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

BACHELOR OF COMPUTER APPLICATION (BCA)

CSB-1112 (Problem Solving Using C)

Block 4: Data Structures and Basic Algorithms in C

Unit 13: Introduction to Data Structures

13.0 Introduction

Data structures serve as the backbone of any computer program, providing a systematic way to organize and store data for efficient manipulation and retrieval. They lay the foundation for solving complex computational problems by offering a structured approach to represent and manage different types of data. In essence, data structures act as a bridge between the raw data and the algorithms that operate on them, enabling programmers to design efficient and scalable solutions.

At its core, the study of data structures revolves around understanding the strengths and weaknesses of various organizational schemes and selecting the most appropriate structure for a given problem. This involves analyzing factors such as access patterns, memory usage, and computational complexity to make informed decisions. Additionally, mastery of data structures empowers developers to optimize performance, minimize resource usage, and enhance the overall robustness of their software solutions.

Moreover, a solid grasp of data structures is essential for building a strong foundation in computer science and programming. It not only equips individuals with the tools to tackle diverse computational challenges but also fosters a deeper understanding of algorithmic principles. Whether it's implementing a simple linked list or designing intricate tree-based data structures, the knowledge of data structures forms the cornerstone of problem-solving in the world of software development. Thus, an introduction to data structures sets the stage for a journey into the realm of algorithmic thinking and computational problem-solving.

13.1 Objectives

After reading this chapter, you will be able to understand,

- Gain a foundational understanding of essential data structures such as arrays, linked lists, stacks, queues, trees, graphs, and hash tables.
- Develop problem-solving skills by learning how to select and implement appropriate data structures and algorithms to solve computational problems efficiently.
- Explore the analysis of algorithmic complexity, focusing on understanding time and space complexity and making informed decisions about algorithm selection.
- Gain practical experience through hands-on programming assignments, projects, and lab exercises, reinforcing understanding and application of data structures in real-world scenarios.
- Foster collaboration and communication skills through group projects, discussions, and peer feedback sessions, preparing students for teamwork and effective communication in the field of computer science.

13.2 Data Structure

A data structure is a particular method of arranging and storing information in a computer's memory for later, effective usage. Data can be organized in a variety of ways; for example, a data structure is a specific arrangement of data that follows a logical or mathematical paradigm.

A given data model's diversity is determined by two things:

- Its structure needs to be sufficiently loaded to represent the real relationships between the data and the real-world object; and
- Its formation needs to be straightforward enough for anyone to process the data effectively whenever it's needed.

Data structures are specialized formats for organizing, processing, and storing data. They enable efficient access and modification of data. Common data structures include arrays, linked lists, stacks, queues, trees, and graphs.

Importance of Data Structures

- **Efficient Data Management:** Helps in organizing data for efficient access and modification.
- **Optimized Performance:** Improves the performance of algorithms by providing optimal ways of data storage and retrieval.
- **Code Reusability:** Promotes the reuse of data manipulation routines.

Categories of Data Structure:

The data structure can be sub divided into major types:

- Linear Data Structure
- Non-linear Data Structure

Linear Data Structure:

A data structure is said to be linear if its elements combine to form any specific order. There are basically two techniques of representing such linear structure within memory.

- First way is to provide the linear relationships among all the elements represented by means of linear memory location. These linear structures are termed as arrays.

- The second technique is to provide the linear relationship among all the elements represented by using the concept of pointers or links. These linear structures are termed as linked lists. The common examples of linear data structure are:
  - ❖ Arrays
  - ❖ Queues
  - ❖ Stacks
  - ❖ Linked lists

Non linear Data Structure:

This structure is mostly used for representing data that contains a hierarchical relationship among various elements.

Examples of Non Linear Data Structures are listed below:

- Graphs
- Family of trees and
- Table of contents

Operations on the Data Structures:

Following operations can be performed on the data structures:

1. **Traversing:** It is used to access each data item exactly once so that it can be processed.
2. **Searching:** It is used to find out the location of the data item if it exists in the given collection of data items.
3. **Inserting:** It is used to add a new data item in the given collection of data items.
4. **Deleting:** It is used to delete an existing data item from the given collection of data items.
5. **Sorting:** It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.
6. **Merging:** It is used to combine the data items of two sorted files into single file in the sorted form.

13.3 Basic Data Structures in C

Array

An array is a collection of elements of the same type, stored in contiguous memory locations.

