Practical File

Name: Deepak Prakash

Roll Number: 2019UCS2018

Subject: Advanced Algorithms

Subject Code: CMCSE51

INDEX

Experiment 1

AIM: Find the Tree longest route in complete binary tree.

Programming Language: C++

Program:

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

class Tree
{
public:
    int data;
    Tree *left;
    Tree *right;
    Tree(int data)
    {
        this->data = data;
        this->left = NULL;
        this->right = NULL;
    }

    void insert(int data)
    {
        if (data == this->data)
        {
            return;
        }
        if (data < this->data)
        {
            if (left == NULL)
            {
                left = new Tree(data);
            }
            else
            {
                left->insert(data);
            }
        }
        else
        {
            if (right == NULL)
            {
```

```
                right = new Tree(data);
            }
            else
            {
                right->insert(data);
            }
        }
    }
    void display()
    {
        if (left != NULL)
        {
            left->display();
        }
        cout << data << " ";
        if (right != NULL)
        {
            right->display();
        }
    }

    int getDepth(Tree *root)
    {
        if (root == NULL)
            return 0;
        else
        {
            int lDepth = getDepth(root->left);
            int rDepth = getDepth(root->right);

            if (lDepth > rDepth)
                return (lDepth + 1);
            else
                return (rDepth + 1);
        }
    }
    int maxpathlength()
    {
        int lh=0,rh=0;
        int ld=0,rd=0;
        if (this == NULL)
            return 0;

        int ldiameter = this->left->maxpathlength();
        int rdiameter = this->right->maxpathlength();

        int diameter = getDepth(this->left) + getDepth(this->right) + 1;
        return max(diameter,
```

```
            max(ldiameter, rdiameter));
    }
};

int main()
{
    int n;
    cout << "Enter the number of nodes: ";
    cin >> n;

    Tree *root = nullptr;
    srand(time(0));
    cout<<"Data entered:  ";
    for (int i = 0; i < n; i++)
    {
        int data = rand() % 10000;
        cout<<data<<" ";
        if (root == nullptr)
        {
            root = new Tree(data);
        }
        else
        {
            root->insert(data);
        }
    }
    cout << "\nTree: ";
    root->display();

    cout<<endl<<"Max path length: "<<root->maxpathlength();
    return 0;
}
```

<u>Output:</u>

```
(base) PS D:\advanced algorithms\final> .\longest_route.exe
Enter the number of nodes: 10
Data entered:  6215 7999 9226 2176 7323 9696 8867 2281 3495 394
Tree: 394 2176 2281 3495 6215 7323 7999 8867 9226 9696
Max path length: 7
(base) PS D:\advanced algorithms\final>
```

<u>Time Complexity:</u> $O(N^2)$

<u>Space Complexity:</u> O(N)

Experiment 2

AIM: Implement Min/Max Heap using array data structure

Programming Language: C++

Program:

```
#include<iostream>
#include<climits>
using namespace std;
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);
    void show();
};


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 MinHeap::show(){
    for(int i=0;i<heap_size;i++)
        cout<<harr[i]<<" ";
    cout<<endl;
}


int main()
{
    MinHeap h(50);
    h.insertKey(5);
    h.insertKey(20);
    h.insertKey(3);
    h.insertKey(25);
    h.insertKey(40);
    h.insertKey(45);
    h.insertKey(65);
    cout<<"Heap:"<<endl;
    h.show();
    cout <<"Extract Min:"<<h.extractMin() <<endl;
    cout <<"Get Min:"<< h.getMin() << "\n";
    cout<<"Decrease key 2 to 1"<<endl;
    h.decreaseKey(2, 1);
    h.show();
    cout <<"Get Min:" << h.getMin()<<endl;
```

```
    h.deleteKey(1);
    cout<<"After deleting key 1"<<endl;
    h.show();

    return 0;
}
```

Output:

