.		S.Sab
6.	06-03-2024	AVL TREES.	12.		
7.	20-03-2024.	HEAPSORT	15.		
8.	03-04-2024.	PRIMS & KRUSKAL'S ALGORITHMS	17.		
9.	17-04-2024.	BELLMAN-FORD ALGORITHM	22.		
10.	24-04-2024.	B-TREES.	25.		

Experiment 1: Stack Implementation using Arrays

Aim

To implement a stack data structure using arrays.

Code

```c
//C Program to implemenet Stacks using Arrays.

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

#define MAX 10  // Defining the maximum size of the stack

int stack[MAX];
int top = -1;

void push(int data) {
    if(top == MAX - 1) {
        printf("Stack overflow\n");
    } else {
        top++;
        stack[top] = data;
        printf("%d pushed to stack\n", data);
    }
}

int pop() {
    if(top == -1) {
        printf("Stack underflow\n");
        return -1;
    } else {
        int data = stack[top];
        top--;
        printf("%d popped from stack\n", data);
        return data;
    }
}

void display() {
    if(top == -1) {
        printf("Stack is empty\n");
    } else {
        printf("Stack elements are:\n");
        for(int i = top; i >= 0; i--) {
            printf("%d\n", stack[i]);
        }
    }
}

int main() {
    push(10);
    push(20);
    push(30);
    display();
    pop();
    display();
```

```
    return 0;
}
```

Result

Program successfully pushes, pops, and displays elements of the stack.

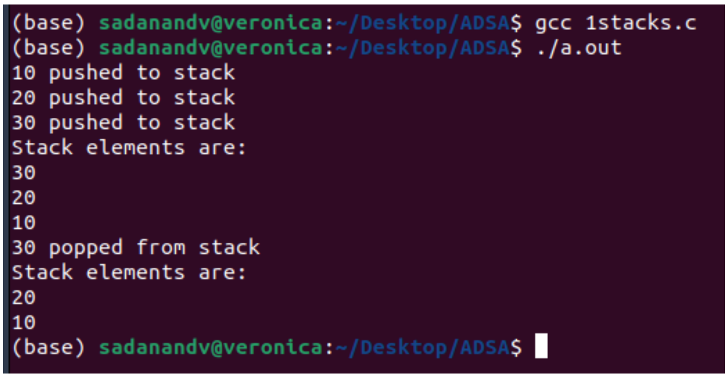

Figure 1: Terminal Output of Experiment 1

Experiment 2: Queue Implementation using Arrays

Aim

To implement a queue data structure using arrays.

Code

```c
#include <stdio.h>
#define MAX 10
int queue[MAX];
int front = -1, rear = -1;
void enqueue(int data) {
    if(rear == MAX - 1)
        printf("Queue overflow\n");
    else {
        if(front == -1) front = 0;
        queue[++rear] = data;
    }
}
int dequeue() {
    if(front == -1)
        printf("Queue underflow\n");
    return queue[front++];
}
void display() {
    for(int i = front; i <= rear; i++)
        printf("%d ", queue[i]);
    printf("\n");
}
int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    dequeue();
    display();
    return 0;
}
```

Result

Program successfully enqueues, dequeues, and displays the queue.

Figure 2: Terminal Output of Experiment 2

Experiment 3: Implementing circular queue using arrays.

Aim

To implement Circular queues using arrays.