Example:

```
#include <stdio.h>
int main() {
   int arr[5] = {1, 2, 3, 4, 5};
   for (int i = 0; i < 5; i++) {
      printf("%d ", arr[i]);
   }
   return 0;
}
```

Advantages and disadvantages of Array

Advantages of Array

- Arrays allow constant time complexity (O(1)) access to elements using their index, making retrieval operations very fast.
- Arrays have minimal overhead and the contiguous memory layout improves cache performance, resulting in better overall performance.
- Arrays are straightforward to declare, initialize, and iterate over, making them easy to use and understand.
- The fixed size of arrays at compile time can simplify memory management and avoid issues related to dynamic memory allocation.
- Arrays are particularly useful for mathematical computations and algorithms that require direct access to elements, such as matrix operations and numerical simulations.

Disadvantages of Arrays

- The size of an array is fixed at compile time in static arrays, which can lead to wasted memory if the array is not fully utilized or insufficient memory if the array size is underestimated.
- Inserting or deleting elements in the middle of an array requires shifting elements, leading to O(n) time complexity for these operations.
- Arrays require a contiguous block of memory, which can be a limitation for very large arrays, leading to potential memory fragmentation issues.
- Arrays do not support dynamic resizing. If the size needs to change, a new array must be created and the data copied over, which is inefficient.
- If the array is sparsely populated, a significant amount of memory may be wasted. This is often a problem in scenarios where the maximum size of the array is much larger than the typical number of elements stored.

Linked Lists

A linked list is a linear data structure where each element (node) contains a data part and a pointer to the next element in the sequence.

Example:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
   int data;
   struct Node* next;
};
void printList(struct Node* n) {
   while (n != NULL) {
      printf("%d ", n->data);
      n = n->next;
   }
}
```

```
int main() {
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));
    head->data = 1; // Assign data to first node
    head->next = second; // Link first node with second
    second->data = 2; // Assign data to second node
    second->next = third; // Link second node with third
    third->data = 3; // Assign data to third node
    third->next = NULL; // Terminate the list
    printList(head);
    return 0;
}
```

Advantages and disadvantages of Linked List

Advantages of Linked List

- Linked lists can dynamically grow and shrink during runtime, allowing for efficient memory usage as elements can be added or removed without needing to predefine the size.
- Insertion and deletion operations in linked lists are generally efficient, especially for singly linked lists, as they only require adjusting pointers, resulting in O(1) time complexity for these operations.
- Linked lists can be easily adapted to implement various abstract data types and data structures, such as stacks, queues, and hash tables, by modifying the structure and operations of the nodes.

Disadvantages of Linked Lists

- Linked lists require additional memory for storing pointers to the next nodes, leading to higher memory overhead compared to arrays, especially for singly linked lists where each node only stores a single pointer.
- Linked lists do not support direct access to individual elements by index, requiring traversal from the beginning of the list to reach a specific node, resulting in O(n) time complexity for access operations.
- Linked lists use dynamic memory allocation for each node, which can lead to memory fragmentation over time, especially in applications with frequent node creation and deletion, potentially reducing overall system performance.

Stacks

A stack is a linear data structure that follows the Last In First Out (LIFO) principle. Elements can be added and removed only from the top of the stack.

Example:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
struct Stack {
    int arr[MAX];
    int top;
};
void initialize(struct Stack* s) {
    s->top = -1;
}
int isFull(struct Stack* s) {
    return s->top == MAX - 1;
}
int isEmpty(struct Stack* s) {
    return s->top == -1;
}
void push(struct Stack* s, int value) {
    if (isFull(s)) {
        printf("Stack Overflow\n");
        return;
    }
    s->arr[++s->top] = value;
}
int pop(struct Stack* s) {
    if (isEmpty(s)) {
        printf("Stack Underflow\n");
        return -1;
    }
    return s->arr[s->top--];
}
int main() {
    struct Stack s;
    initialize(&s);
```

```
    push(&s, 10);
    push(&s, 20);
    push(&s, 30);
    printf("%d popped from stack\n", pop(&s));
    printf("%d popped from stack\n", pop(&s));
    return 0;
}
```

Advantages and disadvantages of Stacks

Advantages of Stacks

- Stacks are straightforward to implement and understand, making them ideal for managing function calls, expression evaluation, and backtracking algorithms.
- The LIFO principle of stacks makes them suitable for managing resources or tasks that follow a "last in, first out" order, such as undo operations in text editors or browser history.
- Stacks typically have a small memory footprint, as they only store data and a limited amount of metadata (e.g., stack pointer), making them efficient in terms of memory usage.

