Bachelor of Computer Applications (BCA)

Analysis of Algorithms and Data Structure

(OBCACO203T24)

Self-Learning Material (SEM II)

Jaipur National University
Centre for Distance and Online Education

Established by Government of Rajasthan
Approved by UGC under Sec 2(f) of UGC ACT 1956
&
NAAC A+ Accredited

TABLE OF CONTENTS

Course Introduction	i
Unit 1 Introduction to Data Structure	01 – 11
Unit 2 Linear List and Arrays	12 – 21
Unit 3 Stacks	22 – 28
Unit 4 Queue	29 – 40
Unit 5 Linked List	41 – 49
Unit 6 Linked List Operations	50 – 61
Unit 7 Tree	62 – 72
Unit 8 Binary Tree	73 – 84
Unit 9 Heaps	85 – 95

EXPERT COMMITTEE

Prof. Sunil Gupta
(Computer and Systems Sciences, JNU Jaipur)

Dr. Satish Pandey
(Computer and Systems Sciences, JNU Jaipur)

Dr. Shalini Rajawat
(Computer and Systems Sciences, JNU Jaipur)

COURSE COORDINATOR

Dr. Deepak Shekhawat
(Computer and Systems Sciences, JNU Jaipur)

UNIT PREPARATION

Unit Writer(s)

Mr. Ram Lal Yadav
(Computer and Systems Sciences, JNU Jaipur)
(Unit 1- 4)

Mrs. Rashmi Choudhary
(Computer and Systems Sciences, JNU Jaipur)
(Unit 5-9)

Assisting & Proofreading

Mr. Satender Singh
(Computer and Systems Sciences, JNU Jaipur)

Unit Editor

Mr. Shish Dubey
(Computer and Systems Sciences, JNU Jaipur)

Secretarial Assistance

Mr. Mukesh Sharma

COURSE INTRODUCTION

"Clean code always looks like it was written by someone who cares."
- Robert C. Martin

The "Analysis of Algorithms and Data Structures" course offers an in-depth exploration of the fundamental principles and techniques used to design, analyze, and optimize algorithms and data structures. This course is integral for students seeking to develop a deep understanding of computational efficiency and problem-solving strategies in computer science.

This course has 3 credits and is divided into 9 Units. Students will begin by investigating the theoretical underpinnings of algorithm analysis. They will study various methods for evaluating the performance of algorithms, including time and space complexity, and learn to use Big O notation to describe and compare the efficiency of different algorithms. This foundational knowledge is crucial for understanding how algorithms scale and perform under varying conditions.

The course delves into a wide range of fundamental data structures, exploring their characteristics, implementations, and applications. Students will examine structures such as arrays, linked lists, stacks, queues, trees, and graphs, understanding how each can be used to solve specific types of problems effectively. They will also explore advanced data structures, such as hash tables and balanced trees, which are essential for optimizing complex operations and ensuring efficient data retrieval and manipulation.

In addition to understanding individual data structures, students will learn how to apply and combine these structures to develop sophisticated algorithms. They will work with sorting and searching algorithms, exploring various techniques and their efficiencies, and examining how these methods can be adapted and optimized for different types of data and problem scenarios.

Critical thinking and analytical skills are central to the course, as students are encouraged to evaluate the trade-offs and performance implications of different approaches. They will develop a rigorous understanding of how to select and apply appropriate algorithms and data structures to achieve optimal performance for a given problem.

By the end of the course, students will have acquired a robust toolkit for analyzing and solving complex computational problems. They will be adept at designing efficient algorithms, implementing effective data structures, and applying these skills to a variety of practical and theoretical challenges in computer science.

Course Outcomes:

At the completion of the course, a student will be able to:

1. Understand basic data structures (such as an array based list, linked list, stack, queue, binary search tree) and algorithms.
2. Acquire the knowledge to analyze, design, apply and use data structures and algorithms to solve engineering problems
3. Evaluate the solutions of problems by implementing them using the advanced data structures.
4. Apply modern tools to solve engineering problems using C.
5. Describe an understanding of analysis of algorithms.
6. Synthesize an algorithm or program code or segment that contains iterative constructs and analyze the code segment

---

Acknowledgements:

The content we have utilized is solely educational in nature. The copyright proprietors of the materials reproduced in this book have been tracked down as much as possible. The editors apologize for any violation that may have happened, and they will be happy to rectify any such material in later versions of this book.

