MANGALAYATAN UNIVERSITY
Learn Today to Lead Tomorrow
NAAC GRADE A+ Accredited University

BACHELOR OF COMPUTER APPLICATIONS (BCA)

CSB-1151 (Problem Solving Using C Lab)

Block 1: Programming and Algorithm Lab

Unit 2: Advanced Algorithms and Data Structures

Structure

2.0 INTRODUCTION

The "Problem Solving Using 'C' Lab" course is designed to immerse students in the practical application of advanced algorithms and data structures, a critical component for tackling complex computational problems. This course provides a hands-on approach to learning, allowing students to implement and experiment with sophisticated data structures such as trees, graphs, and hash tables directly in C. It also covers the development of efficient algorithms essential for sorting, searching, and optimization tasks, utilizing techniques like dynamic programming and greedy algorithms. By focusing on code performance optimization and memory management, this lab not only reinforces theoretical knowledge but also enhances practical problem-solving skills. This foundational experience is crucial for students aiming to excel in advanced computer science fields, where efficient algorithm design and implementation are paramount.

This lab course will be discussing the separate step compilation process of language C.

2.1 OBJECTIVES

After completing this lab course, you will be able to:

Here are five objectives related to the subject "Problem Solving Using 'C' Lab" focusing on advanced algorithms and data structures:

1. Implement and analyze complex data structures such as trees, graphs, and hash tables in C.
2. Develop efficient algorithms for sorting, searching, and optimization problems using C.
3. Solve real-world problems by applying dynamic programming and greedy algorithms in C.
4. Optimize code performance and memory usage through advanced algorithmic techniques in C.
5. Enhance problem-solving skills by debugging and refining C programs involving advanced data structures and algorithms.

2.2 ALGORITHM IMPLEMENTATION

In the "Problem Solving Using 'C' Lab," the focus on algorithm implementation is crucial for developing practical coding skills and a deep understanding of computational problem-solving. Students are tasked with translating algorithmic concepts into efficient C code, thereby learning to address a wide range of challenges. This hands-on approach includes implementing various algorithms, such as sorting, searching, and optimization techniques, which are essential for handling data efficiently. By working through real-world problems and debugging their implementations, students enhance their logical thinking, improve their code optimization skills, and gain confidence in their ability to solve complex problems programmatically. This comprehensive experience equips students with the necessary tools to tackle advanced algorithmic tasks in their future careers.

2.2.1 Sorting Algorithms: Implementation of Sorting Algorithms

Introduction to Sorting Algorithms:

Sorting algorithms are fundamental tools in computer science and programming, essential for organizing data in a specific order, such as ascending or descending. Effective sorting improves data retrieval efficiency and optimizes the performance of various computational tasks. In the "Problem Solving Using 'C' Lab" course, students will implement and analyze different sorting algorithms using the C programming language. This hands-on approach helps in understanding the underlying principles, complexities, and practical applications of sorting algorithms.

Common Sorting Algorithms:

1. **Bubble Sort**

Bubble Sort is one of the simplest sorting algorithms, which repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the list is sorted.

```
#include <stdio.h>

void bubble Sort(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[] = {64, 34, 25, 12, 22, 11, 90};
  int n = sizeof(arr)/sizeof(arr[0]);
  bubbleSort(arr, n);
  printf("Sorted array: \n");
  for (int i = 0; i < n; i++)
    printf("%d ", arr[i]);
  return 0;
}
```

2. Selection Sort

Selection Sort works by repeatedly finding the minimum element from the unsorted part of the array and putting it at the beginning.

```
#include <stdio.h>
void selection Sort(int arr[], int n) {
  for (int i = 0; i < n-1; i++) {
    int min_idx = i;
    for (int j = i+1; j < n; j++) {
      if (arr[j] < arr[min_idx])
        min_idx = j;
    }
    int temp = arr[min_idx];
    arr[min_idx] = arr[i];
    arr[i] = temp;
  }
}
int main() {
  int arr[] = {64, 25, 12, 22, 11};
```

```
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

3. Insertion Sort

Insertion Sort builds the sorted array one item at a time, with each new element being inserted into its proper place among the previously sorted elements.