```
Heap:
3 20 5 25 40 45 65
Extract Min:3
Get Min:5
Decrease key 2 to 1
1 20 5 25 40 65
Get Min:1
After deleting key 1
1 25 5 65 40
```

Time Complexity:

InsertKey = O(log(N))

DeleteKey = O(log(N))

DecreaseKey = O(log(N)

GetMin = O(1)

ExtractMin = O(log(N))

Space Complexity:

O(N) for heap of n-elements

Experiment 3

AIM: Implement Min Binomial heap

Programming Language: C++

Program:

```
#include <bits/stdc++.h>
using namespace std;

struct Node
{
    int data, degree;
    Node *child, *sibling, *parent;
};

Node *newNode(int key)
{
    Node *temp = new Node;
    temp->data = key;
    temp->degree = 0;
    temp->child = temp->parent = temp->sibling = NULL;
    return temp;
}

Node *mergeBinomialTrees(Node *b1, Node *b2)
{

    if (b1->data > b2->data)
        swap(b1, b2);

    b2->parent = b1;
    b2->sibling = b1->child;
    b1->child = b2;
    b1->degree++;

    return b1;
}

list<Node *> unionBionomialHeap(list<Node *> l1,
                                list<Node *> l2)
{
    list<Node *> _new;
    list<Node *>::iterator it = l1.begin();
    list<Node *>::iterator ot = l2.begin();
```

```
    while (it != l1.end() && ot != l2.end())
    {

        if ((*it)->degree <= (*ot)->degree)
        {
            _new.push_back(*it);
            it++;
        }

        else
        {
            _new.push_back(*ot);
            ot++;
        }
    }

    while (it != l1.end())
    {
        _new.push_back(*it);
        it++;
    }

    while (ot != l2.end())
    {
        _new.push_back(*ot);
        ot++;
    }
    return _new;
}

list<Node *> adjust(list<Node *> bin_heap)
{
    if (bin_heap.size() <= 1)
        return bin_heap;
    list<Node *> newbin_heap;
    list<Node *>::iterator it1, it2, it3;
    it1 = it2 = it3 = bin_heap.begin();

    if (bin_heap.size() == 2)
    {
        it2 = it1;
        it2++;
        it3 = bin_heap.end();
    }
    else
    {
        it2++;
        it3 = it2;
```

```
        it3++;
    }
    while (it1 != bin_heap.end())
    {

        if (it2 == bin_heap.end())
            it1++;

        else if ((*it1)->degree < (*it2)->degree)
        {
            it1++;
            it2++;
            if (it3 != bin_heap.end())
                it3++;
        }

        else if (it3 != bin_heap.end() &&
                (*it1)->degree == (*it2)->degree &&
                (*it1)->degree == (*it3)->degree)
        {
            it1++;
            it2++;
            it3++;
        }

        else if ((*it1)->degree == (*it2)->degree)
        {
            Node *temp;
            *it1 = mergeBinomialTrees(*it1, *it2);
            it2 = bin_heap.erase(it2);
            if (it3 != bin_heap.end())
                it3++;
        }
    }
    return bin_heap;
}

list<Node *> insertATreeInHeap(list<Node *> bin_heap,
                               Node *tree)
{

    list<Node *> temp;

    temp.push_back(tree);

    temp = unionBionomialHeap(bin_heap, temp);

    return adjust(temp);
```

```
}

list<Node *> removeMinFromTreeReturnBHeap(Node *tree)
{
    list<Node *> heap;
    Node *temp = tree->child;
    Node *lo;

    while (temp)
    {
        lo = temp;
        temp = temp->sibling;
        lo->sibling = NULL;
        heap.push_front(lo);
    }
    return heap;
}

list<Node *> insert(list<Node *> _head, int key)
{
    Node *temp = newNode(key);
    return insertATreeInHeap(_head, temp);
}

Node *getMin(list<Node *> bin_heap)
{
    list<Node *>::iterator it = bin_heap.begin();
    Node *temp = *it;
    while (it != bin_heap.end())
    {
        if ((*it)->data < temp->data)
            temp = *it;
        it++;
    }
    return temp;
}

list<Node *> extractMin(list<Node *> bin_heap)
{
    list<Node *> newbin_heap, lo;
    Node *temp;

    temp = getMin(bin_heap);
    list<Node *>::iterator it;
    it = bin_heap.begin();
    while (it != bin_heap.end())
    {
        if (*it != temp)
```

