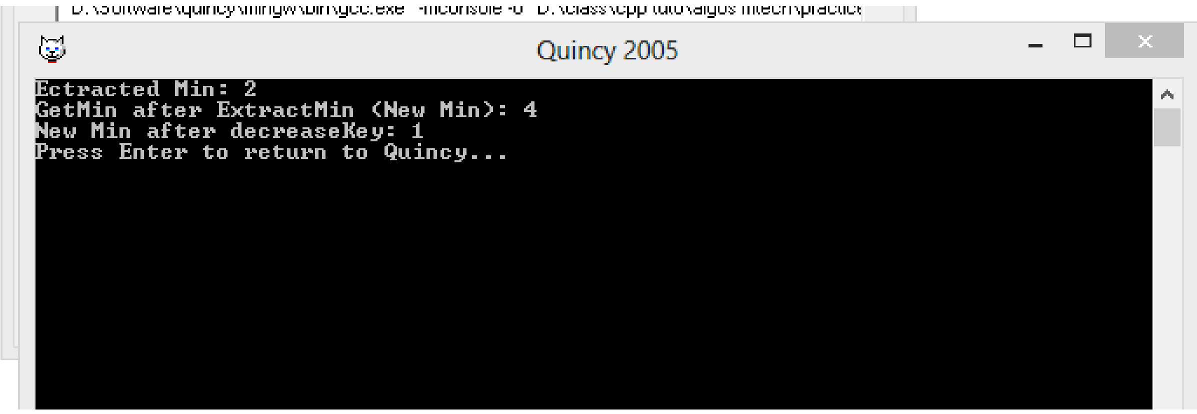
Practical-3: Implementation of operations on binary heaps.

```
#include<iostream.h>
void swap(int *x, int *y);

class MinHeap
{
	int *harr;
	int capacity;
	int heap_size;
public:
	MinHeap(int capacity);
	void MinHeapify(int );

	int parent(int i) { return (i-1)/2; }
	int left(int i) { return (2*i + 1); }
	int right(int i) { return (2*i + 2); }
	int extractMin();
	void decreaseKey(int i, int new_val);
	int getMin() { return harr[0]; }
	void deleteKey(int i);
	void insertKey(int k);
};
MinHeap::MinHeap(int cap)
{
	heap_size = 0;
	capacity = cap;
	harr = new int[cap];
}
void MinHeap::insertKey(int k)
{
	if (heap_size == capacity)
	{
		cout << "\nOverflow: Could not insertKey\n";
		return;
	}
	heap_size++;
	int i = heap_size - 1;
	harr[i] = k;
	while (i != 0 && harr[parent(i)] > harr[i])
	{
	swap(&harr[i], &harr[parent(i)]);
	i = parent(i);
	}
}
void MinHeap::decreaseKey(int i, int new_val)
{
```

```
	harr[i] = new_val;
	while (i != 0 && harr[parent(i)] > harr[i])
	{
	swap(&harr[i], &harr[parent(i)]);
	i = parent(i);
	}
}

int MinHeap::extractMin()
{
	if (heap_size <= 0)
		return INT_MAX;
	if (heap_size == 1)
	{
		heap_size--;
		return harr[0];
	}

	int root = harr[0];
	harr[0] = harr[heap_size-1];
	heap_size--;
	MinHeapify(0);

	return root;
}


void MinHeap::deleteKey(int i)
{
	decreaseKey(i, INT_MIN);
	extractMin();
}

void MinHeap::MinHeapify(int i)
{
	int l = left(i);
	int r = right(i);
	int smallest = i;
	if (l < heap_size && harr[l] < harr[i])
		smallest = l;
	if (r < heap_size && harr[r] < harr[smallest])
		smallest = r;
	if (smallest != i)
	{
		swap(&harr[i], &harr[smallest]);
		MinHeapify(smallest);
	}
```

```
}
void swap(int *x, int *y)
{
	int temp = *x;
	*x = *y;
	*y = temp;
}
int main()
{
	MinHeap h(11);
	h.insertKey(3);
	h.insertKey(2);
	h.deleteKey(1);
	h.insertKey(15);
	h.insertKey(5);
	h.insertKey(4);
	h.insertKey(45);
	cout <<"Ectracted Min: "<< h.extractMin() << "\n";
	cout << "GetMin after ExtractMin (New Min): "<< h.getMin() << "\n";
	h.decreaseKey(2, 1);
	cout << "New Min after decreaseKey: "<<h.getMin();
	return 0;
}
```

Output:



Practical-4: Implementation of operations on Fibonacci heaps.