Unit: 1

Introduction to Data Structure

Learning Objectives:

1. Introduction to Data Structures
2. Understanding the complex operations
3. Understand the concept of Arrays- singular and multi-dimensional arrays.
4. Understanding the concept of Time and space trade-off

Structure:

1.1 Introduction to Data Structures
1.2 Data Structure Operations
1.3 Asymptotic Notation
1.4 Time/Space Trade-off
1.5 Summary
1.6 Keywords
1.7 Self-Assessment Questions
1.8 Case Study
1.9 References

1.1 Introduction to Data Structures

The data structure of a computer program is a method for organising and storing data in a way that makes it useful and easily accessible. Data structures give for efficient searching, arranging, adding, and removing of data alongside for the oversight of large amounts of data. The type and volume of data that will be recovered, the operations that must be performed on the data, and the performance requirements of the program all influence the choice of data structure for a given job. When data structures are utilized properly, a program can run faster and consume less memory.

1.1.1 Importance:

Effective data processing: Data structures provide a method for arranging and storing information so that it may be retrieved, altered, and stored in an effective manner.Data can be constantly accessible, for instance, if it is stored in a hash table.

Memory management: Effective data structure use can lower memory requirements and maximise resource utilisation. For instance, employing dynamic arrays rather than static arrays can make better use of memory.Code reuse is facilitated by the usage of data structures as building blocks in a variety of algorithms and programmes.

Abstraction: Data structures offer a level of abstraction that frees programmers from having to worry about the specifics of how data is saved and handled, allowing them to concentrate on the logical structure of the data and the operations that can be carried out on it.

Design of algorithms: To function effectively, many algorithms depend on particular data structures. Designing and putting into practise efficient algorithms requires a solid understanding of data structures.Primitive and non-primitive data structures are the two categories into which data structures can be divided.

1.2 Data Structure Operations

Every data structure supports a variety of operations that can be used to manipulate the data.

- **Traversing** a data structure is to go from element to element within it. It makes systematic trips to the data. Any DS can be used to accomplish this. The programme used to demonstrate array traversal is shown below.

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

// Driver Code
int main(){
    // Initialise array
    int arr[] = { 1, 2, 3, 4 };

    // size of array
    int N = sizeof(arr) / sizeof(arr[0]);

    // Traverse the element of arr[]
    for (int i = 0; i< N; i++) {

        // Print the element
cout<<arr[i] << ' ';
    }
    return 0;
}
```

- **Searching** is the process of locating a certain element within a specified data structure. When the necessary component is located, the endeavour is deemed successful. Arrays, linked lists, trees, graphs, etc. are the data structures which can all be searched on. The code for performing a search on an array is shown below:

```
 // C++ program to searching in an array
#include <iostream>
using namespace std;
// Function that finds element K in the
// array
void findElement(int arr[], int N, int K){
    // Traverse the element of arr[]
    // to find element K
    for (int i = 0; i< N; i++) {

        // If Element is present then
        // print the index and return
        if (arr[i] == K) {
cout<< "Element found!";
            return;
        }
    }
cout<< "Element Not found!";
}
 // Driver Code
int main(){
    // Initialise array
    int arr[] = { 1, 2, 3, 4 };

    // Element to be found
    int K = 3;

    // size of array
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function Call
findElement(arr, N, K);
    return 0;
}
```

- **Insertion**: the procedure that is applicable to all data structures. Insertion refers to the act of adding a new element to a data structure. When the required element is added to the required data-structure, the insertion operation is successful. In other instances, it fails because there is no more room to add any more elements to the data structure because it is already full. The insertion is known by the same term as an insertion into an array, linked list, graph, or tree of data. This action is referred to as Push in a stack. This action is referred to

as Enqueue in the queue. The code for the insertion operation on Arrays is provided below.

```
// C++ program for insertion in array
#include <iostream>
using namespace std;

// Function to print the array element
void printArray(int arr[], int N)
{
      // Traverse the element of arr[]
      for (int i = 0; i< N; i++) {

            // Print the element
            cout<<arr[i] << ' ';
      }
}

 // Driver Code
int main()
{
      // Initialise array
      int arr[4];

      // size of array
      int N = 4;

      // Insert elements in array
      for (int i = 1; i< 5; i++) {
            arr[i - 1] = i;
      }

      // Print array element
      printArray(arr, N);
      return 0;
}
```

- **Deletion**: It is the procedure we use on all data structures. A data structure's element can be deleted using deletion. When the necessary element is removed from the data structure, the deletion operation is successful. A deletion in a data structure such an array, linked list, graph, tree, etc. has the same name. This procedure is known as Pop in a stack. Dequeue is the name of this operation in Queue. The deletion code for the Array is provided below.

