Annexure - I

Optimizing the N-Queen Problem: A Visual Implementation

from PEP classes

Lovely Professional University

Submitted in partial fulfilment of the requirements for the award of degree of

Program Name: Bachelor of Technology

(Computer Science and Engineering)

Lovely Professional University

PHAGWARA, PUNJAB

From 05/06/2024 to 26/06/2024

Submitted by

Name: Nikhil Kumar Agarwal K

Registration No: 12215039

Signature of student: 

Annexure-II Student Declaration

To whom so ever it may concern

I, **Nikhil Kumar Agarwal, 12215039**, hereby declare that the work done by me on **"Optimizing the N-Queen Problem: A Visual Implementation"** from **05/06/2024 to 26/06/2024**, is a record of original work for the partial fulfilment of the requirements for the award of the degree**, Bachelor of Technology.**

Nikhil Kumar Agarwal K

12215039

Dated: 30/08/2024

Index

Chapter 1: INTRODUCTION OF THE PROJECT UNDERTAKEN

1.1 Objectives

The primary objective of the Placement Enhancement Program (PEP) was to equip students with advanced skills in competitive programming using C++. The program aimed to develop a solid foundation in data structures, algorithms, and problem-solving techniques, enabling students to excel in coding competitions and technical interviews. Additionally, students were expected to complete a project that applied the concepts learned during the program.

1.2 Scope of the Work

The Placement Enhancement Program (PEP) focused on a comprehensive and structured approach to mastering competitive programming using C++. The scope of work included:

1. **Core Curriculum:**
    - Data Structures: Understanding and implementing fundamental data structures such as arrays, linked lists, stacks, queues, trees, and graphs.
    - Algorithms: Learning and applying key algorithms including sorting, searching, dynamic programming, greedy algorithms, and graph algorithms.
    - Problem-Solving Techniques: Developing techniques for solving complex problems efficiently, with an emphasis on analysing time and space complexity.
2. **Advanced Topics:**
    - Graph Theory: In-depth exploration of graph-related algorithms and their applications in solving real-world problems.
    - String Algorithms: Mastery of algorithms related to string manipulation, pattern matching, and text processing.
3. **Practical Application:**

- Project Implementation: Completing a project that demonstrates the practical application of the concepts learned, such as the N-Queen problem with visual implementation.

4. **Evaluation and Feedback:**
    - Practice Contests: Participation in practice contests to simulate real coding competition environments.
    - Regular Assessments: Periodic assessments to monitor progress and provide feedback for continuous improvement.

The program's scope ensured that students were not only equipped with the necessary technical skills but also prepared for real-world coding challenges and technical interviews.

1.3 Importance and Applicability

Competitive programming plays a crucial role in the field of computer science, offering numerous benefits. It enhances problem-solving abilities, improves coding efficiency, and prepares students for real-world challenges. The knowledge gained from this program is directly applicable to various domains, including software development, algorithm design, and system optimization. Moreover, the skills acquired are valuable for participating in coding competitions such as Codeforces, LeetCode, and CodeChef, which are important platforms for showcasing talent and securing job offers from top tech companies.

1.4 Scope and Relevance

The Placement Enhancement Program (PEP) covered an extensive range of topics, ensuring a well-rounded understanding of both foundational and advanced concepts in competitive programming. The curriculum was meticulously designed to include:

1. **Basic Data Structures:**
    - Arrays and Linked Lists: Essential for storing and managing collections of data, these structures form the building blocks for more complex data manipulation.

- Stacks and Queues: Key to understanding sequential data processing, these structures are widely used in algorithms involving recursion, backtracking, and breadth-first search.

2. **Advanced Data Structures:**
    - Trees and Graphs: Critical for representing hierarchical data and solving problems related to networking, databases, and web development. Graphs are indispensable in solving problems that involve connections and pathways.
    - Heaps and Hash Tables: Fundamental for optimizing search operations and implementing priority queues, heaps and hash tables play a crucial role in performance-critical applications.
3. **Algorithms:**
    - Sorting and Searching: Mastery of these algorithms is essential for optimizing data retrieval and organization, which are vital in large-scale applications.
    - Dynamic Programming and Greedy Algorithms: These techniques are crucial for solving optimization problems efficiently, where the goal is to find the best possible solution among many.
    - Graph Algorithms: Understanding algorithms like Dijkstra's, Kruskal's, and Bellman-Ford is essential for solving problems related to shortest paths, spanning trees, and network flow.
4. **Advanced Problem-Solving Techniques:**
    - Complexity Analysis: A deep understanding of time and space complexity is vital for designing efficient algorithms that can manage large inputs without compromising performance.
    - String Manipulation: Advanced string algorithms are indispensable in fields such as bioinformatics, cryptography, and text processing.