```
#include <cmath>
#include <cstdlib>
#include <iostream>
struct node {
	node* parent;
	node* child;
	node* left;
	node* right;
	int key;
	int degree;
	char mark;
	char c;
};
struct node* mini = NULL;
int no_of_nodes = 0;
void insertion(int val)
{
	struct node* new_node = (struct node*)malloc(sizeof(struct node));
	new_node->key = val;
	new_node->degree = 0;
	new_node->mark = 'W';
	new_node->c = 'N';
	new_node->parent = NULL;
	new_node->child = NULL;
	new_node->left = new_node;
	new_node->right = new_node;
	if (mini != NULL) {
		(mini->left)->right = new_node;
		new_node->right = mini;
		new_node->left = mini->left;
		mini->left = new_node;
		if (new_node->key < mini->key)
			mini = new_node;
	}
	else {
		mini = new_node;
	}
	no_of_nodes++;
}
void Fibonnaci_link(struct node* ptr2, struct node* ptr1)
{
	(ptr2->left)->right = ptr2->right;
	(ptr2->right)->left = ptr2->left;
	if (ptr1->right == ptr1)
		mini = ptr1;
```

```
        ptr2->left = ptr2;
        ptr2->right = ptr2;
        ptr2->parent = ptr1;
        if (ptr1->child == NULL)
                ptr1->child = ptr2;
        ptr2->right = ptr1->child;
        ptr2->left = (ptr1->child)->left;
        ((ptr1->child)->left)->right = ptr2;
        (ptr1->child)->left = ptr2;
        if (ptr2->key < (ptr1->child)->key)
                ptr1->child = ptr2;
        ptr1->degree++;
}
void Consolidate()
{
        int temp1;
        float temp2 = (log(no_of_nodes)) / (log(2));
        int temp3 = temp2;
        struct node* arr[temp3];
        for (int i = 0; i <= temp3; i++)
                arr[i] = NULL;
        node* ptr1 = mini;
        node* ptr2;
        node* ptr3;
        node* ptr4 = ptr1;
        do {
                ptr4 = ptr4->right;
                temp1 = ptr1->degree;
                while (arr[temp1] != NULL) {
                        ptr2 = arr[temp1];
                        if (ptr1->key > ptr2->key) {
                                ptr3 = ptr1;
                                ptr1 = ptr2;
                                ptr2 = ptr3;
                        }
                        if (ptr2 == mini)
                                mini = ptr1;
                        Fibonnaci_link(ptr2, ptr1);
                        if (ptr1->right == ptr1)
                                mini = ptr1;
                        arr[temp1] = NULL;
                        temp1++;
                }
                arr[temp1] = ptr1;
                ptr1 = ptr1->right;
        } while (ptr1 != mini);
        mini = NULL;
```

```
		for (int j = 0; j <= temp3; j++) {
			if (arr[j] != NULL) {
				arr[j]->left = arr[j];
				arr[j]->right = arr[j];
				if (mini != NULL) {
					(mini->left)->right = arr[j];
					arr[j]->right = mini;
					arr[j]->left = mini->left;
					mini->left = arr[j];
					if (arr[j]->key < mini->key)
						mini = arr[j];
				}
				else {
					mini = arr[j];
				}
				if (mini == NULL)
					mini = arr[j];
				else if (arr[j]->key < mini->key)
					mini = arr[j];
			}
		}
	}
	void Extract_min()
	{
		if (mini == NULL)
			cout << "The heap is empty" << endl;
		else {
			node* temp = mini;
			node* pntr;
			pntr = temp;
			node* x = NULL;
			if (temp->child != NULL) {

				x = temp->child;
				do {
					pntr = x->right;
					(mini->left)->right = x;