Code

```c
//C Program to implement circular queue using arrays.

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

#define MAX 5

int q[MAX];
int f = -1, r = -1;

void enqueue(int data)
{
        if((f == 0 && r == MAX-1)|| (r == (f-1) % (MAX - 1)))
                printf("\nQueue is full");
        else
        {
                if( f == -1)
                        f = r = 0;
                else if( r == MAX - 1 && f != 0)
                        r = 0;
                else
                        r++;
        q[r] = data;
        printf("\n%d Enqueued to Queue at location %d",data, r);
        }
}

int dequeue()
{
        if (f == -1)
        {
                printf("\nQueue is Empty");
                return -1;
        }

        int data = q[f];
        if (f == r)
                f = r = -1;
        else if( f == MAX -1)
                f = 0;
        else
                f++;
        printf("\n%d Dequeues from Queue", data);
        return data;
}

void display()
{
        if(f == -1)
                printf("\nQueue is empty");
```

```
        else
        {
                if(r >=f)
                        for(int i = f; i<=r; i++)
                                printf("%d", q[i]);
                else
                {
                        for(int i = f; i<MAX; i++)
                                printf("%d ", q[i]);
                        for(int i = 0; i<=r; i++)
                                printf("%d ", q[i]);
                }
                printf("\n");
        }
}

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    enqueue(40);
    enqueue(50);
    display(); // Full queue
    dequeue(); // Removing one item
    dequeue(); // Removing another item
    display(); // Showing final queue
    enqueue(60); // Adding new item
    enqueue(70); // Adding another item
    display(); // Final display of circular nature
    return 0;
}
```

Result

Program successfully enqueues, dequeues, and displays the Circular queue.

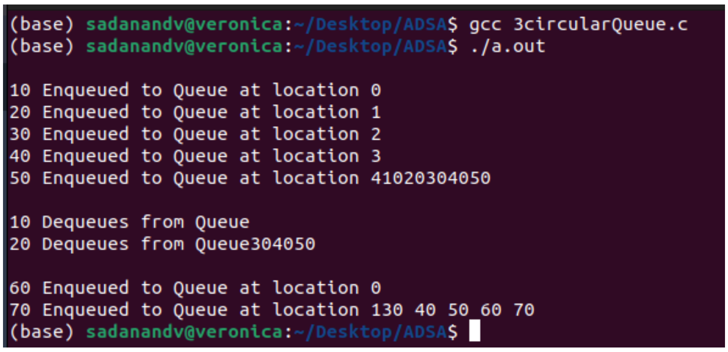

Figure 3: Terminal Output of Experiment 3

Experiment 4: (a) Implementing linked list, (b) Implementing doubly linked list.

Aim

To implement a Singly linked list and doubly linked list data structures.

Code (a)

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

typedef struct Node {
    int data;
    struct Node* next;
} Node;

Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Error creating a new node.\n");
        exit(0);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

void append(Node** head, int data) {
    Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

void deleteNode(Node** head, int key) {
    Node *temp = *head, *prev = NULL;
    if (temp != NULL && temp->data == key) {
        *head = temp->next;
        free(temp);
        return;
    }
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL) return;
    prev->next = temp->next;
    free(temp);
}

void display(Node* head) {
```

```
    Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    Node* head = NULL;
    append(&head, 10);
    append(&head, 20);
    append(&head, 30);
    display(head);
    deleteNode(&head, 20);
    display(head);
    return 0;
}
```

Code (b)

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

typedef struct DNode {
    int data;
    struct DNode* prev;
    struct DNode* next;
} DNode;

DNode* createDNode(int data) {
    DNode* newNode = (DNode*)malloc(sizeof(DNode));
    if (newNode == NULL) {
        printf("Error creating a new node.\n");
        exit(0);
    }
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

void appendDNode(DNode** head, int data) {
    DNode* newNode = createDNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        DNode* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
}

void deleteDNode(DNode** head, int key) {
```

```c
    DNode* temp = *head;
    if (temp != NULL && temp->data == key) {
        *head = temp->next;
        if (*head != NULL) {
            (*head)->prev = NULL;
        }
        free(temp);
        return;
    }
    while (temp != NULL && temp->data != key) {
        temp = temp->next;
    }
    if (temp == NULL) return;
    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    }
    free(temp);
}

void displayD(DNode* head) {
    DNode* temp = head;
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    DNode* head = NULL;
    appendDNode(&head, 10);
    appendDNode(&head, 20);
    appendDNode(&head, 30);
    displayD(head);
    deleteDNode(&head, 20);
    displayD(head);
    return 0;
}
```

Result

Program successfully inserts, deletes and displays elements in the Linked Lists.