Relevance

The relevance of these topics in modern software development is profound. As software systems become increasingly complex and data-intensive, the ability to design efficient algorithms and optimize code is more critical than ever. Here is how the relevance of these concepts is manifested:

1. **Efficient Algorithm Design:**
    - The knowledge gained from the program enables students to craft algorithms that solve problems faster and with fewer resources, which is crucial in scenarios where performance is key, such as real-time systems, financial modelling, and large-scale data analysis.
2. **Code Optimization:**
    - By understanding and applying the principles of data structures and algorithms, developers can optimize their code to run more efficiently, reducing computational overhead and improving application performance. This is particularly important in environments with limited resources, such as embedded systems and mobile applications.
3. **Complex Problem Solving:**
    - The program equips students with the skills to tackle a broad spectrum of problems, from routine tasks like data sorting and searching to intricate challenges such as network optimization and machine learning. This deep algorithmic thinking is crucial in fields like artificial intelligence, cybersecurity, and game development.
4. **Preparation for Technical Interviews:**
    - Mastery of these topics is not only essential for academic success but also for excelling in technical interviews, where a strong grasp of data structures and algorithms is often a determining factor in securing a job in top tech companies.

Chapter 2: LEARNING AND IMPLEMENTATION

2.1 Introduction to Data Structures

Data structures are fundamental components in computer science, providing a way to organize and store data efficiently. The study of data structures included understanding their properties, operations, and applications. The key data structures covered in this program were arrays, linked lists, stacks, queues, trees, graphs, heaps, and hash tables.

2.2 Arrays

Arrays are one of the most basic data structures, providing a way to store elements in a contiguous block of memory. The program covered various operations on arrays, including searching, sorting, and prefix sums. Sorting algorithms such as quicksort, merge sort, and heapsort were studied in detail, along with their time and space complexities

Basic code of array and its implementation:

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

// Linear Search
int linearSearch(int arr[], int n, int x)
{
    for (int i = 0; i < n; i++)
    {
        if (arr[i] == x)
        {
            return i; // Return the index if element is found
        }
    }
    return -1; // Return -1 if the element is not found
}

// Reverse the Array
void reverseArray(int arr[], int n)
{
    int start = 0;
    int end = n - 1;
    while (start < end)
    {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

// Bubble Sort
void bubbleSort(int arr[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
```

```
        for (int j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main()
{
    int arr[5] = {40, 10, 30, 20, 50};
    int n = sizeof(arr) / sizeof(arr[0]);

    // 1. Linear Search
    int x = 30;
    int searchResult = linearSearch(arr, n, x);
    if (searchResult != -1)
        cout << "Element " << x << " found at index " << searchResult << endl;
    else
        cout << "Element " << x << " not found" << endl;

    // 2. Reverse the Array
    reverseArray(arr, n);
    cout << "Reversed array: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;

    // 3. Bubble Sort
    bubbleSort(arr, n);
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;

    return 0;
}
```

2.3 Strings

Strings are sequences of characters, and string manipulation is a common task in programming. The program focused on pattern matching algorithms, such as the Knuth-Morris-Pratt (KMP) algorithm and hashing techniques for efficient string comparisons. The applications of these techniques in real-world problems were also explored.

Basic code of string's implementation:

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

// Function to create the Longest Prefix Suffix (LPS) array
void computeLPSArray(string pattern, int m, vector<int> &lps)
{
    int length = 0;
    lps[0] = 0;
    int i = 1;

    while (i < m)
    {
        if (pattern[i] == pattern[length])
        {
            length++;
            lps[i] = length;
            i++;
        }
        else
        {
            if (length != 0)
            {
                length = lps[length - 1];
            }
            else
            {
                lps[i] = 0;
                i++;
            }
        }
    }
}

// KMP Pattern Matching Algorithm
void KMPSearch(string text, string pattern)
```

```
{
    int n = text.size();
    int m = pattern.size();

    vector<int> lps(m);
    computeLPSArray(pattern, m, lps);

    int i = 0;
    int j = 0;
    while (i < n)
    {
        if (pattern[j] == text[i])
        {
            i++;
            j++;
        }

        if (j == m)
        {
            cout << "Pattern found at index " << i - j << endl;
            j = lps[j - 1];
        }
        else if (i < n && pattern[j] != text[i])
        {
            if (j != 0)
            {
                j = lps[j - 1];
            }
            else
            {
                i++;
            }
        }
    }
}

int main()
{
    string text = "ABABDABACDABABCABAB";
    string pattern = "ABABCABAB";

    KMPSearch(text, pattern);

    return 0;
}
```

2.4 Linked Lists

Linked lists are dynamic data structures that allow efficient insertion and deletion of elements. The program covered single, double, and circular linked lists, along with various operations such as traversal, insertion, deletion, and reversal. The advantages and disadvantages of linked lists compared to arrays were discussed.