					x->right = mini;
					x->left = mini->left;
					mini->left = x;
					if (x->key < mini->key)
						mini = x;
					x->parent = NULL;
					x = pntr;
				} while (pntr != temp->child);
			}
			(temp->left)->right = temp->right;
```

```
		(temp->right)->left = temp->left;
		mini = temp->right;
		if (temp == temp->right && temp->child == NULL)
			mini = NULL;
		else {
			mini = temp->right;
			Consolidate();
		}
		no_of_nodes--;
	}
}

void Cut(struct node* found, struct node* temp)
{
	if (found == found->right)
		temp->child = NULL;

	(found->left)->right = found->right;
	(found->right)->left = found->left;
	if (found == temp->child)
		temp->child = found->right;

	temp->degree = temp->degree - 1;
	found->right = found;
	found->left = found;
	(mini->left)->right = found;
	found->right = mini;
	found->left = mini->left;
	mini->left = found;
	found->parent = NULL;
	found->mark = 'B';
}

void Cascase_cut(struct node* temp)
{
	node* ptr5 = temp->parent;
	if (ptr5 != NULL) {
		if (temp->mark == 'W') {
			temp->mark = 'B';
		}
		else {
			Cut(temp, ptr5);
			Cascase_cut(ptr5);
		}
	}
}
void Decrease_key(struct node* found, int val)
```

```
{
	if (mini == NULL)
		cout << "The Heap is Empty" << endl;

	if (found == NULL)
		cout << "Node not found in the Heap" << endl;

	found->key = val;

	struct node* temp = found->parent;
	if (temp != NULL && found->key < temp->key) {
		Cut(found, temp);
		Cascase_cut(temp);
	}
	if (found->key < mini->key)
		mini = found;
}
void Find(struct node* mini, int old_val, int val)
{
	struct node* found = NULL;
	node* temp5 = mini;
	temp5->c = 'Y';
	node* found_ptr = NULL;
	if (temp5->key == old_val) {
		found_ptr = temp5;
		temp5->c = 'N';
		found = found_ptr;
		Decrease_key(found, val);
	}
	if (found_ptr == NULL) {
		if (temp5->child != NULL)
			Find(temp5->child, old_val, val);
		if ((temp5->right)->c != 'Y')
			Find(temp5->right, old_val, val);
	}
	temp5->c = 'N';
	found = found_ptr;
}

void Deletion(int val)
{
	if (mini == NULL)
		cout << "The heap is empty" << endl;
	else {

		Find(mini, val, 0);
```

```
		Extract_min();
		cout << "Key Deleted" << endl;
	}
}

void display()
{
	node* ptr = mini;
	if (ptr == NULL)
		cout << "The Heap is Empty" << endl;

	else {
		cout << "The root nodes of Heap are: " << endl;
		do {
			cout << ptr->key;
			ptr = ptr->right;
			if (ptr != mini) {
				cout << "-->";
			}
		} while (ptr != mini && ptr->right != NULL);
		cout << endl
			<< "The heap has " << no_of_nodes << " nodes" << endl
			<< endl;
	}
}

int main()
{
	cout << "Creating an initial heap" << endl;
	insertion(5);
	insertion(2);
	insertion(8);

	display();

	cout << "Extracting min" << endl;
	Extract_min();
	display();

	cout << "Decrease value of 8 to 7" << endl;
	Find(mini, 8, 7);
	display();

	cout << "Delete the node 7" << endl;
	Deletion(7);
	display();
```

```
	return 0;
}
```

Output:

```
Creating an initial heap
The root nodes of Heap are:
2-->5-->8
The heap has 3 nodes

Extracting min
The root nodes of Heap are:
5
The heap has 2 nodes

Decrease value of 8 to 7
The root nodes of Heap are:
5
The heap has 2 nodes

Delete the node 7
Key Deleted
The root nodes of Heap are:
5
The heap has 1 nodes
```

Practical-5: Implementation on operations on B-Trees.