```
        {

            newbin_heap.push_back(*it);
        }
        it++;
    }
    lo = removeMinFromTreeReturnBHeap(temp);
    newbin_heap = unionBionomialHeap(newbin_heap, lo);
    newbin_heap = adjust(newbin_heap);
    return newbin_heap;
}

void printTree(Node *h)
{
    while (h)
    {
        cout << h->data << " ";
        printTree(h->child);
        h = h->sibling;
    }
}

void printHeap(list<Node *> bin_heap)
{
    list<Node *>::iterator it;
    it = bin_heap.begin();
    while (it != bin_heap.end())
    {
        printTree(*it);
        it++;
    }
}

int main()
{
    int ch, key;
    list<Node *> bin_heap;

    bin_heap = insert(bin_heap, 5);
    bin_heap = insert(bin_heap, 20);
    bin_heap = insert(bin_heap, 8);
    bin_heap = insert(bin_heap, 13);
    bin_heap = insert(bin_heap, 28);

    cout << "Heap elements after insertion:\n";
    printHeap(bin_heap);

    Node *temp = getMin(bin_heap);
```

```
    cout << "\nMinimum element of heap "
        << temp->data << "\n";

    bin_heap = extractMin(bin_heap);
    cout << "Heap after deletion of minimum element\n";
    printHeap(bin_heap);

    return 0;
}
```

Output:

```
Heap elements after insertion:
28 5 8 13 20
Minimum element of heap 5
Heap after deletion of minimum element
8 20 28 13
```

Time Complexity:

Insert: O(log(N))

GetMin: O(log(N))

ExtractMin: O(log(N))

Space Complexity:

O(N) for heap of N-elements

Experiment 4

AIM: Implement Fibonacci heap

Programming Language: C++

Program:

```
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <malloc.h>
using namespace std;

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 = new 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 + 1];
    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(10);
    insertion(20);
    insertion(8);
    insertion(14);
    insertion(4);
    insertion(2);
    display();
    cout << "Extracting min" << endl;
    Extract_min();
    display();
    cout << "Decrease value of 8 to 3" << endl;
    Find(mini, 8, 3);
    display();
    cout << "Delete the node 2" << endl;
    Deletion(2);
    display();

    return 0;
}
```

Output:

```
(base) PS D:\advanced algorithms\final> .\fi
Creating an initial heap
The root nodes of Heap are:
2--4--8--10--20--14
The heap has 6 nodes

Extracting min
The root nodes of Heap are:
4--14
The heap has 5 nodes

Decrease value of 8 to 3
The root nodes of Heap are:
3--4--14
The heap has 5 nodes

Delete the node 2
Key Deleted
The root nodes of Heap are:
4--14
The heap has 4 nodes
```

Time Complexity:

Making of Heap: O(1)

Insertion: O(1)

GetMin: O(1)

ExtractMin: O(log(N))

DecreaseKey: O(1)

DeleteKey: O(log(N))

Space Complexity:

O(N) for heap of N-elements

Experiment 5

AIM: Implementation of splay tree using its all kinds of rotation

Programming Language: C++

Program:

```
#include <bits/stdc++.h>
using namespace std;
class node
{
public:
    int key;
    node *left, *right;
};
node *TreeNode(int key)
{
    node *Node = new node();
    Node->key = key;
    Node->left = Node->right = NULL;
    return (Node);
}
node *rightRotate(node *x)
{
    node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}
node *leftRotate(node *x)
{
    node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

node *splay(node *root, int key)
{
    if (root == NULL || root->key == key)
        return root;
    if (root->key > key)
    {
        if (root->left == NULL)
            return root;
```