Basic code of Linked List's implementation:

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

struct Node
{
    int data;
    Node *next;
    Node(int val) : data(val), next(nullptr) {}
};

class LinkedList
{
private:
    Node *head;

public:
    LinkedList() : head(nullptr) {}

    void insertAtBeginning(int data)
    {
        Node *newNode = new Node(data);
        newNode->next = head;
        head = newNode;
    }
    void deleteNode(int key)
    {
        Node *temp = head, *prev = nullptr;
        if (temp != nullptr && temp->data == key)
        {
            head = temp->next;
            delete temp;
            return;
        }
        while (temp != nullptr && temp->data != key)
        {
            prev = temp;
```

```
            temp = temp->next;
        }
        if (temp == nullptr)
            return;
        prev->next = temp->next;
        delete temp;
    }
    void reverse()
    {
        Node *prev = nullptr, *current = head, *next = nullptr;
        while (current != nullptr)
        {
            next = current->next;
            current->next = prev;
            prev = current;
            current = next;
        }
        head = prev;
    }
    void printList()
    {
        Node *temp = head;
        while (temp != nullptr)
        {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};
int main()
{
    LinkedList list;
    list.insertAtBeginning(10);
    list.insertAtBeginning(20);
    list.insertAtBeginning(30);
    list.printList();

    list.deleteNode(20);
    list.printList();

    list.reverse();
    list.printList();

    return 0;
}
```

2.5 Stacks and Queues

Stacks and queues are linear data structures with specific rules for insertion and deletion. The program covered their implementation using arrays and linked lists, along with applications such as expression evaluation and task scheduling. Monotonic stacks and queues were also introduced as advanced variations with specific use cases.

Basic code of Stack and Queue's implementation:

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

class Stack
{
private:
    int top;
    int capacity;
    int *stack;

public:
    Stack(int size)
    {
        capacity = size;
        stack = new int[capacity];
        top = -1;
    }
    void push(int x)
    {
        if (top == capacity - 1)
        {
            cout << "Stack Overflow" << endl;
            return;
        }
        stack[++top] = x;
    }
    void pop()
    {
        if (top == -1)
        {
            cout << "Stack Underflow" << endl;
            return;
        }
        top--;
```

```
    }
    int peek()
    {
        return (top == -1) ? -1 : stack[top];
    }
    bool isEmpty()
    {
        return top == -1;
    }
    void display()
    {
        for (int i = 0; i <= top; i++)
        {
            cout << stack[i] << " ";
        }
        cout << endl;
    }
    ~Stack()
    {
        delete[] stack;
    }
};
// Queue Implementation using Array
class Queue
{
private:
    int front, rear, capacity;
    int *queue;
public:
    Queue(int size)
    {
        capacity = size;
        queue = new int[capacity];
        front = 0;
        rear = -1;
    }
    void enqueue(int x)
    {
        if (rear == capacity - 1)
        {
            cout << "Queue Overflow" << endl;
            return;
        }
        queue[++rear] = x;
    }
    void dequeue()
    {
        if (front > rear)
```

```
        {
            cout << "Queue Underflow" << endl;
            return;
        }
        front++;
    }
    int peek()
    {
        return (front > rear) ? -1 : queue[front];
    }
    bool isEmpty()
    {
        return front > rear;
    }
    void display()
    {
        for (int i = front; i <= rear; i++)
        {
            cout << queue[i] << " ";
        }
        cout << endl;
    }
    ~Queue()
    {
        delete[] queue;
    }
};
int main()
{
    Stack s(5);
    s.push(10);
    s.push(20);
    s.push(30);
    s.display();
    s.pop();
    s.display();

    Queue q(5);
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();
    q.dequeue();
    q.display();

    return 0;
}
```

2.6 Trees

Trees are hierarchical data structures that are widely used in various applications, such as database indexing and expression parsing. The program focused on binary trees, binary search trees, segment trees, and Fenwick trees. Operations such as insertion, deletion, and traversal were covered in detail, along with the properties and applications of each type of tree.

Basic code of Tree's implementation:

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

struct Node
{
    int data;
    Node *left;
    Node *right;
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class BinarySearchTree
{
private:
    Node *root;

    Node *insert(Node *node, int data)
    {
        if (node == nullptr)
        {
            return new Node(data);
        }
        if (data < node->data)
        {
            node->left = insert(node->left, data);
        }
        else
        {
            node->right = insert(node->right, data);
        }
        return node;
    }
```

```
    Node *deleteNode(Node *node, int key)
    {
        if (node == nullptr)
            return node;
        if (key < node->data)
        {
            node->left = deleteNode(node->left, key);
        }
        else if (key > node->data)
        {
            node->right = deleteNode(node->right, key);
        }
        else
        {
            if (node->left == nullptr)
                return node->right;
            else if (node->right == nullptr)
                return node->left;
            Node *temp = node->right;
            while (temp && temp->left != nullptr)
                temp = temp->left;
            node->data = temp->data;
            node->right = deleteNode(node->right, temp->data);
        }
        return node;
    }

    void inorder(Node *node)
    {
        if (node)
        {
            inorder(node->left);
            cout << node->data << " ";
            inorder(node->right);
        }
    }