```
#include<iostream>
using namespace std;

class BTreeNode
{
	int *keys;
	int t;
	BTreeNode **C;
	int n;
	bool leaf;

public:

	BTreeNode(int _t, bool _leaf);
	void traverse();

	BTreeNode *search(int k);
	int findKey(int k);

	void insertNonFull(int k);

	void splitChild(int i, BTreeNode *y);

	void remove(int k);

	void removeFromLeaf(int idx);

	void removeFromNonLeaf(int idx);

	int getPred(int idx);

	int getSucc(int idx);

	void fill(int idx);

	void borrowFromPrev(int idx);

	void borrowFromNext(int idx);

	void merge(int idx);

	friend class BTree;
};

class BTree
{
```

```
	BTreeNode *root;
	int t;
public:

	BTree(int _t)
	{
		root = NULL;
		t = _t;
	}

	void traverse()
	{
		if (root != NULL) root->traverse();
	}

	BTreeNode* search(int k)
	{
		return (root == NULL)? NULL : root->search(k);
	}

	void insert(int k);
	void remove(int k);

};

BTreeNode::BTreeNode(int t1, bool leaf1)
{
	t = t1;
	leaf = leaf1;
	keys = new int[2*t-1];
	C = new BTreeNode *[2*t];
	n = 0;
}

int BTreeNode::findKey(int k)
{
	int idx=0;
	while (idx<n && keys[idx] < k)
		++idx;
	return idx;
}
void BTreeNode::remove(int k)
{
	int idx = findKey(k);
	if (idx < n && keys[idx] == k)
	{
```

```
		if (leaf)
			removeFromLeaf(idx);
		else
			removeFromNonLeaf(idx);
	}
	else
	{

		if (leaf)
		{
			cout << "The key "<< k <<" is does not exist in the tree\n";
			return;
		}
		bool flag = ( (idx==n)? true : false );

		if (C[idx]->n < t)
			fill(idx);

		if (flag && idx > n)
			C[idx-1]->remove(k);
		else
			C[idx]->remove(k);
	}
	return;
}

void BTreeNode::removeFromLeaf (int idx)
{

	for (int i=idx+1; i<n; ++i)
		keys[i-1] = keys[i];

	n--;

	return;
}

void BTreeNode::removeFromNonLeaf(int idx)
{

	int k = keys[idx];

	if (C[idx]->n >= t)
	{
		int pred = getPred(idx);
		keys[idx] = pred;
		C[idx]->remove(pred);
```

```
	}

	else if (C[idx+1]->n >= t)
	{
		int succ = getSucc(idx);
		keys[idx] = succ;
		C[idx+1]->remove(succ);
	}
	else
	{
		merge(idx);
		C[idx]->remove(k);
	}
	return;
}

int BTreeNode::getPred(int idx)
{
	BTreeNode *cur=C[idx];
	while (!cur->leaf)
		cur = cur->C[cur->n];

	return cur->keys[cur->n-1];
}

int BTreeNode::getSucc(int idx)
{

	BTreeNode *cur = C[idx+1];
	while (!cur->leaf)
		cur = cur->C[0];

	return cur->keys[0];
}

void BTreeNode::fill(int idx)
{
	if (idx!=0 && C[idx-1]->n>=t)
		borrowFromPrev(idx);

	else if (idx!=n && C[idx+1]->n>=t)
		borrowFromNext(idx);

	else
	{
		if (idx != n)
			merge(idx);
```

```
		else
			merge(idx-1);
	}
	return;
}
void BTreeNode::borrowFromPrev(int idx)
{

	BTreeNode *child=C[idx];