```
        if (root->left->key > key)
        {
            root->left->left = splay(root->left->left, key);
            root = rightRotate(root);
        }
        else if (root->left->key < key)
        {
            root->left->right = splay(root->left->right, key);
            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }
        return (root->left == NULL) ? root : rightRotate(root);
    }

    else
    {
        if (root->right == NULL)
            return root;
        if (root->right->key > key)
        {
            root->right->left = splay(root->right->left, key);
            if (root->right->left != NULL)
                root->right = rightRotate(root->right);
        }
        else if (root->right->key < key)
        {
            root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        }
        return (root->right == NULL) ? root : leftRotate(root);
    }
}

node *bstSearch(node *root, int key)
{
    return splay(root, key);
}

void preOrder(node *root)
{
    if (root != NULL)
    {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

```cpp
int main()
{
    node *root = TreeNode(10);
    root->left = TreeNode(15);
    root->right = TreeNode(25);
    root->left->left = TreeNode(40);
    root->left->left->left = TreeNode(30);
    root->left->left->left->left = TreeNode(20);
    cout<<"Preorder before search: \n";
    preOrder(root);
    root = bstSearch(root, 20);
    cout<<"\nPreorder after search of 20: \n";
    preOrder(root);
    return 0;
}
```

Output:

```
Preorder before search:
10 15 40 30 20 25
Preorder after search of 20:
25 10 15 40 30 20
(base) PS D:\advanced algorithms\final>
```

Time Complexity: O(log(N))

Space Complexity: O(N)

Experiment 6

AIM: Implement incremental dynamic connectivity problem

Programming Language: C++

Program:

```
#include <bits/stdc++.h>
using namespace std;
int N, Q, ans[10];
int nc, sz;
map<pair<int, int>, vector<pair<int, int> > > graph;
int p[10], r[10];
int *t[20], v[20];
int n[20];
int setv(int* a, int b, int toAdd)
{
    t[sz] = a;
    v[sz] = *a;
    *a = b;
    n[sz] = toAdd;
    ++sz;
    return b;
}
void rollback(int x)
{
    for (; sz > x;) {
        --sz;
        *t[sz] = v[sz];
        nc += n[sz];
    }
}
int find(int n)
{
    return p[n] ? find(p[n]) : n;
}
bool merge(int a, int b)
{
    a = find(a), b = find(b);
    if (a == b)
        return 0;
    nc--;
    if (r[b] > r[a])
        std::swap(a, b);
    setv(r + b, r[a] + r[b], 0);
```

```
    return setv(p + b, a, 1), 1;
}
void solve(int start, int end)
{
    int tmp = sz;
    for (auto it = graph.begin();
        it != graph.end(); ++it) {
        int u = it->first.first;
        int v = it->first.second;
        for (auto it2 = it->second.begin();
            it2 != it->second.end(); ++it2) {
            int w = it2->first, c = it2->second;
            if (w <= start && c >= end) {
                merge(u, v);
                break;
            }
        }
    }
    if (start == end) {
        ans[start] = nc;
        return;
    }
    int mid = (start + end) >> 1;
    solve(start, mid);
    solve(mid + 1, end);
    rollback(tmp);
}

void componentAtInstant(vector<int> queries[])
{
    nc = N;
    for (int i = 0; i < Q; i++) {
        int t = queries[i][0];
        int u = queries[i][1], v = queries[i][2];
        if (u > v)
            swap(u, v);
        if (t == 1) {
            graph[{ u, v }].push_back({ i, Q });
        }
        else {
            graph[{ u, v }].back().second = i - 1;
        }
    }
    solve(0, Q);
}


int main()
{
```

```
    N = 3, Q = 4;
    vector<int> queries[] = { { 1, 1, 2 }, { 1, 2, 3 }, { 2, 1, 2 }, { 2, 2, 3
} };
    componentAtInstant(queries);
    for (int i = 0; i < Q; i++)
        cout << ans[i] << " ";
    return 0;
}
```

Output:

```
(base) PS D:\advanced algorithms\final> .\incremental.exe
2 1 2 3
(base) PS D:\advanced algorithms\final> 
```

Time Complexity: O(α(N))

Space Complexity: O(N)

Experiment 7

AIM: Implementation of Rabin karp fingerprinting algorithm for checking whether a given string exist in other string or not.