public:
    BinarySearchTree() : root(nullptr) {}

    void insert(int data)
    {
        root = insert(root, data);
    }

    void deleteNode(int key)
    {
        root = deleteNode(root, key);
```

```
    }

    void inorderTraversal()
    {
        inorder(root);
        cout << endl;
    }
};

int main()
{
    BinarySearchTree bst;
    bst.insert(50);
    bst.insert(30);
    bst.insert(20);
    bst.insert(40);
    bst.insert(70);
    bst.insert(60);
    bst.insert(80);

    cout << "Inorder traversal: ";
    bst.inorderTraversal();

    bst.deleteNode(20);
    cout << "Inorder traversal after deleting 20: ";
    bst.inorderTraversal();

    bst.deleteNode(30);
    cout << "Inorder traversal after deleting 30: ";
    bst.inorderTraversal();

    bst.deleteNode(50);
    cout << "Inorder traversal after deleting 50: ";
    bst.inorderTraversal();

    return 0;
}
```

2.7 Graphs

Graphs are versatile data structures that represent relationships between objects. The program covered graph representation techniques, such as adjacency matrices and adjacency lists, and explored algorithms for traversing graphs, including Breadth-First Search (BFS) and Depth-First Search (DFS). Shortest path algorithms like Dijkstra's and Bellman-Ford were also studied, along with minimum spanning tree algorithms such as Prim's and Kruskal's.

Basic code of Graph's implementation:

```
#include <iostream>
#include <vector>
#include <queue>
#include <limits>

using namespace std;

class Graph
{
private:
    int V;
    vector<vector<pair<int, int>>> adj;

public:
    Graph(int vertices) : V(vertices)
    {
        adj.resize(V);
    }

    void addEdge(int u, int v, int w)
    {
        adj[u].emplace_back(v, w);
        adj[v].emplace_back(u, w);
    }

    void DFSUtil(int v, vector<bool> &visited)
    {
        visited[v] = true;
        cout << v << " ";
        for (auto &edge : adj[v])
        {
            if (!visited[edge.first])
            {
                DFSUtil(edge.first, visited);
            }
```

```
        }
    }

    void DFS(int start)
    {
        vector<bool> visited(V, false);
        DFSUtil(start, visited);
        cout << endl;
    }

    void BFS(int start)
    {
        vector<bool> visited(V, false);
        queue<int> q;
        visited[start] = true;
        q.push(start);

        while (!q.empty())
        {
            int v = q.front();
            q.pop();
            cout << v << " ";

            for (auto &edge : adj[v])
            {
                if (!visited[edge.first])
                {
                    visited[edge.first] = true;
                    q.push(edge.first);
                }
            }
        }
        cout << endl;
    }

    void Dijkstra(int src)
    {
        vector<int> dist(V, numeric_limits<int>::max());
        dist[src] = 0;
        priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;
        pq.push({0, src});

        while (!pq.empty())
        {
            int u = pq.top().second;
            pq.pop();
```

```
            for (auto &edge : adj[u])
            {
                int v = edge.first;
                int weight = edge.second;

                if (dist[u] + weight < dist[v])
                {
                    dist[v] = dist[u] + weight;
                    pq.push({dist[v], v});
                }
            }
        }

        cout << "Vertex Distance from Source " << src << ":\n";
        for (int i = 0; i < V; i++)
        {
            cout << i << "\t\t" << dist[i] << endl;
        }
    }
};

int main()
{
    Graph g(5);
    g.addEdge(0, 1, 10);
    g.addEdge(0, 4, 5);
    g.addEdge(1, 2, 1);
    g.addEdge(1, 4, 2);
    g.addEdge(2, 3, 4);
    g.addEdge(3, 0, 7);
    g.addEdge(4, 1, 3);
    g.addEdge(4, 2, 9);
    g.addEdge(4, 3, 2);

    cout << "DFS starting from vertex 0:\n";
    g.DFS(0);

    cout << "BFS starting from vertex 0:\n";
    g.BFS(0);

    g.Dijkstra(0);