```
#include <stdio.h>
void insertion Sort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    insertionSort(arr, n);
    printf("Sorted array: \n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

4. Quick Sort

Quick Sort is a highly efficient sorting algorithm that uses a divide-and-conquer approach. It selects a 'pivot' element and partitions the array into two sub-arrays according to whether they are less than or greater than the pivot.

```
#include <stdio.h>
void swap(int* a, int* b) {
   int t = *a;
   *a = *b;
   *b = t;
}
int partition (int arr[], int low, int high) {
   int pivot = arr[high];
   int i = (low - 1);
   for (int j = low; j <= high-1; j++) {
      if (arr[j] <= pivot) {
         i++;
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}
void quickSort(int arr[], int low, int high) {
   if (low < high) {
      int pi = partition(arr, low, high);

      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
   }
}
int main() {
   int arr[] = {10, 7, 8, 9, 1, 5};
   int n = sizeof(arr)/sizeof(arr[0]);
   quickSort(arr, 0, n-1);
```

```
    printf("Sorted array: \n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

2.2.2 Searching Algorithms: Implementation of Searching Algorithms

Searching algorithms are fundamental tools in computer science, enabling efficient retrieval of specific elements from a collection of data. In the context of the Problem Solving Using 'C' Lab, students delve into the implementation and analysis of various searching algorithms using the C programming language. This hands-on exploration provides students with a practical understanding of the underlying principles and applications of searching algorithms, equipping them with essential problem-solving skills.

1. Linear Search

Linear search, also known as sequential search, is a simple searching algorithm that sequentially checks each element in a collection until the target element is found or the entire collection has been traversed. It is best suited for unordered or unsorted data sets.

```c
#include <stdio.h>

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

int main() {
    int arr[] = {12, 34, 45, 67, 89};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 45;
    int result = linearSearch(arr, n, target);
    if (result != -1) {
        printf("Element %d found at index %d\n", target, result);
    } else {
        printf("Element %d not found in the array\n", target);
    }
    return 0;
}
```

2. Binary Search

Binary search is a highly efficient searching algorithm that operates on sorted collections by repeatedly dividing the search interval in half. It compares the target value with the middle element of the collection and eliminates half of the remaining elements each time.

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

```
        int binary Search(int arr[], int left, int right, int target) {
         if (right >= left) {
        int mid = left + (right - left) / 2;
         if (arr[mid] == target) {
                return mid; // Return the index if target is found
        }
      if (arr[mid] > target) {
        return binarySearch(arr, left, mid - 1, target); // Search in the left half
      }
      return binarySearch(arr, mid + 1, right, target); // Search in the right half
   }
   return -1; // Return -1 if target is not found
}
int main() {
   int arr[] = {10, 20, 30, 40, 50, 60};
   int n = sizeof(arr) / sizeof(arr[0]);
   int target = 40;
   int result = binarySearch(arr, 0, n - 1, target);
   if (result != -1) {
      printf("Element %d found at index %d\n", target, result);
   } else {
      printf("Element %d not found in the array\n", target);
   }
   return 0;
}
```

2.3 FUNCTIONS AND RECURSION

In the exploration of functions and recursion plays a pivotal role in equipping students with essential problem-solving tools. By dissecting problems into modular components and encapsulating them within functions, students learn to write efficient and reusable code, fostering good programming practices. Moreover, the introduction to recursion enhances students' understanding of problem-solving strategies by allowing them to break down complex tasks into smaller, more manageable sub problems. Through hands-on exercises and practical applications, students gain proficiency in designing recursive algorithms, honing their ability to solve a diverse array of computational challenges. This immersive experience not only deepens students' comprehension of fundamental programming concepts but also cultivates their analytical thinking skills, empowering them to approach problem-solving tasks with confidence and creativity.

2.3.1 Function Implementation: Writing and using Functions, Passing Parameters

In the realm of the Problem Solving Using 'C' Lab, mastering function implementation is pivotal for developing robust and modular code structures. Functions encapsulate specific tasks, allowing for code reuse and enhancing readability and maintainability. Through hands-on exercises, students learn to write, utilize, and pass parameters to functions effectively, honing their programming skills and problem-solving abilities.

Writing Functions in C

In C programming, functions are defined using the following syntax:

```
return_type function_name(parameters)
{
   // Function body
}
```

Where return_type specifies the data type of the value returned by the function, function_name is the name of the function, and parameters are the input values passed to the function.

Example:

```
#include <stdio.h>
// Function to calculate the sum of two integers
int sum(int a, int b) {
   return a + b;
}
int main() {
   int x = 10, y = 20;
   int result = sum(x, y);
   printf("Sum of %d and %d is %d\n", x, y, result);
   return 0;
}
```

Passing Parameters to Functions:

Parameters can be passed to functions in C by value or by reference. When passed by value, a copy of the parameter's value is passed to the function, while passing by reference allows the function to modify the original parameter's value.

Example

```
#include <stdio.h>
// Function to swap two integers using pass by reference
void swap(int *a, int *b) {
   int temp = *a;
```

```
    *a = *b;
    *b = temp;
}
int main() {
    int x = 10, y = 20;
    printf("Before swapping: x = %d, y = %d\n", x, y);
    swap(&x, &y); // Passing addresses of x and y
    printf("After swapping: x = %d, y = %d\n", x, y);
    return 0;
}
```

2.3.2 Recursive Functions: Implementing Basic Recursion Problems

Understanding and implementing recursive functions is a cornerstone of problem-solving in computer science, and it plays a significant role in the Problem Solving Using 'C' Lab. Recursive functions offer a powerful and elegant solution to problems that can be broken down into smaller, similar sub problems. Through hands-on exercises and practical applications, students delve into the implementation of recursive algorithms, honing their skills in recursion and strengthening their ability to solve complex computational problems efficiently.

Basic Concepts of Recursion

In C programming, a recursive function is a function that calls itself directly or indirectly in order to solve a problem. This approach involves breaking down the problem into smaller instances of the same problem until a base case is reached, at which point the function stops calling itself and returns a result. Understanding the termination condition, or base case, is crucial to prevent infinite recursion.

Example:

```
#include <stdio.h>
// Recursive function to calculate factorial
int factorial(int n) {
    // Base case: factorial of 0 is 1
    if (n == 0) {
        return 1;
    }
    // Recursive case: n! = n * (n-1)!
    else {
        return n * factorial(n - 1);
    }
}
```

```
int main() {
    int n = 5;
    int result = factorial(n);
    printf("Factorial of %d is %d\n", n, result);
    return 0;
}
```

Understanding Recursion in Problem Solving

Through exercises such as factorial calculation, Fibonacci sequence generation, and binary tree traversal, students grasp the concept of recursion and its application in solving a variety of problems. They learn to analyze problems, identify recursive patterns, and implement recursive solutions efficiently. Moreover, recursive thinking enhances students' problem-solving skills by encouraging them to break down complex problems into simpler, more manageable sub problems.

2.4 ARRAY AND STRINGS

In the study of arrays and strings serves as a cornerstone for mastering data manipulation and algorithmic problem-solving. Arrays, as contiguous blocks of memory, enable efficient storage and retrieval of data elements, while strings, represented as arrays of characters, are fundamental in handling textual data. Through hands-on exercises and practical implementations, students delve into array manipulation techniques such as sorting, searching, and traversing, gaining proficiency in optimizing code performance and memory usage. Moreover, exploring string operations like concatenation, comparison, and substring extraction enhances students' ability to process and analyze textual information effectively. By immersing themselves in array and string-related challenges, students develop a robust understanding of data structures and algorithms, preparing them to tackle complex computational problems with confidence and precision in their future endeavors.

2.4.1 Array Operations:

Array operations form the backbone of many algorithms and data structures in computer science, and they are a central focus of the Problem Solving Using 'C' Lab. Through practical exercises and hands-on implementation, students learn to manipulate arrays efficiently, gaining essential skills for solving complex computational problems.

Basic Array Operations

In C programming, arrays are collections of elements of the same data type stored in contiguous memory locations. They offer a convenient way to store and access multiple values under a single variable name. Some fundamental array operations include initialization, traversal, insertion, deletion, searching, and sorting.

Example:

```
    #include <stdio.h>
int main() {
   // Initializing an array
   int arr[5] = {10, 20, 30, 40, 50};
```

```
    // Traversing the array and printing its elements
    printf("Array elements: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
        return 0;
}
```

Common Array Operations

Insertion: Adding elements to an array at a specific position or at the end.

Deletion: Removing elements from an array, either by shifting elements or marking them as invalid.

Searching: Finding the position or existence of a specific element in the array.

Sorting: Arranging the elements of the array in a specific order, such as ascending or descending.

Example of Array Searching:

```
#include <stdio.h>
// Function to search for an element in an array
int search(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return i; // Return index if key is found
        }
    }
    return -1; // Return -1 if key is not found
}
int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 30;
    int result = search(arr, n, key);
    if (result != -1) {
        printf("Element %d found at index %d\n", key, result);
    } else {
        printf("Element %d not found in the array\n", key);
```

```
    }
    return 0;
}
```

2.4.2 String Operations:

String operations are fundamental in computer programming, facilitating the manipulation and processing of textual data. In the Problem Solving Using 'C' Lab, students delve into practical exercises aimed at mastering various string operations using the C programming language. Through hands-on implementation and analysis, students acquire essential skills for solving real-world problems involving strings efficiently.

Basic String Operations:

In C programming, strings are represented as arrays of characters terminated by a null character ('\0'). Some common string operations include string initialization, concatenation, comparison, length determination, and substring extraction.

Example:

```
#include <stdio.h>
int main() {
    // Initializing a string
    char str[] = "Hello, World!";
    // Traversing and printing the characters of the string
    printf("String characters: ");
    for (int i = 0; str[i] != '\0'; i++) {
        printf("%c", str[i]);
    }
        return 0;
    }
```

Common String Operations:

Concatenation: Combining two or more strings to form a single string.

Comparison: Determining the equality or relative order of two strings. Length Determination: Finding the length of a string.

Substring Extraction: Extracting a portion of a string based on specified indices.

Example:

```
#include <stdio.h> #include <string.h>
int main() {
    char str[] = "Hello, World!";
    int length = strlen(str);
```

```
    printf("Length of the string: %d\n", length);
    return 0;
}
```

2.5 LAB EXERCISE 'C' PROGRAM SESSION WISE

Session 1

a) Write a C program to find sum and average of three numbers.

b) Write a C program to check whether a year is a leap year or not.

c) Write a program in c to find even or odd numbers.

d) Write a C program to find the sum of individual digits of a given positive integer.

e) Write a C program to generate the first n terms of the Fibonacci sequence.

Session 2

a) Write a C program to generate prime numbers between 1 to n.

b) Write a C program to Check whether given number is Armstrong Number or Not.

c) Write a program in c to calculate power using recursion.

d) Write a program in c to function to check lowercase letter.

e) Write program to display number 1 to 10 in octal, decimal and hexadecimal system.

Session 3

a) Write a C program to generate following pattern:

```
1
1  2
1  2  3
1  2  3  4
```

b) Write a C program to generate following pattern:

```
*
*  *
*  *  *
```

c) Write a C program to generate following pattern:

```
1
1  1
1  1  1
1  1  1  1
```

d) Write a program to multiplication of two matrix.

e) Write a program to demonstrate multiplication table input from user till 10.

2.6 PROJECT-BASED ASSIGNMENTS

Project-Based Assignments: Applying Algorithms and Data Structures to Real-World Problems:

1) Create a recommendation system for an e-commerce platform to suggest products to users based on their browsing and purchase history.

2) Develop a real-time traffic management system that dynamically adjusts traffic signal timings to optimize traffic flow.

3) Create a health monitoring system that analyzes and predicts potential health issues based on patient data.

4) Design an algorithm to find the shortest delivery route that covers multiple delivery points.

5) Develop a social media feed filter that prioritizes posts based on user preferences and recency.

***