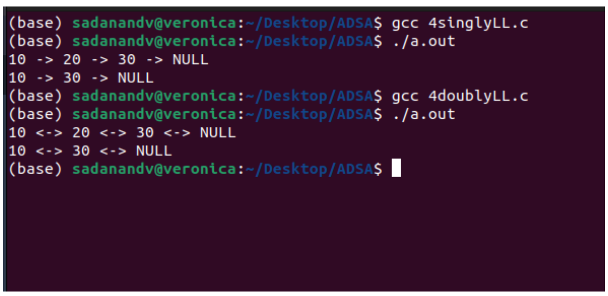

Figure 4: Terminal Output of Experiment 4(a) and 4(b)

Experiment 5: Binary Search Tree

Aim

To implement Binary Search Trees.

Code

```
//C Program to implemenet binary Search Tree.

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

typedef struct Node{
        int data;
        struct Node* l;
        struct Node* r;
}Node;

Node* createNode(int data)
{
        Node* newNode = (Node*)malloc(sizeof(Node));
        if(newNode == NULL)
        {
                printf("\nError creaing new Node");
                exit(0);
        }
        newNode -> data = data;
        newNode -> l = NULL;
        newNode -> r = NULL;
        return newNode;
}

Node* insert(Node* node, int data)
{
        if(node == NULL)
                return createNode(data);
        if(data< node -> data)
                node -> l = insert(node ->l, data);
        else if(data> node -> data)
                node -> r = insert(node -> r, data);

        return node;
}

void inorder(Node *root)
{
        if(root != NULL)
        {
                inorder(root -> l);
                printf("%d ", root -> data);
                inorder(root -> r);
        }
}

Node* search(Node* root, int key)
{
        if(root == NULL || root -> data == key)
```

```c
                return root;
        if(root -> data < key)
                return search(root -> r, key);

        return search(root -> l, key);
}

int main() {
    Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    printf("Inorder traversal of the given tree: \n");
    inorder(root);
    printf("\n");


    int key = 65;
    Node* result = search(root, key);
    if (result != NULL) {
        printf("Element %d found in the BST.\n", key);
    } else {
        printf("Element %d not found in the BST.\n", key);
    }

    return 0;
}
```

Result

Program successfully implements a Binary Search Tree and displays the In-order Traversal of the Tree along with search output.

Figure 5: Terminal Output of Experiment 5

Experiment 6: AVL Tree

Aim

To implement AVL Trees.

Code

```
//C Program to implemenet AVL Search Tree.

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

typedef struct Node{
        int key;
        struct Node* l;
        struct Node* r;
        int height;
}Node;

int height(Node *N)
{
        if(N == NULL)
                return 0;
        return N-> height;
}

int max(int a,int b)
{
        return (a>b)? a:b;
}

Node* newNode(int key)
{
        Node* node = (Node*)malloc(sizeof(Node));
        node ->key = key;
        node -> l = NULL;
        node -> r = NULL;
        node -> height = 1;
        return(node);
}

Node *rRotate(Node *y)
{
        Node *x = y -> l;
        Node *T2 = x -> r;

        x -> r = y;
        y -> l = T2;

        y -> height = max(height(y->l), height(y->r)) + 1;
        x -> height = max(height(x->l), height(x->r)) + 1;

        return x;
}

Node *lRotate(Node *x) {
    Node *y = x->r;
```

```
    Node *T2 = y->l;

    y->l = x;
    x->r = T2;

    x->height = max(height(x->l), height(x->r)) + 1;
    y->height = max(height(y->l), height(y->r)) + 1;

    return y;
}

int getBalance(Node *N)
{
        if(N == NULL)
                return 0;
        return height(N -> l) - height(N->r);
}


Node* insert(Node* node, int key) {
    // 1. Perform the normal BST insertion
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->l = insert(node->l, key);
    else if (key > node->key)
        node->r = insert(node->r, key);
    else // Equal keys are not allowed in the BST
        return node;

    // 2. Update height of this ancestor node
    node->height = 1 + max(height(node->l), height(node->r));

    // 3. Get the balance factor of this ancestor node to check whether
    // this node became unbalanced
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // l l Case
    if (balance > 1 && key < node->l->key)
        return rRotate(node);

    // r r Case
    if (balance < -1 && key > node->r->key)
        return lRotate(node);

    // l r Case
    if (balance > 1 && key > node->l->key) {
        node->l = lRotate(node->l);
        return rRotate(node);
    }

    // r l Case
    if (balance < -1 && key < node->r->key) {
        node->r = rRotate(node->r);
        return lRotate(node);
```

```
    }

    // return the (unchanged) node pointer
    return node;
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(Node *root) {
    if(root != NULL) {
        printf("%d ", root->key);
        preOrder(root->l);
        preOrder(root->r);
    }
}

int main() {
    Node *root = NULL;

    /* Constructing tree given in the above figure */
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    printf("Preorder traversal of the constructed AVL tree is \n");
    preOrder(root);

    return 0;
}
```