    return 0;
}
```

2.8 Heaps

Heaps are specialized tree-based data structures that support efficient priority queue operations. The program covered heap implementation, heap sort, and the heapify operation. Applications of heaps in scheduling algorithms and graph algorithms were discussed.

Basic code of Heap's implementation:

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

class MinHeap
{
private:
    vector<int> heap;

    void heapifyUp(int index)
    {
        while (index > 0 && heap[parent(index)] > heap[index])
        {
            swap(heap[parent(index)], heap[index]);
            index = parent(index);
        }
    }

    void heapifyDown(int index)
    {
        int left = leftChild(index);
        int right = rightChild(index);
        int smallest = index;

        if (left < heap.size() && heap[left] < heap[smallest])
        {
            smallest = left;
        }
        if (right < heap.size() && heap[right] < heap[smallest])
        {
            smallest = right;
        }
        if (smallest != index)
        {
            swap(heap[index], heap[smallest]);
            heapifyDown(smallest);
        }
    }
```

```
    int parent(int index)
    {
        return (index - 1) / 2;
    }

    int leftChild(int index)
    {
        return 2 * index + 1;
    }

    int rightChild(int index)
    {
        return 2 * index + 2;
    }

public:
    void insert(int value)
    {
        heap.push_back(value);
        heapifyUp(heap.size() - 1);
    }

    void deleteMin()
    {
        if (heap.empty())
            return;
        heap[0] = heap.back();
        heap.pop_back();
        heapifyDown(0);
    }

    int getMin()
    {
        return heap.empty() ? -1 : heap[0];
    }

    void heapSort()
    {
        vector<int> sorted;
        while (!heap.empty())
        {
            sorted.push_back(getMin());
            deleteMin();
        }
        for (int val : sorted)
        {
            cout << val << " ";
```

```
        }
        cout << endl;
    }

    void display()
    {
        for (int val : heap)
        {
            cout << val << " ";
        }
        cout << endl;
    }
};

int main()
{
    MinHeap minHeap;

    minHeap.insert(3);
    minHeap.insert(1);
    minHeap.insert(6);
    minHeap.insert(5);
    minHeap.insert(2);
    minHeap.insert(4);

    cout << "Min Heap elements: ";
    minHeap.display();

    cout << "Minimum element: " << minHeap.getMin() << endl;

    minHeap.deleteMin();
    cout << "Min Heap after deleting minimum element: ";
    minHeap.display();

    cout << "Heap sort result: ";
    minHeap.heapSort();

    return 0;
}
```

2.9 Hashing

Hashing is a technique used to map data to a fixed-size table, enabling efficient data retrieval. The program covered hash tables, collision resolution techniques, and the applications of hashing in various domains, such as database indexing and caching.

Basic code of Hashing's implementation:

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

class HashTable
{
private:
    vector<list<int>> table;
    int size;

public:
    HashTable(int s) : size(s)
    {
        table.resize(size);
    }

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

    void insert(int key)
    {
        int index = hashFunction(key);
        table[index].push_back(key);
    }

    void remove(int key)
    {
        int index = hashFunction(key);
        table[index].remove(key);
    }

    bool search(int key)
    {
        int index = hashFunction(key);
        for (int k : table[index])
```

```
        {
            if (k == key)
            {
                return true;
            }
        }
        return false;
    }

    void display()
    {
        for (int i = 0; i < size; i++)
        {
            cout << i << ": ";
            for (int k : table[i])
            {
                cout << k << " -> ";
            }
            cout << "nullptr" << endl;
        }
    }
};

int main()
{
    HashTable ht(10);

    ht.insert(15);
    ht.insert(11);
    ht.insert(27);
    ht.insert(8);
    ht.insert(12);
    ht.insert(18);

    cout << "Hash Table contents:\n";
    ht.display();

    cout << "Searching for 27: " << (ht.search(27) ? "Found" : "Not Found") <<
endl;
    ht.remove(27);
    cout << "After removing 27:\n";
    ht.display();

    return 0;
}
```

2.10 Tries and Dynamic Arrays

Tries, also known as prefix trees, are specialized data structures used for efficient string search operations. The program covered the implementation and applications of tries in search problems. Dynamic arrays, such as vectors, sets, and maps in STL (C++), were also introduced, along with their advantages over static arrays.

Basic code of Tries's implementation:

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

class TrieNode
{
public:
    TrieNode *children[26]; // 26 letters of the alphabet
    bool isEndOfWord;

    TrieNode()
    {
        isEndOfWord = false;
        for (int i = 0; i < 26; i++)
        {
            children[i] = nullptr;
        }
    }
};

class Trie
{
private:
    TrieNode *root;

public:
    Trie()
    {
        root = new TrieNode();
    }

    void insert(const string &word)
    {
        TrieNode *current = root;
        for (char c : word)
        {
            int index = c - 'a';
            if (!current->children[index])
```

```
            {
                current->children[index] = new TrieNode();
            }
            current = current->children[index];
        }
        current->isEndOfWord = true;
    }

    bool search(const string &word)
    {
        TrieNode *current = root;
        for (char c : word)
        {
            int index = c - 'a';
            if (!current->children[index])
            {
                return false;
            }
            current = current->children[index];
        }
        return current->isEndOfWord;
    }
    void display(TrieNode *node, string prefix)
    {
        if (node->isEndOfWord)
        {
            cout << prefix << endl;
        }
        for (int i = 0; i < 26; i++)
        {
            if (node->children[i])
            {
                display(node->children[i], prefix + char(i + 'a'));
            }
        }
    }
    void displayWords()
    {
        display(root, "");
    }
};
int main()
{
    Trie trie;
    trie.insert("apple");
    trie.insert("app");
    trie.insert("bat");
    trie.insert("ball");
```

```
    trie.insert("batman");
    cout << "Searching for 'app': " << (trie.search("app") ? "Found" : "Not
Found") << endl;
    cout << "Searching for 'banana': " << (trie.search("banana") ? "Found" :
"Not Found") << endl;

    cout << "Words in the trie:\n";
    trie.displayWords();

    return 0;
}
```