```
#include <bits/stdc++.h>
using namespace std;
// Function to print the element in stack
void printStack(stack<int> St)
{
    // Traverse the stack
    while (!St.empty()) {
        // Print top element
cout<<St.top() << ' ';

        // Pop top element
St.pop();
    }
}
// Driver Code
int main()
{
    // Initialise stack
    stack<int> St;

    // Insert Element in stack
St.push(4);
St.push(3);
St.push(2);
St.push(1);

    // Print elements before pop
    // operation on stack
printStack(St);

cout<<endl;

    // Pop the top element
St.pop();

    // Print elements after pop
    // operation on stack
printStack(St);
    return 0;
```

Some other methods

Create

By declaring them, it sets aside memory for programme elements. constructing data structures can be carried out both at compile and run time. The malloc() function is available.

Selection: -

It chooses a particular data from the current data. By including criteria in a loop, you can choose any piece of data.

Update

The data structure's contents are updated. You may also update any specified data by putting a condition in a loop, similar to the select method.

Sort

Arranging information into a specific (ascending or descending) order. To sort data quick, we can use different sorting methods. A bubble sort, for instance, sorts data in O(n2) time. There are numerous algorithms, including rapid sort, insertion sort, merge sort, and selection sort.

Merge

It is possible to merge data from two different orders in an ascending or descending order. To combine sorts of data, we utilise merge sort.

Split Data

Dividing data into many sub-components to speed up the process.

1.3 Asymptotic Notation

Asymptotic notation can be used to express the space or running time complexity of an algorithm based on the size of the input. It is often used in complexity analysis to describe the behavior of an algorithm as the amount of the input rises. The three most commonly used notations are Big O, Omega, and Theta.

- **Big O notation (O):** The running time or storage needs of an algorithm are given an upper bound by this notation. It symbolises the worst-case scenario or the most time or space an algorithm could use to try to solve a problem. When an algorithm's execution time, for instance, is O(n), it signifies that it scales linearly with the input size, n, or a smaller number.

Omega notation (Ω): This notation provides a lower constraint on the growth rate of the running time or space use of an algorithm. It represents the best-case scenario, which is the least amount of time or space that an algorithm might use to complete a

task. For example, if the running time of an algorithm is (n), it means that the running time of the method increases linearly to n or more input sizes.

- **Theta notation (Θ):**This notation provides an upper and lower bound on the rate of growth in the time or physical location needed to execute an algorithm. It indicates a typical scenario, i.e., the amount of time or space normally needed for an algorithm to complete a task. For instance, if an algorithm's execution time is (n), it means that it increases linearly as the size of the input n

1.4 Time/Space trade-off

When something improves while the other deteriorates, there has been a trade-off. This approach is used to solve problems in two ways: in a small space with a lot of time, or in a larger space with less time.

An algorithm is considered optimal when it helps solve a problem with little memory use and yields results rapidly. It isn't always feasible to satisfy both conditions at once, though.An algorithm based on a lookup table occurs most frequently. This suggests that the answers to some questions can be recorded for every potential value. This issue can be resolved by writing down the complete lookup table, but doing so will take up a lot of space and make it difficult to discover the solutions quickly.

Another method, which takes very little paper but could take a while, is to simply calculate the results without recording anything. So, the more time-efficient algorithms you have, the less space-efficient they would be.

1.4.1 Applications:

Compressed or Uncompressed data: A space-time trade-off can be used to tackle the problem of data storage. Uncompressed data loads more quickly but occupies more space. On the other hand, compressed data takes up more processing time to decompress but requires less storage space. In many circumstances, working directly with compressed data is feasible. When dealing with compressed bitmap indices, working with compression is faster than working without it.

- **Re-Rendering or Stored images:** In this situation, saving just the source and rendering it as an image would take up more memory but take less time;

similarly, caching an image is quicker than re-rendering but uses up more memory.

- ➢ **Smaller code or Loop Unrolling**: Smaller code takes up less memory, but it takes more time to compute because it must go back to the start of the loop after each iteration. Loop unrolling can increase binary size but improve execution performance. Although it takes up more memory space, it takes less time to process.

- ➢ **Lookup tables or Recalculation**: A lookup table's implementation option allows for full table inclusion, which shortens computation time but requires more memory. It can recalculate, or compute table entries, as necessary, lengthening computation times while shortening memory needs.

```
classMyClass {
    // Function to find Nth Fibonacci term
    static int Fibonacci(int N)   {
        // Base Case
        if (N < 2)
            return N;
        // Recursively computing the term
        // using recurrence relation
        return Fibonacci(N - 1) + Fibonacci(N - 2);
}    // Driver Code
    public static void main(String[] args)    {
        int N = 5;
        // Function Call
System.out.print(Fibonacci(N));
    }
}
```

Due to repeated calculations of the same sub problems, the above implementation's temporal complexity is exponential. There isn't much supplementary space. Our goal is to shorten the process's duration, despite the fact that it requires more room. The goal is to use Dynamic Programming to optimise the strategy by memorising the overlapping sub problems.