Disadvantages of Stacks

- Stacks offer limited access to elements, with only the topmost element being accessible. This lack of random access can be restrictive for applications requiring access to elements other than the top.
- In languages that do not provide automatic memory management, managing dynamic memory allocation for stack operations (push and pop) can be error-prone and lead to memory leaks or segmentation faults if not handled properly.
- Stacks are suitable for specific applications with well-defined behaviors, such as function call management and expression evaluation. However, they may not be optimal for other scenarios that do not follow the LIFO principle.

Queues

A queue is a linear data structure that follows the First In First Out (FIFO) principle. Elements are added at the rear and removed from the front.

Example:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
struct Queue {
    int arr[MAX];
    int front, rear;
};
void initialize(struct Queue* q) {
    q->front = -1;
```

```
    q->rear = -1;
}
int isFull(struct Queue* q) {
    return q->rear == MAX - 1;
}
int isEmpty(struct Queue* q) {
    return q->front == -1 || q->front > q->rear;
}
void enqueue(struct Queue* q, int value) {
    if (isFull(q)) {
        printf("Queue Overflow\n");
        return;
    }
    if (isEmpty(q)) {
        q->front = 0;
    }
    q->arr[++q->rear] = value;
}
int dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue Underflow\n");
        return -1;
    }
    return q->arr[q->front++];
}
int main() {
    struct Queue q;
    initialize(&q);
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    printf("%d dequeued from queue\n", dequeue(&q));
    printf("%d dequeued from queue\n", dequeue(&q));
    return 0;
```

```
}
```

Advantages and disadvantages of Queues

Advantages of Queues

- Queues maintain the order of elements based on the principle of First In, First Out (FIFO). This makes them suitable for scenarios where the order of processing or execution is critical, such as task scheduling or message queuing systems.
- Queues are effective for managing shared resources or coordinating communication between concurrent processes or threads. They help in controlling access to resources in a synchronized manner, preventing race conditions and ensuring orderly execution.
- Queues act as buffers, allowing temporary storage of data when the producer and consumer processes operate at different speeds. They help in regulating the flow of data between components, preventing data loss or overflow.

Disadvantages of Queues

- Similar to stacks, queues offer limited access to elements, with only the front and rear elements being accessible. Direct access to elements other than the front or rear requires dequeuing or traversing the entire queue, leading to inefficiency for certain operations.
- As with stacks, managing dynamic memory allocation for queue operations (enqueue and dequeue) can be challenging, especially in languages that do not provide automatic memory management. Improper memory handling can result in memory leaks or fragmentation issues.
- Queues may incur additional space overhead due to the need to maintain pointers or references to queue nodes, especially in linked queue implementations. This overhead can increase memory usage, particularly for large queues.

13.4 Advanced Data Structures in C

Trees

A tree is a hierarchical data structure consisting of nodes, with a single node called the root and sub-nodes forming the branches.

Binary Tree

Example:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
   int data;
   struct Node* left;
   struct Node* right;
};
struct Node* newNode(int data) {
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
```

```
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
int main() {
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    printf("Inorder traversal: ");
    inorder(root);
    return 0;
}
```

Advantages and disadvantages of Tree

Advantages of Tree

- Trees provide a hierarchical organization of data, allowing for efficient representation and management of hierarchical relationships. This makes them suitable for applications such as file systems, organizational charts, and network routing tables.
- Binary search trees (BSTs) offer efficient search, insertion, and deletion operations with average-case time complexity of O(log n). This makes them suitable for implementing dictionary-like data structures and searching algorithms.
- Balanced trees, such as AVL trees and red-black trees, maintain a balanced structure, ensuring that the height of the tree remains relatively small compared to the number of elements. This results in improved performance for search, insertion, and deletion operations.

Disadvantages of Tree

- Trees can be complex to implement and maintain, especially balanced trees with intricate balancing algorithms. Managing tree structure, node rotations, and rebalancing operations can introduce overhead and complexity.
- Trees can consume more memory compared to other data structures, especially in scenarios where many small nodes are allocated. This can lead to increased memory usage and potential memory fragmentation, particularly in large tree structures.
- Certain tree operations, such as finding the predecessor or successor of a given node in a binary search tree, can be more complex and require additional logic compared to other data structures like arrays or linked lists.

Graphs

A graph is a collection of nodes (vertices) and edges connecting pairs of nodes. Graphs can be used to represent networks, such as social networks, communication networks, and transportation systems.