Basic code of Dynamic Array's implementation:

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

int main()
{
    vector<int> vec;
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    cout << "Vector elements: ";
    for (int val : vec)
    {
        cout << val << " ";
    }
    cout << endl;
    set<int> s;
    s.insert(5);
    s.insert(10);
    s.insert(5);
    cout << "Set elements: ";
    for (int val : s)
    {
        cout << val << " ";
    }
    cout << endl;
    return 0;
}
```

2.11 LeetCode Problems Solved

During the Placement Enhancement Program (PEP) classes, I solved various LeetCode problems that enhanced my understanding of key data structures and algorithms. These problems were crucial for improving my problem-solving skills and preparing for technical interviews. Below is a summary of some of the LeetCode problems I worked on:

1. Two Sum

- **Problem Type:** Array, Hash Map
- **Difficulty:** Easy
- **Concepts Covered:** Hashing, Linear Search

```cpp
#include <vector>
#include <unordered_map>
using namespace std;

vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> hash_map;
    for (int i = 0; i < nums.size(); i++) {
        int complement = target - nums[i];
        if (hash_map.find(complement) != hash_map.end()) {
            return {hash_map[complement], i};
        }
        hash_map[nums[i]] = i;
    }
    return {};
}
```

2. Longest Substring Without Repeating Characters

- **Problem Type:** String, Sliding Window
- **Difficulty:** Medium
- **Concepts Covered:** Sliding Window, Hash Map

```cpp
#include <string>
#include <unordered_map>
using namespace std;

int lengthOfLongestSubstring(string s)
{
    unordered_map<char, int> char_map;
    int left = 0, max_len = 0;

    for (int right = 0; right < s.size(); right++)
    {
```

```
        if (char_map.find(s[right]) != char_map.end())
        {
            left = max(char_map[s[right]] + 1, left);
        }
        char_map[s[right]] = right;
        max_len = max(max_len, right - left + 1);
    }

    return max_len;
}
```

3. Merge Intervals

- **Problem Type:** Array, Sorting
- **Difficulty:** Medium
- **Concepts Covered:** Sorting, Interval Merging

```
#include <vector>
#include <algorithm>
using namespace std;

vector<vector<int>> merge(vector<vector<int>> &intervals)
{
    if (intervals.empty())
        return {};

    sort(intervals.begin(), intervals.end());
    vector<vector<int>> merged;
    merged.push_back(intervals[0]);

    for (int i = 1; i < intervals.size(); i++)
    {
        if (merged.back()[1] >= intervals[i][0])
        {
            merged.back()[1] = max(merged.back()[1], intervals[i][1]);
        }
        else
        {
            merged.push_back(intervals[i]);
        }
    }

    return merged;
}
```

4. Kth Largest Sum in a Binary Tree

- **Problem Type:** Binary Tree, Priority Queue
- **Difficulty:** Hard
- **Concepts Covered:** Tree Traversal, Heap

```cpp
#include <queue>
using namespace std;

struct TreeNode {
    int val;
    TreeNode *left, *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    int kthLargestSum(TreeNode* root, int k) {
        priority_queue<int, vector<int>, greater<int>> min_heap;
        dfs(root, min_heap, k);
        return min_heap.size() == k ? min_heap.top() : -1;
    }

private:
    int dfs(TreeNode* node, priority_queue<int, vector<int>, greater<int>>&
min_heap, int k) {
        if (!node) return 0;

        int left_sum = dfs(node->left, min_heap, k);
        int right_sum = dfs(node->right, min_heap, k);
        int total_sum = node->val + left_sum + right_sum;

        min_heap.push(total_sum);
        if (min_heap.size() > k) {
            min_heap.pop();
        }

        return total_sum;
    }
};
```

5. Robot Collisions

- **Problem Type:** Simulation, Stack
- **Difficulty:** Medium
- **Concepts Covered:** Stack, Collision Detection

```cpp
#include <vector>
using namespace std;

vector<int> robotCollisions(vector<int> &robots)
{
    vector<int> stack;

    for (int robot : robots)
    {
        while (!stack.empty() && stack.back() > 0 && robot < 0)
        {
            if (stack.back() + robot == 0)
            {
                stack.pop_back();
                robot = 0;
                break;
            }
            else if (stack.back() + robot < 0)
            {
                stack.pop_back();
            }
            else
            {
                robot = 0;
                break;
            }
        }
        if (robot != 0)
        {
            stack.push_back(robot);
        }
    }