Result

Program successfully implements an AVLTree and displays the Pre-order Traversal of the Tree.

Figure 6: Terminal Output of Experiment 6

Experiment 7: Heap Sort

Aim

To implement Heap Sort using min/max heaps.

Code

```
#include <stdio.h>

// Function to swap two elements
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void maxHeapify(int arr[], int n, int i) {
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left = 2*i + 1
    int right = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not root
    if (largest != i) {
        swap(&arr[i], &arr[largest]);

        // Recursively heapify the affected sub-tree
        maxHeapify(arr, n, largest);
    }
}

// Function to build a max-heap from an array
void buildMaxHeap(int arr[], int n) {
    // Index of last non-leaf node
    int startIdx = (n / 2) - 1;

    // Perform reverse level order traversal from last non-leaf node and heapify each n
    for (int i = startIdx; i >= 0; i--) {
        maxHeapify(arr, n, i);
    }
}

// Main function to do heap sort
void heapSort(int arr[], int n) {
    // Build heap (rearrange array)
    buildMaxHeap(arr, n);

    // One by one extract an element from heap
```

```
    for (int i = n - 1; i >= 0; i--) {
        // Move current root to end
        swap(&arr[0], &arr[i]);

        // call max heapify on the reduced heap
        maxHeapify(arr, i, 0);
    }
}

// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; ++i)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program
int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array is \n");
    printArray(arr, n);

    heapSort(arr, n);

    printf("Sorted array is \n");
    printArray(arr, n);
    return 0;
}
```

Result

Program successfully implements an AVLTree and displays the Pre-order Traversal of the Tree.

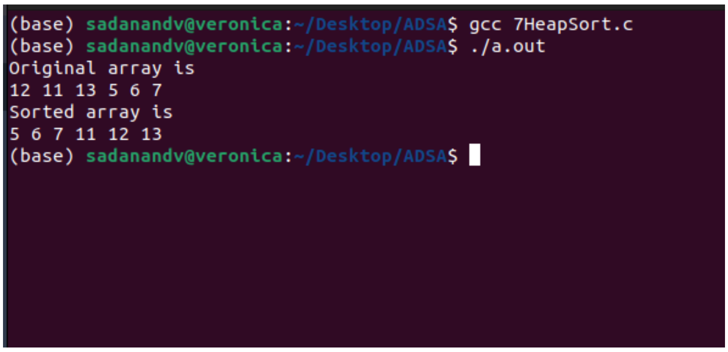

Figure 7: Terminal Output of Experiment 7

Experiment 8: Prim's and Kruskal's Algorithms.

Aim

To implement Prim's and Kruskals' algorithm for MST.

Code (a) Prim's Algorithm

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

#define V 5  // Number of vertices in the graph

// A utility function to find the vertex with minimum key value, from the set of vertice
int minKey(int key[], int mstSet[]) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == 0 && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V]) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}

// Function to construct and print MST for a graph represented using adjacency matrix r
void primMST(int graph[V][V]) {
    int parent[V]; // Array to store constructed MST
    int key[V];    // Key values used to pick minimum weight edge in cut
    int mstSet[V]; // To represent set of vertices not yet included in MST

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = 0;