```
class MyClass {
// Function to find Nth Fibonacci term
static int Fibonacci(int N){
int[] f= new int[N + 2];
    int i;
    // 0th and 1st number of the
    // series are 0 and 1
f[0] = 0;
f[1] = 1;
    // Iterate over the range [2, N]
    for (i = 2; i<= N; i++) {
        // Add the previous 2 numbers
        // in the series and store it
        f[i] = f[i - 1] + f[i - 2];
    }
    // Return Nth Fibonacci Number
    return f[N];}

// Driver Code
public static void main (String[] args) {
    int N = 5;
    // Function Call
System.out.println(Fibonacci(N));
}
}
```

1.5 Summary

- A data structure is a method used in computer programs to efficiently organize and store data for easy access and usage.
- Data structures offer a method for efficiently retrieving, manipulating, and storing data by organizing and storing it in a certain way.
- Primitive and non-primitive data structures are the two categories into which data structures fall.
- To visit an element that is stored in a data structure is to traverse it. It examines facts in an orderly fashion.
- Searching is the process of locating a certain element inside the provided data structure. A successful outcome is achieved when the necessary component is located.

1.6 Keywords

- **Code Reusability**: Code reuse is facilitated by the fact that data structures may be utilized as building blocks in a variety of methods and applications.
- **Asymptotic Notation**: Depending on the amount of the input, asymptotic notation can be used to express the space complexity or running time of an

algorithm. It is frequently used to characterize how an algorithm functions as the size of the input increases in complexity analysis.

- **Time/space trade-off**: A trade-off is when one item goes up while another goes down. This method allows you to solve a problem in less time and with more space, or in less time and with very little space.
- **Lookup tables or Recalculation**: An implementation can use a lookup table to contain the complete table, which lowers computation time but raises memory requirements. When necessary, it can recalculate, or compute table entries, which lengthens computation times but uses less memory.

1.7 Self-Assessment Questions

1. Explain the Deletion operation?
2. Why is the importance of data structures?
3. Explain when the program was used?
4. Explain the concept of asymptotic notation and three commonly used notations?
5. Explain the concept of time/ space trade-off and its applications?

1.8 Case Study

Streamlining Operations at IT Company with Data Structures and Algorithms

A well-established IT company, had a complex problem to solve. Their existing product, a popular project management tool, was facing severe performance issues. With the increasing user base, their system was taking a considerable time to load data. Particularly, their feature which provided a report on project dependencies was underperforming.

Upon investigating, it was identified that the underlying issue was the data structure being used to store project dependency information. The existing system used simple arrays to store and traverse dependencies, resulting in inefficient data retrieval, thereby escalating the time complexity.

The team proposed the use of a Graph data structure. In computer science, a graph is an abstract data type used to model relationships. It was a perfect match for the

project management tool where projects and their dependencies could be represented as nodes and edges respectively.

Upon the implementation of the Graph data structure, the performance issue improved significantly. The time taken to load project dependencies decreased dramatically, leading to an improvement in overall system performance. The system could now handle more data with less processing time, hence, demonstrating the power of choosing the correct data structure.

Furthermore, the team used Dijkstra's algorithm for efficient traversal of the graph to provide the shortest path or sequence of projects based on their dependencies. This helped in enhancing the project management tool's functionality, enabling it to handle more complex project dependencies efficiently.

This real-life case shows how the correct usage of data structures and algorithms can solve real-world problems effectively.

Questions

- Why was the Graph data structure chosen over the simple arrays for this scenario?
- How did the implementation of Dijkstra's algorithm enhance the functionality of the project management tool?
- What other real-world scenarios can be effectively managed using different types of data structures and algorithms?

1.9 References:

1. Schaum Series, “Introduction to Data Structures”, TMH.
2. R.B. Patel, “Expert Data Structures with C”, Second Edition, Khanna Book publishing Co (P) Ltd.
3. Tenenbaum, “Data Structure using C++”, PHI.
4. Chattopadhyay S., Dastidar d G.and Chattopadhyay Matangini., “Data Structure through C language”, BPB publications.

`

Unit: 2

Linear List and Arrays

Learning Objectives:

1. Understand the concept of linear lists and their representation using arrays.
2. Perform operations on arrays.
3. Compare the efficiency of different search algorithms, specifically sequential search, and binary search.

Structure:

2.1 Linear Lists
2.2 Address Calculation in Single-Dimensional Arrays
2.3 Address Calculation in Multidimensional Arrays
2.4 Operations on Arrays
2.5 Sequential Search
2.6 Binary Search
2.7 Summary
2.8 Keywords
2.9 Self-Assessment Questions
2.10 Case Study
2.11 References

2.1 Linear Lists

Linear lists are important data structures used in programming, a good example being Arrays. A linear list talks about a sequence of elements stored in continuous memory locations.

2.2 Address Calculation in Single-Dimensional Arrays:

In a one-dimensional array, every element occupies a one-size memory block, and the elements are stored in memory in a continuous manner. The element's address is calculated with the base address of the array and the corresponding index of the element.

The formula for calculating the address of an element in a single-dimensional array is:

address = base_address + (element_size * index)

Here, the base_address is the memory point where the array begins, element_size is the size of each element in bytes, and index is the position of the element within the array (starting from 0).

2.3 Address Calculation in Multidimensional Arrays:

In multidimensional arrays, elements are arranged in a tabular form with rows and columns. The address of an element in a multidimensional array can be calculated using similar principles to a single-dimensional array.

For example, consider a two-dimensional array with rows and columns. The formula for calculating the address of an element is:

address = base_address + (element_size * (row * columns + column))

Here, base_address is the memory location where the array starts, element_size is the size of each element in bytes, row is the row index, and column is the column index. The multiplication (row * columns) calculates the offset caused by the row, and (row * columns + column) calculates the overall offset for the element in the array.

2.4 Operations on Arrays:

Arrays support various operations, including:

1. **Accessing Elements**: Accessing each element in an array can be done by using their indices. For example, thisarray[index] retrieves the element at the specified index. Following is an example of retrieving an element from the array.