Programming Language: C++

Program:

```
#include <bits/stdc++.h>
using namespace std;
#define d 10

void rabinKarp(string pattern, string text, int q)
{
    int m = pattern.length();
    int n = text.length();
    int i, j;
    int p = 0;
    int t = 0;
    int h = 1;

    for (i = 0; i < m - 1; i++)
        h = (h * d) % q;

    for (i = 0; i < m; i++)
    {
        p = (d * p + pattern[i]) % q;
        t = (d * t + text[i]) % q;
    }

    for (i = 0; i <= n - m; i++)
    {
        if (p == t)
        {
            for (j = 0; j < m; j++)
            {
                if (text[i + j] != pattern[j])
                    break;
            }

            if (j == m)
                cout << "Pattern is found at position: " << i + 1 << endl;
        }

        if (i < n - m)
```

```
        {
            t = (d * (t - text[i] * h) + text[i + m]) % q;
            if (t < 0)
                t = (t + q);
        }
    }
}

int main()
{
    string text, pattern;
    cout << "Enter text : ";
    getline(cin, text);
    cout << "Enter pattern : ";
    getline(cin,pattern);
    int q = 13;
    rabinKarp(pattern, text, q);
    return 0;
}
```

Output:

```
(base) PS D:\advanced algorithms\final> .\rabin.exe
Enter text : hello world this is a new text
Enter pattern : this
Pattern is found at position: 13
(base) PS D:\advanced algorithms\final>
```

Time Complexity:

Average and Best case: O(n+m)

Worst case: O(nm)

Space Complexity: O(1)

Experiment 8

AIM: Implement a suffix tree for a given string

Programming Language: C++

Program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define MAX_CHAR 256
struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];
    struct SuffixTreeNode *suffixLink;
    int start;
    int *end;
    int suffixIndex;
};
typedef struct SuffixTreeNode Node;
char text[100];
Node *root = NULL;
Node *lastNewNode = NULL;
Node *activeNode = NULL;
int count=0;
int activeEdge = -1;
int activeLength = 0;
int remainingSuffixCount = 0;
int leafEnd = -1;
int *rootEnd = NULL;
int *splitEnd = NULL;
int size = -1;
Node *newNode(int start, int *end)
{
    count++;
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;
    node->suffixLink = root;
    node->start = start;
    node->end = end;
    node->suffixIndex = -1;
    return node;
```

```
}
int edgeLength(Node *n) {
    return *(n->end) - (n->start) + 1;
}
int walkDown(Node *currNode)
{
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge =
        (int)text[activeEdge+edgeLength(currNode)]-(int)' ';
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}
void extendSuffixTree(int pos)
{
    leafEnd = pos;
    remainingSuffixCount++;
    lastNewNode = NULL;

    while(remainingSuffixCount > 0) {
        if (activeLength == 0) {

            activeEdge = (int)text[pos]-(int)' ';
        }


        if (activeNode->children[activeEdge] == NULL)
        {

            activeNode->children[activeEdge] =
                                  newNode(pos, &leafEnd);

            if (lastNewNode != NULL)
            {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = NULL;
            }
        }


        else
        {
```

```
            Node *next = activeNode->children[activeEdge];
            if (walkDown(next))
            {

                continue;
            }

            if (text[next->start + activeLength] == text[pos])
            {



                if(lastNewNode != NULL && activeNode != root)
                {
                    lastNewNode->suffixLink = activeNode;
                    lastNewNode = NULL;
                }

                activeLength++;

                break;
            }

            splitEnd = (int*) malloc(sizeof(int));
            *splitEnd = next->start + activeLength - 1;