	BTreeNode *sibling=C[idx-1];

	for (int i=child->n-1; i>=0; --i)
		child->keys[i+1] = child->keys[i];

	if (!child->leaf)
	{
		for(int i=child->n; i>=0; --i)
			child->C[i+1] = child->C[i];
	}

	child->keys[0] = keys[idx-1];

	if(!child->leaf)
		child->C[0] = sibling->C[sibling->n];

	keys[idx-1] = sibling->keys[sibling->n-1];

	child->n += 1;
	sibling->n -= 1;

	return;
}

void BTreeNode::borrowFromNext(int idx)
{

	BTreeNode *child=C[idx];
	BTreeNode *sibling=C[idx+1];

	child->keys[(child->n)] = keys[idx];

	if (!(child->leaf))
		child->C[(child->n)+1] = sibling->C[0];

	keys[idx] = sibling->keys[0];

	for (int i=1; i<sibling->n; ++i)
```

```
            sibling->keys[i-1] = sibling->keys[i];

        if (!sibling->leaf)
        {
            for(int i=1; i<=sibling->n; ++i)
                sibling->C[i-1] = sibling->C[i];
        }

        child->n += 1;
        sibling->n -= 1;

        return;
}

void BTreeNode::merge(int idx)
{
        BTreeNode *child = C[idx];
        BTreeNode *sibling = C[idx+1];

        child->keys[t-1] = keys[idx];

        for (int i=0; i<sibling->n; ++i)
            child->keys[i+t] = sibling->keys[i];

        if (!child->leaf)
        {
            for(int i=0; i<=sibling->n; ++i)
                child->C[i+t] = sibling->C[i];
        }
        for (int i=idx+1; i<n; ++i)
            keys[i-1] = keys[i];

        for (int i=idx+2; i<=n; ++i)
            C[i-1] = C[i];

        child->n += sibling->n+1;
        n--;
        delete(sibling);
        return;
}
void BTree::insert(int k)
{
        if (root == NULL)
        {
            root = new BTreeNode(t, true);
            root->keys[0] = k;
            root->n = 1;
```

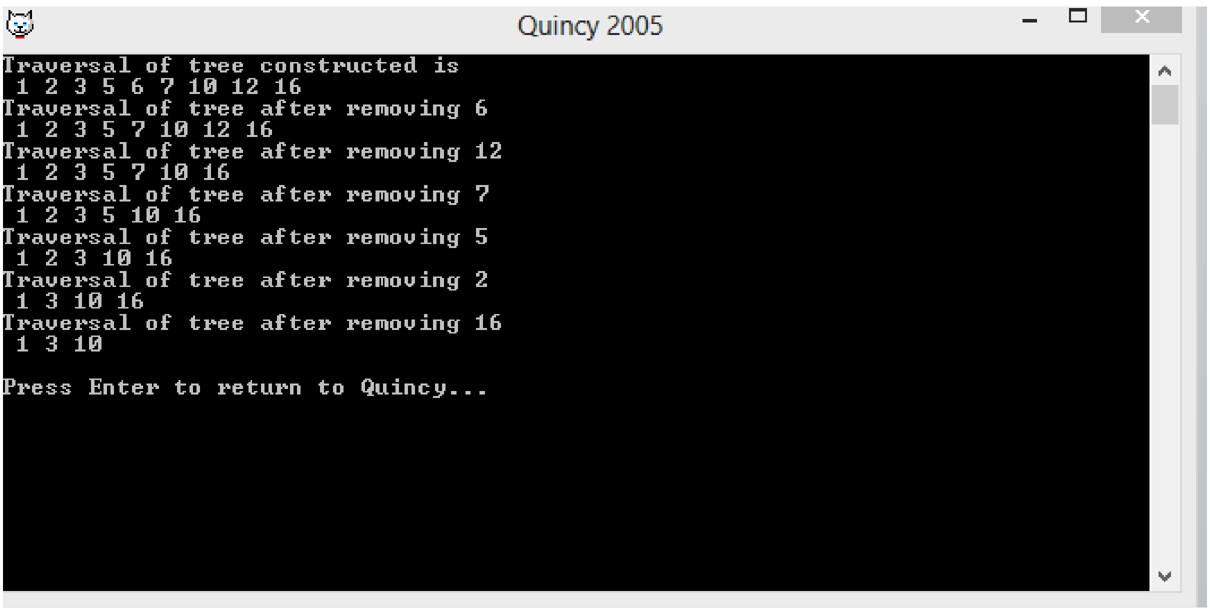
```
        }
        else
        {
                if (root->n == 2*t-1)
                {
                        BTreeNode *s = new BTreeNode(t, false);

                        s->C[0] = root;
                        s->splitChild(0, root);
                        int i = 0;
                        if (s->keys[0] < k)
                                i++;
                        s->C[i]->insertNonFull(k);
                        root = s;
                }
                else
                        root->insertNonFull(k);
        }
}
void BTreeNode::insertNonFull(int k)
{
        int i = n-1;
        if (leaf == true)
        {
                while (i >= 0 && keys[i] > k)
                {
                        keys[i+1] = keys[i];
                        i--;
                }
                keys[i+1] = k;