    // Always include first 1st vertex in MST.
    key[0] = 0;      // Make key 0 so that this vertex is picked as first vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = 1;
        for (int v = 0; v < V; v++)
          if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v])
            parent[v] = u, key[v] = graph[u][v];
    }

    printMST(parent, graph);
}

// Driver program to test above functions
```

```
int main() {
    /* Let us create the following graph
          2    3
      (0)--(1)--(2)
       |   / \   |
      6| 8/   \5 |7
       | /     \ |
      (3)-------(4)
            9
    */
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };

    primMST(graph);

    return 0;
}
```

Code (b) Kruskal's Algorithm

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

// a structure to represent a weighted edge in graph
struct Edge {
    int src, dest, weight;
};

// a structure to represent a connected, undirected and weighted graph
struct Graph {
    int V, E;
    struct Edge* edge;
};

struct Graph* createGraph(int V, int E) {
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );
    return graph;
}

// A structure to represent a subset for union-find
struct subset {
    int parent;
    int rank;
};

// Find set of an element i (uses path compression technique)
int find(struct subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}
```

```c
// A function that does union of two sets of x and y (uses union by rank)
void Union(struct subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
int compare(const void* a, const void* b) {
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// Function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph) {
    int V = graph->V;
    struct Edge result[V];  // Tnis will store the resultant MST
    int e = 0;  // An index variable, used for result[]
    int i = 0;  // An index variable, used for sorted edges

    // Step 1:  Sort all the edges in non-decreasing order of their weight
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), compare);

    // Allocate memory for creating V ssubsets
    struct subset *subsets = (struct subset*) malloc( V * sizeof(struct subset) );

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1 && i < graph->E) {
        // Step 2: Pick the smallest edge. And increment the index for next iteration
        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge does't cause cycle, include it in result
        // and increment the index of result for next edge
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
        // Else discard the next_edge
    }
```

```
    // print the contents of result[] to display the built MST
    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
    return;
}

// Driver program to test above functions
int main() {
    /* Let's create following weighted graph
             10
        0--------1
        |  \     |
       6|   5\   |15
        |      \ |
        2--------3
             4        */
    int V = 4;  // Number of vertices in graph
    int E = 5;  // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    KruskalMST(graph);

    return 0;
}
```

Result

Program successfully implements both Prim's and Kruskal's Algorithms for the following Graphs:

(a) Prim's:

```
      2    3
 (0)--(1)--(2)
  |   / \   |
 6| 8/   \5 |7
  | /     \ |
 (3)-------(4)
       9
```

(b) Kruskal's:

```
        10
   0--------1
   |  \     |
  6|   5\   |15
   |     \  |
   2--------3
       4
```

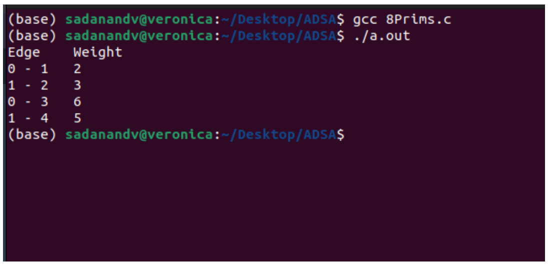

Figure 8: Prim's Algorithm Output

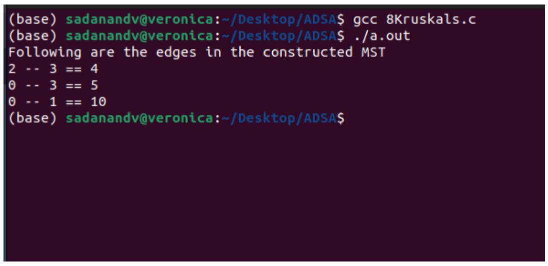

Figure 9: Kruskal's Algorithm Output

Experiment 9: Bellman-Ford Algorithm

Aim

To implement Bellman ford algorithm to find shortest path in graphs.