```
public class WorkonArray {
public static int accessValue(int[] array, int index) {
if (index < 0 || index >= array.length) {
System.out.println("Invalid index");
return -1;
}
return array[index];
}
public static void main(String[] args) {
int[] thisArray = {10, 20, 30, 40, 50};
int index = 4; // Index of the element we want to access
int result = accessValue(thisArray, index);
if (result != -1) {
```

```
System.out.println("Element at index " + index + " is " + result);
}
}
}
```

In this code, the accessValue method checks the range of the parameter index. If the index is out of the range then it prints “Invalid Index”. The method is invoked in the main() and the arguments passes with the method hold the variable index.

2. **Insertion**: Insertion involves inserting elements to an array. When attempting to insert it can be done by shifting some elements to accommodate the new element. It can be done at the beginning, middle, or end of the array.
3. **Deletion**: This process removes an element from the array. Just like insertion, deletion may require shifting elements to fill the empty space.
4. **Updating**: Updating an element in an array involves modifying the value at a specific index.
5. **Searching**: Searching an array involves finding the index or value of a specific element within the array. Common search algorithms include linear search and binary search.
6. **Sorting**: Sorting is the process of arranging the elements of an array in a particular order, such as ascending or descending. Various sorting algorithms exist, each with its own characteristics and efficiency. Common sorting algorithms include bubble sort, insertion sort, merge sort, and quicksort. These algorithms differ in their approach to comparing and rearranging elements, as well as their time and space complexities.
7. **Merging**: Merging combines two or more arrays into a single sorted array. We can combine two arrays by creating a new array whose size is equal to the total of the two arrays' lengths. The items of the second array can then be copied into the new array after the elements of the first array.

These are some of the basic operations on arrays, and there are additional advanced operations and algorithms available depending on the specific programming language and requirements.

2.5 Sequential Search

The presence and location of a target element within a set of data can be found using a simple search process known as sequential search. It is referred to as "sequential"

because it systematically examines each data structure element until a match is discovered or the entire structure has been read across.

Following is the process that the Algorithm may make to perform the operation

1. Start from the beginning of the data structure.
2. Compare the target element with the current element being examined.
3. If the current element matches the target element, the search is successful, and the position or index of the element is returned.
4. If the current element does not match, move to the next element in the structure.
5. Repeat steps 2-4 until a match is found or the end of the structure is reached.
6. If the entire structure is traversed without finding a match, the search is considered unsuccessful.

Sequential search seems appropriate for small or unordered collections with unsorted data. Due to its time complexity of O(n), where n is the count of elements in the collection, it may not be effective for larger collections. There can be different search methods such as binary search (applicable exclusively on sorted collections) or hash-based searching may be more effective for bigger data sets.

Sequential search, also known as linear search, is a suitable method for small or unordered collections containing unsorted data. The algorithm follows a straightforward process:

1. It initiates from the beginning of the data structure.
2. The target element is compared with the current element being inspected.
3. If the current element matches the target, the search is successful, and the position or index of the element is returned.
4. If there is no match, the algorithm moves to the next element in the structure.
5. Steps 2 through 4 are repeated until a match is found or the end of the structure is reached.
6. If the entire structure is traversed without finding a match, the search is deemed unsuccessful.

However, due to its time complexity of O(n), where n represents the number of elements in the collection, sequential search may not be efficient for larger collections. In such cases, alternative search methods such as binary search (suitable for sorted collections) or hash-based searching could offer better performance.