                n = n+1;
        }
        else
        {
                while (i >= 0 && keys[i] > k)
                        i--;
                if (C[i+1]->n == 2*t-1)
                {
                        splitChild(i+1, C[i+1]);
                if (keys[i+1] < k)
                                i++;
                }
                C[i+1]->insertNonFull(k);
        }
}
void BTreeNode::splitChild(int i, BTreeNode *y)
{
```

```
	BTreeNode *z = new BTreeNode(y->t, y->leaf);
	z->n = t - 1;

	for (int j = 0; j < t-1; j++)
		z->keys[j] = y->keys[j+t];

	if (y->leaf == false)
	{
		for (int j = 0; j < t; j++)
			z->C[j] = y->C[j+t];
	}

	y->n = t - 1;

	for (int j = n; j >= i+1; j--)
		C[j+1] = C[j];
	C[i+1] = z;
	for (int j = n-1; j >= i; j--)
		keys[j+1] = keys[j];

	keys[i] = y->keys[t-1];

	n = n + 1;
}
void BTreeNode::traverse()
{
	int i;
	for (i = 0; i < n; i++)
	{
		if (leaf == false)
			C[i]->traverse();
		cout << " " << keys[i];
	}
	if (leaf == false)
		C[i]->traverse();
}
BTreeNode *BTreeNode::search(int k)
{
	int i = 0;
	while (i < n && k > keys[i])
		i++;

	if (keys[i] == k)
		return this;
	if (leaf == true)
		return NULL;
	return C[i]->search(k);
```

```
}

void BTree::remove(int k)
{
	if (!root)
	{
		cout << "The tree is empty\n";
		return;
	}
	root->remove(k);
	if (root->n==0)
	{
		BTreeNode *tmp = root;
		if (root->leaf)
			root = NULL;
		else
			root = root->C[0];


		delete tmp;
	}
	return;
}

int main()
{
	BTree t(3);

	t.insert(1);
	t.insert(2);
	t.insert(5);
	t.insert(10);
	t.insert(7);
	t.insert(3);
	t.insert(6);
	t.insert(16);
	t.insert(12);


	cout << "Traversal of tree constructed is\n";
	t.traverse();
	cout << endl;

	t.remove(6);
	cout << "Traversal of tree after removing 6\n";
	t.traverse();
	cout << endl;
```

```
        t.remove(12);
        cout << "Traversal of tree after removing 12\n";
        t.traverse();
        cout << endl;

        t.remove(7);
        cout << "Traversal of tree after removing 7\n";
        t.traverse();
        cout << endl;

        t.remove(5);
        cout << "Traversal of tree after removing 5\n";
        t.traverse();
        cout << endl;

        t.remove(2);
        cout << "Traversal of tree after removing 2\n";
        t.traverse();
        cout << endl;

        t.remove(16);
        cout << "Traversal of tree after removing 16\n";
        t.traverse();
        cout << endl;

        return 0;
}
```

Output:



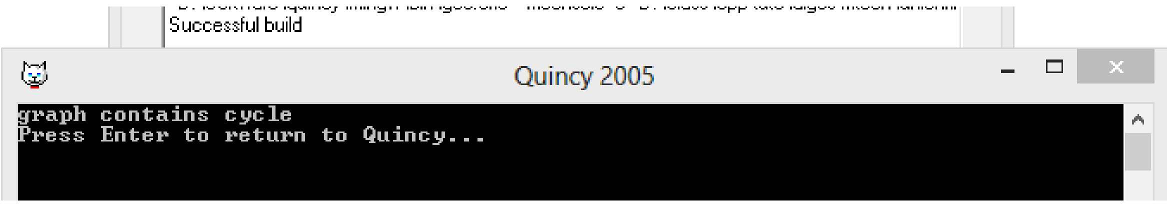
Practical-6: Implementation of operations on union-find data structures

.