Code

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

// Define "INFINITY" (you may choose an appropriately large value depending on your con
#define INFINITY 999999
#define MAX_VERTICES 1000

// a structure to represent a weighted edge in graph
struct Edge {
    int src, dest, weight;
};

// a structure to represent a connected, directed and weighted graph
struct Graph {
    int V; // Number of vertices in the graph
    int E; // Number of edges in the graph
    struct Edge* edge; // array of edges
};

// Function to create a graph with V vertices and E edges
struct Graph* createGraph(int V, int E) {
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    graph->edge = (struct Edge*) malloc(graph->E * sizeof(struct Edge));
    return graph;
}

// A utility function used to print the solution
void printArr(int dist[], int n) {
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to all other vertices using
void BellmanFord(struct Graph* graph, int src) {
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src to all other vertices as INFINITE
    for (int i = 0; i < V; i++)
        dist[i] = INFINITY;
    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times. A simple shortest path from src to any oth
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
```

```c
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INFINITY && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    // Step 3: Check for negative-weight cycles. The above step guarantees shortest dist
    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INFINITY && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle\n");
            return; // If negative cycle is detected, we simply return
        }
    }

    printArr(dist, V);
}

// Driver code to test above functions
int main() {
    int V = 5; // Number of vertices in graph
    int E = 8; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = -1;

    // add edge 0-2 (or A-C)
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 4;

    // add edge 1-2 (or B-C)
    graph->edge[2].src = 1;
    graph->edge[2].dest = 2;
    graph->edge[2].weight = 3;

    // add edge 1-3 (or B-D)
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 2;

    // add edge 1-4 (or B-E)
    graph->edge[4].src = 1;
    graph->edge[4].dest = 4;
    graph->edge[4].weight = 2;

    // add edge 3-2 (or D-C)
    graph->edge[5].src = 3;
    graph->edge[5].dest = 2;
    graph->edge[5].weight = 5;

    // add edge 3-1 (or D-B)
```

```
    graph->edge[6].src = 3;
    graph->edge[6].dest = 1;
    graph->edge[6].weight = 1;

    // add edge 4-3 (or E-D)
    graph->edge[7].src = 4;
    graph->edge[7].dest = 3;
    graph->edge[7].weight = -3;

    BellmanFord(graph, 0);