    return stack;
}
```

Chapter 3: N-QUEEN PROBLEM PROJECT

3.1 Introduction to the N-Queen Problem

The N-Queen problem is a classic example of a constraint satisfaction problem, where the objective is to place N queens on an N×N chessboard such that no two queens threaten each other. This project aimed to provide a visual implementation of the N-Queen problem, demonstrating how backtracking can be used to find all possible solutions.

3.2 Problem Statement and Requirements

The problem statement was to design a program that can solve the N-Queen problem for any given value of N. The program was required to visualize the placement of queens on the board and highlight the valid and invalid configurations. The solution needed to be efficient, leveraging backtracking to explore the solution space systematically.

3.3 Algorithm and Implementation

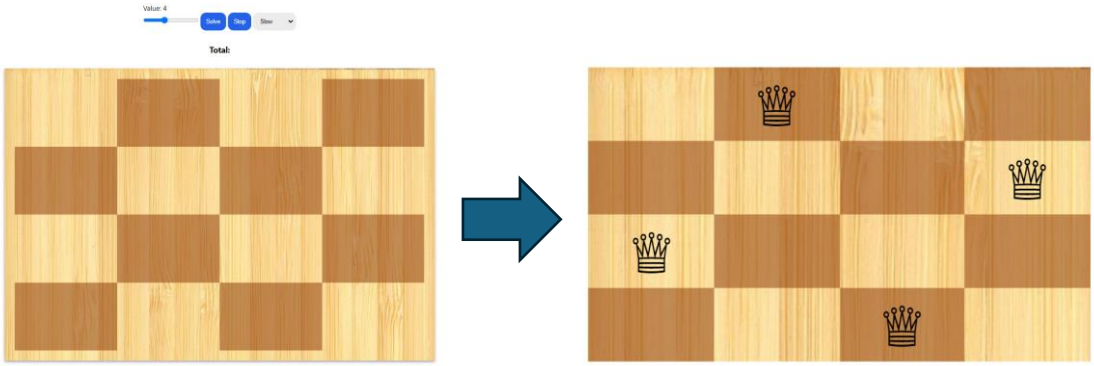
The algorithm used for solving the N-Queen problem was based on backtracking. The program started by placing a queen in the first row and then recursively tried to place queens in the subsequent rows. If a valid configuration was found, it was recorded; otherwise, the program backtracked to try a different configuration. The implementation involved checking for conflicts between queens by verifying that no two queens shared the same row, column, or diagonal.

```
bool isSafe(int board[N][N], int row, int col)
{
    for (int i = 0; i < col; i++)
        if (board[row][i])
            return false;

    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;
```

```
    for (int i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

bool solveNQUtil(int board[N][N], int col)
{
    if (col >= N)
        return true;

    for (int i = 0; i < N; i++)
    {
        if (isSafe(board, i, col))
        {
            board[i][col] = 1;

            if (solveNQUtil(board, col + 1))
                return true;

            board[i][col] = 0;
        }
    }

    return false;
}
```

3.4 Visual Implementation

To enhance understanding, a visual representation of the N-Queen problem was developed. The interface displayed the chessboard and dynamically updated it as the algorithm progressed. The placement of queens was shown in real-time, with invalid configurations highlighted. This visualization helped in comprehending the backtracking process and the challenges involved in finding all solutions.



3.5 Testing and Results

The program was tested for various values of N, ranging from 4 to 12. The results showed that the algorithm efficiently found all possible solutions for each value of N. The visualization provided clear insights into the number of solutions and the complexity of the problem as N increased. The project demonstrated the power of backtracking in solving constraint satisfaction problems and highlighted the importance of algorithmic efficiency in such problems.

```
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            cout << " " << board[i][j] << " ";
        cout << endl;
    }
}

bool solveNQ()
{
    int board[N][N] = {0};

    if (solveNQUtil(board, 0) == false)
    {
        cout << "Solution does not exist";
        return false;
    }

    printSolution(board);
    return true;
}
```

The code snippets provided above illustrate the key functions used in the N-Queen problem. The isSafe function checks whether a queen can be placed on a given square, while the `solveNQUntil` function recursively attempts to place queens on the board. The `printSolution` function outputs the final arrangement of the queens on the chessboard.

3.6 Complete code of N Queen Solver:

```
#include <iostream>
#include <vector>

using namespace std;

bool isSafe(const vector<vector<int>> &board, int row, int col, int N)
{
    for (int i = 0; i < row; i++)
    {
        if (board[i][col] == 1)
        {
            return false;
        }
    }
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
    {
        if (board[i][j] == 1)
        {
            return false;
        }
    }
    for (int i = row, j = col; i >= 0 && j < N; i--, j++)
    {
        if (board[i][j] == 1)
        {
            return false;
        }
    }
    return true;
}

bool solveNQueens(vector<vector<int>> &board, int row, int N)
{
    if (row >= N)
    {
        return true;
    }
    for (int col = 0; col < N; col++)
    {
        if (isSafe(board, row, col, N))
        {
            board[row][col] = 1;
            if (solveNQueens(board, row + 1, N))
            {
                return true;
            }
```