```
#include <iostream.h>
class Edge
{
	public:
	int src, dest;
};
class Graph
{
	public:
	int V, E;
	Edge* edge;
};
Graph* createGraph(int V, int E)
{
	Graph* graph = new Graph();
	graph->V = V;
	graph->E = E;

	graph->edge = new Edge[graph->E * sizeof(Edge)];
	return graph;
}

int find(int parent[], int i)
{
	if (parent[i] == -1)
		return i;
	return find(parent, parent[i]);
}

void Union(int parent[], int x, int y)
{
	int xset = find(parent, x);
	int yset = find(parent, y);
	if(xset != yset)
	{
		parent[xset] = yset;
	}
}
int isCycle( Graph* graph )
{
	int *parent = new int[graph->V * sizeof(int)];
	memset(parent, -1, sizeof(int) * graph->V);
	for(int i = 0; i < graph->E; ++i)
	{
```

```
			int x = find(parent, graph->edge[i].src);
			int y = find(parent, graph->edge[i].dest);

			if (x == y)
				return 1;

			Union(parent, x, y);
		}
		return 0;
}
int main()
{
		int V = 3, E = 3;
		Graph* graph = createGraph(V, E);
		graph->edge[0].src = 0;
		graph->edge[0].dest = 1;
		graph->edge[1].src = 1;
		graph->edge[1].dest = 2;

		graph->edge[2].src = 0;
		graph->edge[2].dest = 2;

		if (isCycle(graph))
			cout<<"graph contains cycle";
		else
			cout<<"graph doesn't contain cycle";

		return 0;
}
```

Output:



Practical-7: Implementation of Bellman-Ford algorithm

```
#include <bits/stdc++.h>
struct Edge {
	int src, dest, weight;
};
struct Graph {
	int V, E;
	struct Edge* edge;
};
struct Graph* createGraph(int V, int E)
{
	struct Graph* graph = new Graph;
	graph->V = V;
	graph->E = E;
	graph->edge = new Edge[E];
	return graph;
}
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]);
}
void BellmanFord(struct Graph* graph, int src)
{
	int V = graph->V;
	int E = graph->E;
	int dist[V];
	for (int i = 0; i < V; i++)
		dist[i] = INT_MAX;
	dist[src] = 0;
	for (int i = 1; i <= V - 1; i++) {
		for (int j = 0; j < E; j++) {
			int u = graph->edge[j].src;
			int v = graph->edge[j].dest;
			int weight = graph->edge[j].weight;
			if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
				dist[v] = dist[u] + weight;
		}
	}
	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] != INT_MAX && dist[u] + weight < dist[v]) {
			printf("Graph contains negative weight cycle");
			return; // If negative cycle is detected, simply return
```

```
                }
        }
        printArr(dist, V);
        return;
}
int main()
{
        int V = 5;
        int E = 8;
        struct Graph* graph = createGraph(V, E);
        graph->edge[0].src = 0;
        graph->edge[0].dest = 1;
        graph->edge[0].weight = -1;
        graph->edge[1].src = 0;
        graph->edge[1].dest = 2;
        graph->edge[1].weight = 4;
        graph->edge[2].src = 1;
        graph->edge[2].dest = 2;
        graph->edge[2].weight = 3;
        graph->edge[3].src = 1;
        graph->edge[3].dest = 3;
        graph->edge[3].weight = 2;
        graph->edge[4].src = 1;
        graph->edge[4].dest = 4;
        graph->edge[4].weight = 2;
        graph->edge[5].src = 3;
        graph->edge[5].dest = 2;
        graph->edge[5].weight = 5;
        graph->edge[6].src = 3;        graph->edge[6].dest = 1;
        graph->edge[6].weight = 1;
        graph->edge[7].src = 4; graph->edge[7].dest = 3;     graph->edge[7].weight = -3;
        BellmanFord(graph, 0);

        return 0;
}
```

Output:

```
Vertex Distance from Source
0                0
1                -1
2                2
3                -2
4                1
```