    return 0;
}
```

Result

Program successfully implements an AVLTree and displays the Pre-order Traversal of the Tree.

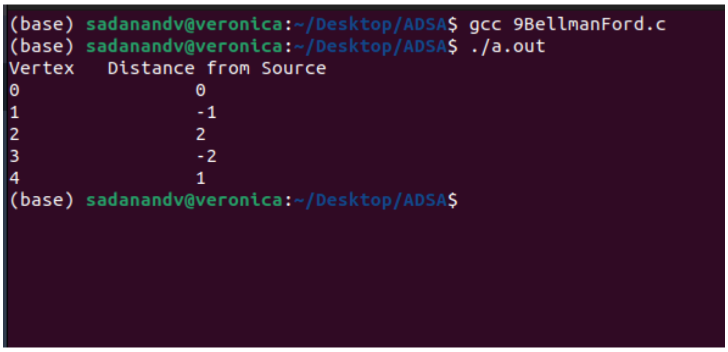

Figure 10: Terminal Output of Experiment 9

Experiment 10: B-Trees

Aim

To implement B-Trees Data Structure.

Code

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

// A BTree node
typedef struct BTreeNode {
    int *keys;            // An array of keys
    int t;                // Minimum degree (defines the range for number of keys)
    struct BTreeNode **C; // An array of child pointers
    int n;                // Current number of keys
    int leaf;             // Is true when node is leaf. Otherwise false
} BTreeNode;

// A utility function to create a new B-Tree node
BTreeNode* createNode(int t, int leaf) {
    BTreeNode* newNode = (BTreeNode*)malloc(sizeof(BTreeNode));
    newNode->t = t;
    newNode->leaf = leaf;
    newNode->keys = (int*)malloc((2*t-1) * sizeof(int));
    newNode->C = (BTreeNode**)malloc(2*t * sizeof(BTreeNode*));
    newNode->n = 0;
    return newNode;
}

// A utility function to insert a new key in this node
void insertNonFull(BTreeNode* node, int k) {
    int i = node->n - 1; // Initialize i as index of right-most element

    // If this is a leaf node
    if (node->leaf) {
        // Find the location of new key to be inserted and move all greater keys to one
        while (i >= 0 && node->keys[i] > k) {
            node->keys[i+1] = node->keys[i];
            i--;
        }

        // Insert the new key at found location
        node->keys[i+1] = k;
        node->n = node->n + 1;
    } else { // If this node is not leaf
        // Find the child which is going to have the new key
        while (i >= 0 && node->keys[i] > k)
            i--;

        // See if the found child is full
        if (node->C[i+1]->n == 2*node->t-1) {
            splitChild(node, i+1, node->C[i+1]);

            // After split, the middle key of C[i] goes up and C[i] is split into two. S
            if (node->keys[i+1] < k)
                i++;
```

```
        }
        insertNonFull(node->C[i+1], k);
    }
}

// A utility function to split the child y of this node. i is index of y in child array
void splitChild(BTreeNode* parent, int i, BTreeNode* y) {
    // Create a new node which is going to store (t-1) keys of y
    BTreeNode* z = createNode(y->t, y->leaf);
    z->n = parent->t - 1;

    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < parent->t-1; j++)
        z->keys[j] = y->keys[j + parent->t];

    // Copy the last t children of y to z
    if (!y->leaf) {
        for (int j = 0; j < parent->t; j++)
            z->C[j] = y->C[j + parent->t];
    }

    // Reduce the number of keys in y
    y->n = parent->t - 1;

    // Since this node is going to have a new child, create space of new child
    for (int j = parent->n; j >= i+1; j--)
        parent->C[j+1] = parent->C[j];

    // Link the new child to this node
    parent->C[i+1] = z;

    // A key of y will move to this node. Find the location of new key and move all gre
    for (int j = parent->n-1; j >= i; j--)
        parent->keys[j+1] = parent->keys[j];

    // Copy the middle key of y to this node
    parent->keys[i] = y->keys[parent->t-1];
    parent->n = parent->n + 1;
}

// Function to traverse all nodes in a subtree rooted with this node
void traverse(BTreeNode* root) {
    int i;
    for (i = 0; i < root->n; i++) {
        // If this is not leaf, then before printing key[i], traverse the subtree rootec
        if (!root->leaf)
            traverse(root->C[i]);
        printf("%d ", root->keys[i]);
    }

    // Print the subtree rooted with last child
    if (!root->leaf)
        traverse(root->C[i]);
}

// The main function that inserts a new key in this B-Tree
void insert(BTreeNode** rootRef, int k, int t) {
    BTreeNode* root = *rootRef;
```

```
    // If tree is empty
    if (root == NULL) {
        root = createNode(t, 1);
        root->keys[0] = k;  // Insert key
        root->n = 1;  // Update number of keys in root
        *rootRef = root;
    } else { // If tree is not empty
        // If root is full, then tree grows in height
        if (root->n == 2*t-1) {
            BTreeNode* s = createNode(t, 0);

            // Make old root as child of new root
            s->C[0] = root;

            // Split the old root and move 1 key to the new root
            splitChild(s, 0, root);

            // New root has two children now. Decide which of the two children is going
            int i = 0;
            if (s->keys[0] < k)
                i++;
            insertNonFull(s->C[i], k);

            // Change root
            *rootRef = s;
        } else  // If root is not full, call insertNonFull for root
            insertNonFull(root, k);
    }
}

// Driver program to test above functions
int main() {
    BTreeNode* root = NULL;
    int t = 3; // A B-Tree of minimum degree 3
    insert(&root, 10, t);
    insert(&root, 20, t);
    insert(&root, 5, t);
    insert(&root, 6, t);
    insert(&root, 12, t);
    insert(&root, 30, t);
    insert(&root, 7, t);
    insert(&root, 17, t);

    printf("Traversal of the constucted B-tree is:\n");
    traverse(root);

    return 0;
}
```

Result

The B-tree operations correctly handle multiple split operations and maintain the tree's properties.After the sequence of insertions, the B-Tree with minimum degree 3 is structured as follows

```
      [10, 20]
     /    |    \
[5, 6, 7]  [12, 17]  [30]
```

Figure 11: Terminal Output of Experiment 10