Graph Representation using Adjacency List:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
   int data;
   struct Node* next;
};
struct Graph {
   int numVertices;
   struct Node** adjLists;
};
struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;
   return newNode;
}
struct Graph* createGraph(int vertices) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->numVertices = vertices;
   graph->adjLists = (struct Node**)malloc(vertices * sizeof(struct Node*));
   for (int i = 0; i < vertices; i++) {
      graph->adjLists[i] = NULL;
   }
```

```
    return graph;
}
void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
void printGraph(struct Graph* graph) {
    for (int v = 0; v < graph->numVertices; v++) {
        struct Node* temp = graph->adjLists[v];
        printf("\n Adjacency list of vertex %d\n head ", v);
        while (temp) {
            printf("-> %d", temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}
int main() {
    int vertices = 5;
    struct Graph* graph = createGraph(vertices);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);
    printGraph(graph);
    return 0;
}
```

Advantages and disadvantages of Graph

Advantages of Graph

- Graphs can represent various types of relationships between entities, making them adaptable to different scenarios.
- They allow for both directed and undirected edges, weighted and unweighted edges, and cyclic and acyclic structures.
- Graph algorithms offer solutions to diverse problems like pathfinding, traversal, connectivity analysis, and optimization.

Disadvantages of Graphs

- Implementing and analyzing graph algorithms can be complex, especially for large or dense graphs.
- Graphs can consume significant memory, particularly for dense graphs or large datasets.
- Some graph operations, like finding shortest paths or detecting cycles, can have high computational complexity.

13.5 Summary

Data structures are essential for organizing, managing, and storing data efficiently in C programming. Understanding the basic and advanced data structures, such as arrays, linked lists, stacks, queues, trees, and graphs, is crucial for writing efficient and effective programs. Mastery of these concepts will enable you to solve complex problems and optimize the performance of your applications.

The importance of data structures in problem-solving cannot be overstated. They provide the foundation for designing efficient algorithms and solving complex problems effectively. Mastery of data structures enables developers to optimize performance, manage resources efficiently, and simplify the implementation of intricate algorithms, making them indispensable tools in the arsenal of any programmer or computer scientist.

13.6 Questions

1. **What is a data structure?**

Ans.1 A data structure is a way of organizing and storing data in a computer so that it can be accessed and manipulated efficiently. It defines the relationship between data elements and facilitates operations such as insertion, deletion, and traversal.

2. **What are the basic operations performed on data structures?**

Ans.2 The basic operations performed on data structures include:

- **Insertion:** Adding a new element to the structure.
- **Deletion:** Removing an element from the structure.
- **Traversal:** Visiting and accessing each element of the structure.
- **Search:** Finding the location of a specific element within the structure.
- **Update:** Modifying the value of an existing element.

3. **Write down the differences between arrays and linked lists.**

Ans.3 **Arrays:**

- Contiguous block of memory.
- Fixed size determined at compile time.

- Allows direct access to elements using index.
- Insertion and deletion operations can be inefficient due to shifting elements.

Linked Lists:

- Non-contiguous nodes connected by pointers.
- Dynamic size, can grow or shrink during runtime.
- Requires traversal to access elements.
- Efficient for insertion and deletion operations, especially in the middle of the list.

4. Explain the concept of algorithmic complexity.

Ans.4 Algorithmic complexity refers to the analysis of the performance of an algorithm in terms of its time and space requirements. Time complexity measures the amount of time taken by an algorithm to complete as a function of the size of the input. Space complexity measures the amount of memory space required by an algorithm to execute. Understanding algorithmic complexity helps in evaluating the efficiency of algorithms and making informed decisions about their suitability for different applications.

5. What are some common applications of trees in computer science?

Ans.5 Trees have various applications in computer science, including:

- Hierarchical data representation, such as file systems and organizational charts.
- Binary search trees for efficient searching and sorting operations.
- Expression parsing and evaluation in compilers and interpreters.
- Decision trees for decision-making in artificial intelligence and machine learning.
- Network routing algorithms for efficient data transmission.
- Representing hierarchical relationships in XML and JSON documents.

13.7 References

- Kernighan, B. W., & Ritchie, D. M. (1988). The C Programming Language (2nd ed.). Prentice Hall.
- King, K. N. (2008). C Programming: A Modern Approach (2nd ed.). W. W. Norton & Company.
- Prata, S. (2014). C Primer Plus (6th ed.). Addison-Wesley Professional.
- Deitel, P., & Deitel, H. (2016). C How to Program (8th ed.). Pearson.

***