```
public class SequentialSearch {
public static int sequentialSearch(int[] array, int target) {
for (int i = 0; i<array.length; i++) {
if (array[i] == target) {
return i;  // Return the index where the target is found
}
}
return -1;  // Return -1 if the target is not found
}
public static void main(String[] args) {
int[] array = {5, 2, 10, 8, 3};
int target = 10;
int index = sequentialSearch(array, target);
if (index != -1) {
System.out.println("Target found at index " + index);
} else {
System.out.println("Target not found");
}
}
}
```

In this code, the sequentialSearch method performs a sequential search on the given integer array to find the target value. It iterates through each element of the array and compares it with the target. If a match is found, the index is returned. If the entire array is traversed without finding a match, -1 is returned. The main() shows an example use of the sequential Search method. It initializes an array and a target value, calls the method, and prints the result accordingly.

2.6 Binary Search

Binary search is a search process used to determine the position or index of a target element within a sorted collection of data. It operates by repeatedly halving the search field until the desired element is located or determined to be absent.

Here's how the algorithm typically executes:

- It begins by selecting the middle element of the sorted collection.
- The middle element is compared with the target element.

- If a match is found, the search is successful, and the position or index is returned.
- If the middle element exceeds the target, the search process continues on the left half of the collection.
- Conversely, if the middle element is less than the target, the search proceeds on the right half of the collection.
- Steps 2 through 5 are repeated until a match is discovered or the search space is exhausted.

The sorted nature of the collection enhances the efficiency of binary search, as it allows the algorithm to eliminate half of the remaining search space at each step. Consequently, binary search significantly outperforms sequential search, particularly for large collections.

Binary search exhibits a time complexity of O(log n), where n represents the number of elements in the collection. This logarithmic complexity stems from the approach of dividing the search space in half with each iteration, employing a divide-and-conquer strategy. As a result, binary search is highly efficient, especially for datasets with a considerable number of elements.

```
public class BinarySearch {
public static int binarySearch(int[] array, int target) {
int left = 0;
int right = array.length - 1;
while (left <= right) {
int mid = left + (right - left) / 2;
if (array[mid] == target) {
return mid;  // Return the index where the target is found
}
if (array[mid] < target) {
left = mid + 1;  // Search in the right half
} else {
right = mid - 1;  // Search in the left half
}
}
return -1;  // Return -1 if the target is not found
```