            Node *split = newNode(next->start, splitEnd);
            activeNode->children[activeEdge] = split;

            split->children[(int)text[pos]-(int)' '] =
                                        newNode(pos, &leafEnd);
            next->start += activeLength;
            split->children[activeEdge] = next;

            if (lastNewNode != NULL)
            {

                lastNewNode->suffixLink = split;
            }

            lastNewNode = split;
        }

        remainingSuffixCount--;
        if (activeNode == root && activeLength > 0)
```

```
        {
            activeLength--;
            activeEdge = (int)text[pos -
                            remainingSuffixCount + 1]-(int)' ';
        }


        else if (activeNode != root)
        {
            activeNode = activeNode->suffixLink;
        }
    }
}
void print(int i, int j)
{
    int k;
    for (k=i; k<=j; k++)
        printf("%c", text[k]);
}
void setSuffixIndexByDFS(Node *n, int labelHeight)
{
    if (n == NULL) return;
    if (n->start != -1)
    {

        print(n->start, *(n->end));
    }
    int leaf = 1;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            if (leaf == 1 && n->start != -1)
                printf(" [%d]\n", n->suffixIndex);


            leaf = 0;
            setSuffixIndexByDFS(n->children[i],
                labelHeight + edgeLength(n->children[i]));
        }
    }
    if (leaf == 1)
    {
        n->suffixIndex = size - labelHeight;
        printf(" [%d]\n", n->suffixIndex);
```

```
    }
}
void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int));
    *rootEnd = - 1;
    root = newNode(-1, rootEnd);
    activeNode = root;
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);

    freeSuffixTreeByPostOrder(root);
}
int main(int argc, char *argv[])
{
    strcpy(text, "hello"); buildSuffixTree();
    printf("Number of nodes in suffix tree are %d\n",count);
    return 0;
}
```

Output:

```
(base) PS D:\advanced algorithms\final> .\suffix.exe
ello [1]
hello [0]
l [-1]
lo [2]
o [3]
o [4]
Number of nodes in suffix tree are 7
(base) PS D:\advanced algorithms\final>
```

Time Complexity: O(N)

Space Complexity: $O(N^2)$

Experiment 9

AIM: Perform ford Fulkerson algorithm in Maximum cost flow network

Programming Language: C++

Program:

```
#include <bits/stdc++.h>
using namespace std;

bool bfs(vector<vector<int>> &graph, int s, int t, int n, vector<int> &parent)
{
    fill(parent.begin(), parent.end(), -1);
    queue<pair<int, int>> q;
    q.push({s, 1e9});
    parent[s] = -2;

    while (!q.empty())
    {
        int u = q.front().first;
        int cap = q.front().second;
        q.pop();

        for (int v = 0; v < n; v++)
        {
            if (u != v && graph[u][v] != 0 && parent[v] == -1)
            {
                parent[v] = u;
                int min_cap = min(cap, graph[u][v]);
                if (v == t)
                {
                    return min_cap;
                }
                q.push({v, min_cap});
            }
        }
    }
    return 0;
}

int fordFulkerson(vector<vector<int>> &graph, int s, int t, int n)
{
    vector<int> parent(n, -1);
    int max_flow = 0, min_cap = 0;
```

```
    while (min_cap = bfs(graph, s, t, n, parent))
    {
        max_flow += min_cap;
        int v = t;

        while (v != s)
        {
            int u = parent[v];
            graph[u][v] -= min_cap;
            graph[v][u] += min_cap;
            v = u;
        }
    }
    return max_flow;
}

void addEdge(vector<vector<int>> &graph, int u, int v, int w)
{
    graph[u][v] = w;
}

int main()
{
    int n;
    cout << "Enter number of vertex : ";
    cin >> n;

    int e;
    cout << "Enter number of edge : ";
    cin >> e;

    vector<vector<int>> graph(n, vector<int>(n, 0));
    int u, v, w;
    for (int I = 0; I < e; i++)
    {
        cout << "Enter vertex(u) , vertex(v) and weight(w) : ";
        cin >> u >> v >> w;
        addEdge(graph, u, v, w);
    }
    cout << "Max Flow: " << fordFulkerson(graph, 0, 5, n) << endl;
}
```

Output:

```
(base) PS D:\advanced algorithms\final> .\a.exe
Enter number of vertex : 7
Enter number of edge : 12
Enter vertex(u) , vertex(v) and weight(w) : 0 1 7
Enter vertex(u) , vertex(v) and weight(w) : 0 2 10
Enter vertex(u) , vertex(v) and weight(w) : 1 2 1
Enter vertex(u) , vertex(v) and weight(w) : 1 3 3
Enter vertex(u) , vertex(v) and weight(w) : 1 4 5
Enter vertex(u) , vertex(v) and weight(w) : 2 3 2
Enter vertex(u) , vertex(v) and weight(w) : 2 5 7
Enter vertex(u) , vertex(v) and weight(w) : 3 4 3
Enter vertex(u) , vertex(v) and weight(w) : 3 5 2
Enter vertex(u) , vertex(v) and weight(w) : 4 5 2
Enter vertex(u) , vertex(v) and weight(w) : 4 6 10
Enter vertex(u) , vertex(v) and weight(w) : 5 6 4
Max Flow: 11
(base) PS D:\advanced algorithms\final>
```

Time Complexity: $O(V*E^2)$

Space Complexity: $O(V)$

Experiment 10

AIM: Find maximum bipartite matching in a bipartite graph

Programming Language: C++

Program:

```
#include <iostream>
#define M 5
#define N 6
using namespace std;

bool bipartiteGraph[M][N] = {
    {0, 1, 1, 0, 0, 0},
    {1, 0, 0, 1, 0, 0},
    {0, 0, 1, 0, 0, 0},
    {0, 0, 1, 1, 0, 0},
    {0, 0, 0, 0, 0, 0}};

bool bipartiteMatch(int u, bool visited[], int assign[])
{
    for (int v = 0; v < N; v++)
    {
        if (bipartiteGraph[u][v] && !visited[v])
        {
            visited[v] = true;

            if (assign[v] < 0 || bipartiteMatch(assign[v], visited, assign))
            {
                assign[v] = u;
                return true;
            }
        }
    }
    return false;
}

int maxMatch()
{
    int assign[N];
    for (int i = 0; i < N; i++)
        assign[i] = -1;
    int jobCount = 0;
```

```cpp
    for (int u = 0; u < M; u++)
    {
        bool visited[N];
        for (int i = 0; i < N; i++)
            visited[i] = false;
        if (bipartiteMatch(u, visited, assign))
            jobCount++;
    }
    return jobCount;
}

int main()
{
    cout << "Maximum number of applicants matching for job: " << maxMatch();
}
```

Output:

```
(base) PS D:\advanced algorithms\final> g++ bipartite.cpp -o b
(base) PS D:\advanced algorithms\final> .\bipartite.exe
Maximum number of applicants matching for job: 4
(base) PS D:\advanced algorithms\final>
```

Time Complexity: O(V*E)

Space Complexity: O(V+E)

Experiment 11

AIM: Consider all the subset of vertices one by one and find out whether it covers all edges of the graph or not.

Programming Language: C++

Program:

```
#include <bits/stdc++.h>
using namespace std;

class Graph
{
    int V;
    list<int> *adj;

public:
    Graph(int V);
    void addEdge(int v, int w);
    void printVertexCover();
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);
}

void Graph::printVertexCover()
{
    bool visited[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    list<int>::iterator i;

    for (int u = 0; u < V; u++)
```

```
    {
        if (visited[u] == false)
        {
            for (i = adj[u].begin(); i != adj[u].end(); ++i)
            {
                int v = *i;
                if (visited[v] == false)
                {
                    visited[v] = true;
                    visited[u] = true;
                    break;
                }
            }
        }
    }

    for (int i = 0; i < V; i++)
        if (visited[i])
            cout << i << " ";
}

int main()
{
    int n;
    cout << "Enter no. of vertices : ";
    cin >> n;
    Graph g(n);

    int e;
    cout << "Enter no. of edges : ";
    cin >> e;

    for (int i = 0; i < e; i++)
    {
        int u, v;
        cout << "Enter start and end vertex of a edge : ";
        cin >> u >> v;
        g.addEdge(u, v);
    }

    cout << "Vertex Cover set in given graph : ";
    g.printVertexCover();

    return 0;
}
```

Output:

```
Enter no. of vertices : 4
Enter no. of edges : 5
Enter start and end vertex of a edge : 0 1
Enter start and end vertex of a edge : 0 2
Enter start and end vertex of a edge : 2 3
Enter start and end vertex of a edge : 0 3
Enter start and end vertex of a edge : 1 3
Vertex Cover set in given graph : 0 1 2 3
(base) PS D:\advanced algorithms\final>
```

Time Complexity: O(V+E)

Space Complexity: O(V)

Experiment 12

AIM: Implementation of maximal independent set from a given graph using backtracking

Programming Language: C++

Program:

```
#include <bits/stdc++.h>
using namespace std;

set<set<int>> independentSets;

set<set<int>> maximalIndependentSets;

map<pair<int, int>, int> edges;
vector<int> vertices;

void printAllIndependentSets()
{
    for (auto iter : independentSets)
    {
        cout << "{ ";
        for (auto iter2 : iter)
        {
            cout << iter2 << " ";
        }
        cout << "}";
    }
    cout << endl;
}

void printMaximalIndependentSets()
{
    int maxCount = 0;
    int localCount = 0;
    for (auto iter : independentSets)
    {

        localCount = 0;
        for (auto iter2 : iter)
        {
            localCount++;
```

```
        }
        if (localCount > maxCount)
            maxCount = localCount;
    }
    for (auto iter : independentSets)
    {

        localCount = 0;
        set<int> tempMaximalSet;

        for (auto iter2 : iter)
        {
            localCount++;
            tempMaximalSet.insert(iter2);
        }
        if (localCount == maxCount)
            maximalIndependentSets
                .insert(tempMaximalSet);
    }
    for (auto iter : maximalIndependentSets)
    {
        cout << "{ ";
        for (auto iter2 : iter)
        {
            cout << iter2 << " ";
        }
        cout << "}";
    }
    cout << endl;
}

bool isSafeForIndependentSet(
    int vertex,
    set<int> tempSolutionSet)
{
    for (auto iter : tempSolutionSet)
    {
        if (edges[make_pair(iter, vertex)])
        {
            return false;
        }
    }
    return true;
}

void findAllIndependentSets(
```

```
    int currV,
    int setSize,
    set<int> tempSolutionSet)
{
    for (int i = currV; i <= setSize; i++)
    {
        if (isSafeForIndependentSet(
                vertices[i - 1],
                tempSolutionSet))
        {
            tempSolutionSet
                .insert(vertices[i - 1]);
            findAllIndependentSets(
                i + 1,
                setSize,
                tempSolutionSet);
            tempSolutionSet
                .erase(vertices[i - 1]);
        }
    }
    independentSets
        .insert(tempSolutionSet);
}

int main()
{
    int V, E;
    cout<<"Enter no. of vertices: ";
    cin >> V;
    cout<<"Enter no. of edges: ";
    cin >> E;


    for (int i = 1; i <= V; i++)
        vertices.push_back(i);

    pair<int, int> edge;
    int x, y;
    for (int i = 0; i < E; i++)
    {
        cout<<"Enter edge (U,V): ";
        cin >> x >> y;
        edge.first = x;
        edge.second = y;
        edges[edge] = 1;
        int t = edge.first;
```

```
        edge.first = edge.second;
        edge.second = t;
        edges[edge] = 1;
    }

    set<int> tempSolutionSet;

    findAllIndependentSets(1,V,tempSolutionSet);

    printAllIndependentSets();

    printMaximalIndependentSets();

    return 0;
}
```

Output:

```
(base) PS D:\advanced algorithms\final> .\a.exe
Enter no. of vertices: 3
Enter no. of edges: 1
Enter edge (U,V): 1 2
{ }{ 1 }{ 1 3 }{ 2 }{ 2 3 }{ 3 }
{ 1 3 }{ 2 3 }
(base) PS D:\advanced algorithms\final>
```

Time Complexity: $O(2^N)$

Space Complexity: $O(2^N)$