```
            board[row][col] = 0;
        }
    }
    return false;
}

void printSolution(const vector<vector<int>> &board)
{
    for (const auto &row : board)
    {
        for (int cell : row)
        {
            cout << (cell == 1 ? "Q " : ". ");
        }
        cout << endl;
    }
    cout << endl;
}

int main()
{
    int N;
    cout << "Enter the number of queens (N): ";
    cin >> N;
    vector<vector<int>> board(N, vector<int>(N, 0));
    if (solveNQueens(board, 0, N))
    {
        cout << "One of the solutions is:\n";
        printSolution(board);
    }
    else
    {
        cout << "No solution exists for " << N << " queens.\n";
    }
    return 0;
}
```

3.7 Challenges and Learnings

During the implementation of the N-Queen problem, several challenges were encountered. These included optimizing the algorithm for larger values of N and ensuring the visualization accurately represented the recursive nature of backtracking. Through these challenges, valuable lessons were learned about algorithm optimization, debugging complex recursive functions, and the importance of visualization in understanding algorithm behaviour.

Chapter 4: CERTIFICATION IN MONGODB

4.1 Introduction to MongoDB

As part of the Placement Enhancement Program, students were required to obtain a certification in MongoDB. MongoDB is a widely-used NoSQL database that offers flexible schema design and scalability. Understanding its fundamentals is crucial for modern web development and data management.

4.2 Course Overview and Learning Objectives

The MongoDB certification course covered various topics, including the basics of NoSQL databases, CRUD operations, indexing, aggregation framework, and data modelling. The primary learning objectives were to understand how to interact with MongoDB, design efficient data models, and perform advanced queries to retrieve and manipulate data effectively.

4.3 Practical Implementation and Projects

Throughout the course, hands-on exercises and projects were assigned to reinforce learning. Students practiced creating and managing databases, performing queries, and using the aggregation framework to analyse data. A project that involved building a simple application using MongoDB was also completed, showcasing the integration of the database with a web application.

4.4 Certificate

Chapter 5: CONCLUSION

5.1 Summary of Learning Outcomes

The Placement Enhancement Program (PEP) provided a comprehensive and immersive learning experience that significantly enhanced my programming skills and deepened my understanding of data structures and algorithms. Over the course of two months, I engaged with a wide array of topics, from foundational concepts to advanced techniques, allowing me to develop a solid grasp of essential programming principles.

This program was particularly beneficial in bridging the gap between theoretical knowledge and practical implementation. The hands-on projects, especially the N-Queen problem, enabled me to apply theoretical concepts in real-world scenarios, solidifying my understanding and reinforcing my coding skills. The collaborative environment fostered by the program encouraged peer learning and facilitated valuable discussions, further enhancing my learning experience.

I also gained insight into effective problem-solving strategies and the importance of algorithmic thinking. Learning to break down complex problems into manageable parts has equipped me with the skills necessary to tackle challenges in competitive programming and technical interviews. Overall, the PEP has laid a strong foundation for my future endeavours in the field of computer science.

5.2 Future Applications

The skills and knowledge acquired during the Placement Enhancement Program will be instrumental in shaping my career. I plan to leverage my understanding of data structures and algorithms in software development roles, where efficiency and optimization are paramount. The ability to choose the appropriate data structures and algorithms will not only improve the performance of applications I develop but also contribute to scalable and maintainable codebases.

In addition, I aim to participate in coding competitions to further refine my problem-solving abilities. Engaging in these contests will help me stay sharp and continuously improve my coding skills, as well as adapt to new challenges in real time. Competitions also provide an excellent platform for networking with other programmers, sharing knowledge, and gaining exposure to different approaches to problem-solving.

Furthermore, the certification in MongoDB that I obtained during the PEP will enhance my capabilities in database management. With the growing demand for data-driven applications, having proficiency in NoSQL databases like MongoDB will allow me to design and implement robust data storage solutions. This skill will be particularly valuable in projects that require flexibility in data modelling and scalability, positioning me as a more versatile developer in a competitive job market.

5.3 Acknowledgments

I would like to express my heartfelt gratitude to the instructors and mentors of the Placement Enhancement Program for their unwavering guidance and support throughout the training. Their wealth of knowledge, patience, and dedication to teaching have been instrumental in my growth as a programmer. The constructive feedback and encouragement provided by my instructors were invaluable in helping me achieve my learning goals and overcome challenges.

I also extend my thanks to my peers, who contributed to a collaborative and enriching learning environment. The discussions, brainstorming sessions, and shared experiences fostered a sense of community that made the learning process enjoyable and effective. Collaborating with others not only helped me gain different perspectives on various topics but also strengthened my communication and teamwork skills, which are essential in any professional setting.

Lastly, I would like to acknowledge the resources provided by the Placement Enhancement Program, including access to learning materials, coding platforms, and the opportunity to work on real-world projects. These resources have equipped me with the tools needed to succeed in my future endeavours. I look forward to applying what I have learned and continuing my journey in the field of computer science with confidence and enthusiasm.