```
}
public static void main(String[] args) {
int[] array = {2, 5, 8, 10, 15, 18, 20};
int target = 15;
int index = binarySearch(array, target);
if (index != -1) {
System.out.println("Target found at index " + index);
} else {
System.out.println("Target not found");
}
}
}
```

Following is a code to represent the binary search operation.

The binary search method in this code conducts a binary search on the provided sorted integer array to locate the target value. It keeps track of two pointers, "left" and "right," indicating the range of indices to search within. The method continually updates these pointers and compares the middle element with the target until either a match is identified or the search space becomes empty.

The main method demonstrates an example usage of the binarySearch method. It initializes a sorted array and a target value, calls the method, and prints the result accordingly.

2.7 Summary

a. Linear lists are important data structures used in programming, a good example being Arrays.

b. A linear list talks about a sequence of elements stored in continuous memory locations.

c. In a one-dimensional array, every element occupies a one-size memory block, and the elements are stored in memory in a continuous manner.

d. The formula for calculating the address of an element in a single-dimensional array is: address = base_address + (element_size * index)

e. In multidimensional arrays, elements are arranged in a tabular form with rows and columns. The address of an element in a multidimensional array can be calculated using similar principles to a single-dimensional array.

f. The formula for calculating the address of an element is: address = base_address + (element_size * (row * columns + column))
g. Accessing each element in an array can be done by using their indices.
h. Insertion involves inserting elements to an array.
i. Merging combines two or more arrays into a single sorted array. We can combine two arrays by creating a new array whose size is equal to the total of the two arrays' lengths.
j. The presence and location of a target element within a set of data can be found using a simple search process known as sequential search. It is referred to as "sequential" because it systematically examines each data structure element until a match is discovered or the entire structure has been read across.

2.8 Keywords

1. Linear List: A linear list is a data structure that represents a sequence of elements stored in continuous memory locations, where each element is accessed by its position or index.
2. One-Dimensional Array: A one-dimensional array is a linear list where elements are stored in continuous memory locations, and each element occupies a fixed-size memory block.
3. Address Calculation: The process of calculating the memory address of an element in an array using the base address, element size, and index or indices.
4. Multidimensional Array: A multidimensional array is a tabular representation of elements arranged in rows and columns, where each element is accessed using row and column indices.
5. **Sequential Search:** Sequential search, also known as linear search, is a straightforward algorithm employed to locate the existence and position of a target element within a dataset. It operates by systematically inspecting each element in a data structure until either a match is discovered or the entire structure has been traversed.

2.9 Self-Assessment Questions

1. Explain the concept of a linear list and its importance in programming. Provide examples of applications where linear lists are commonly used.

2. How is the address of an element in a single-dimensional array calculated? Discuss the formula for address calculation and provide an example.
3. Discuss the concept of multidimensional arrays and how elements are arranged in a tabular form. Explain the process of calculating the address of an element in a multidimensional array using the formula, and provide an example.
4. Describe the various operations that can be performed on arrays. Discuss insertion, merging, and accessing elements in an array. Provide examples for each operation.
5. Explain the sequential search algorithm and how it is used to find the presence and location of a target element in a linear list. Discuss the process of sequential search and its time complexity. Provide an example to illustrate the algorithm.
6. What is binary search and how does it differ from sequential search? Explain the algorithm of binary search and discuss its time complexity. Provide an example to demonstrate the binary search process.
7. Compare and contrast sequential search and binary search algorithms. Discuss their advantages, disadvantages, and specific use cases.
8. How does address calculation play a role in efficient memory access in arrays? Discuss the importance of address calculation and its impact on array performance.
9. Explain the concept of dynamic memory allocation in linear lists. Discuss the role of functions like malloc(), calloc(), and free() in allocating and deallocating memory for arrays.
10. Discuss the trade-offs between linear lists implemented using arrays versus linked lists. Compare their advantages, disadvantages, and typical use cases in different scenarios.

2.10 Case study:Online Bookstore Search Function

A popular online bookstore "Bookworm" carries over a million books in its inventory. The company is dedicated to helping customers find books efficiently, whether they're looking for a specific title or browsing genres. Initially, Bookworm implemented a Sequential Search algorithm for their search functionality. This algorithm searched through the list of books in the order they were stored. For instance, if a customer searched for a book located at the end of the list, the system had to go through almost every book before finding the correct one.

As the company grew and the inventory expanded, the Sequential Search approach proved inefficient, leading to slow search results and customer dissatisfaction. The technical team then decided to optimize the search function by implementing a Binary Search algorithm. The Binary Search works by continuously dividing the sorted list into two halves until it finds the desired book.

After the implementation, there was a significant improvement in search speed, even with the increased inventory. The average search time decreased substantially, enhancing customer experience and making browsing and shopping on Bookworm's website more enjoyable. The change also reduced server load, leading to additional cost benefits for the company.

However, this improvement came with a prerequisite: the list of books had to be sorted for the Binary Search to work. So, the company also had to implement efficient sorting algorithms and manage regular updates to the inventory to maintain the sorted order.

Questions:

- Why was the Sequential Search algorithm inefficient for the "Bookworm" online bookstore?
- How did implementing a Binary Search algorithm improve the search functionality of the bookstore's website?
- Considering the need for a sorted list for Binary Search, what challenges might the "Bookworm" face, and how could these be addressed while maintaining the improved search performance?

2.11 References

1. Schaum Series, “Introduction to Data Structures”, TMH.
2. R.B. Patel, “Expert Data Structures with C”, Second Edition, Khanna Book publishing Co (P) Ltd.
3. Tenenbaum, “Data Structure using C++”, PHI.
4. Chattopadhyay S., Dastidar d G.and Chattopadhyay Matangini., “Data Structure through C language”, BPB publications.

Unit: 3

Stacks

Learning Objectives:

1. Know the concept of Stacks
2. Learn to implement them
3. Understand their application

Structure:

3.1 Stacks
3.2 Application of Stacks
3.3 Summary
3.4 Keywords
3.5 Self-Assessment Questions
3.6 Case Study
3.7 References

3.1 Stacks

Stacks represent a linear type of data structure governed by the Last In First Out (LIFO) principle. They operate as an abstract data type where elements are added and removed exclusively from one end, typically referred to as the top of the stack.

The two fundamental operations associated with a stack are:

1. **Push**: With this action, an element is added to the top of the stack. Existing elements are moved one position forward and the new element rises to the top of the stack.
2. **Pop:** The element at the top of the stack is removed by this operation. The stack is refreshed after removing the highest element, which was added most recently.

   Additionally, there are a few other operations commonly associated with stacks:

1. **Peek/Top:** Without removing the element from the stack, this action returns the value of the top element. It enables you to look at the top element without changing the stack.

2. **Is Empty:** This process determines whether the stack is empty. It gives back a Boolean result that indicates whether or not the stack has any entries.

These primitive operations form the basic functionality of a stack and provide the essential means for manipulating and accessing the data stored within it.

Implementing stacks using arrays refers to using an array data structure to represent and manipulate a stack. In this implementation, the elements of the stack are stored in a fixed-size array.

The array is used to hold the elements of the stack, and an additional variable called top keeps track of the index of the topmost element in the stack. Initially, when the stack is empty, the top is set to -1.

Here is an example of implementing a stack in Java using an array:

```
public class Stack {
private int maxSize;
private int top;
private int[] stackArray;
public Stack(int size) {
maxSize = size;
stackArray = new int[maxSize];
top = -1;
}
public void push(int value) {
if (top == maxSize - 1) {
System.out.println("Stack is full. Cannot push element.");
 return;
}stackArray[++top] = value;
System.out.println("Pushed element: " + value);
} public int pop() {
if (top == -1) {
System.out.println("Stack is empty. Cannot pop element.");
return -1;
}
int poppedValue = stackArray[top--];
System.out.println("Popped element: " + poppedValue);
```

```
return poppedValue;
}
public int peek() {
if (top == -1) {
System.out.println("Stack is empty. Cannot peek element.");
return -1;
}
return stackArray[top];
}
public booleanisEmpty() {
return (top == -1);
}
public static void main(String[] args) {
Stack stack = new Stack(5);
stack.push(10);
stack.push(20);
stack.push(30);
stack.push(40);
System.out.println("Top element: " + stack.peek());
System.out.println("Is stack empty? " + stack.isEmpty());
stack.pop();
stack.pop();
stack.pop();
```

This example demonstrates the usage of a stack by creating a stack of integers. It initializes the stack with a maximum size of 5 and performs various operations like pushing elements onto the stack, popping elements from the stack, peeking at the top element, and checking if the stack is empty.

This array-based implementation provides a simple and straightforward way to implement stacks. However, it has a fixed capacity, meaning that the maximum number of elements the stack can hold is determined by the size of the array. If the stack exceeds this capacity, it may result in an overflow or require resizing the array.

3.2 Application of Stacks

Stacks are commonly used in arithmetic expression conversion and evaluation. Here are the applications of stacks in these contexts:

1. **Arithmetic Expression Conversion:** Stacks are used to convert arithmetic expressions from one form to another. The most common conversions are:

a. Infix to Postfix: Stacks serve as a valuable tool for converting arithmetic expressions from infix notation (where operators appear between operands) to postfix notation (where operators follow operands). This conversion process facilitates expression evaluation, aiding in the removal of ambiguity related to operator precedence and parentheses.

b. Infix to Prefix: Stacks can also be used to convert an arithmetic expression from infix notation to prefix notation. In this conversion, operators are placed before the operands.

c. Postfix/Prefix to Infix: Stacks can be utilized to convert an arithmetic expression from postfix or prefix notation back to infix notation.

These conversions are helpful in parsing and evaluating arithmetic expressions and are commonly used in compilers, calculators, and expression evaluators.

2. **Arithmetic Expression Evaluation:** Stacks are instrumental in evaluating arithmetic expressions. Once the expression is converted to postfix or prefix notation, a stack can be used to perform the evaluation. The steps involved are:
    - Create an empty stack.
    - Scan the expression from left to right.
    - If an operand is encountered, push it onto the stack.
    - If an operator is encountered, pop the required number of operands from the stack, perform the operation, and push the result back onto the stack.
    - Repeat steps c and d until the entire expression is scanned.
    - At the end, the stack will contain the final result of the expression.

This approach allows for efficient evaluation of arithmetic expressions while considering the precedence and associativity of operators.

Let us look in an example the conversion of an infix expression to postfix notation using a stack that assumes the infix expression contains only single-letter variables and is not handling unary operators or function calls.

In this example, the convert To Postfix() method accepts an infix expression as input and transforms it into postfix notation utilizing a stack. The method iterates through

each character of the infix expression and executes the required conversions according to the character's type (operand, operator, parentheses).

The get Precedence() method is responsible for assigning precedence values to operators such as +, -, *, /, and ^, determining their order of evaluation.

The main method exemplifies the conversion process by providing an infix expression "A + B * (C - D)" and displaying the resulting postfix expression.

3.3 Summary

- Stacks are a linear type of data structure based on the Last In First Out (LIFO) principle.
- This abstract data type follows LIFO, where elements are added and removed from one end, known as the top of the stack.
- Removing the topmost element is done through the pop operation.
- The IsEmpty function checks if the stack is empty, returning a Boolean result indicating its status.
- Elements of the stack are stored in an array, with an additional variable, 'top,' tracking the index of the topmost element.
- Stacks are employed in converting arithmetic expressions from one notation to another.
- They can convert arithmetic expressions from infix to prefix notation as well.
- Stacks play a crucial role in evaluating arithmetic expressions. Once converted to postfix or prefix notation, stacks facilitate the evaluation process.
- The getPrecedence() method assigns precedence values to operators like +, -, *, /, and ^, establishing their order of evaluation.

3.4 Keywords

Stack: A data structure that follows the Last-In-First-Out (LIFO) principle, where elements are added and removed from only one end, known as the top of the stack.

1. **LIFO:** Last-In-First-Out, a principle followed by stacks where the last element inserted into the stack is the first one to be removed.
2. **Pop:** An operation performed on a stack to remove the element at the top of the stack.
3. **IsEmpty:** A process that determines whether a stack is empty or not. It returns a Boolean result indicating whether the stack has any entries.

4. **Array:** A data structure used to hold the elements of a stack, where the additional variable called "top" keeps track of the index of the topmost element in the stack.

3.5 Self-Assessment Questions

1. Explain the concept of a stack and its basic operations.
2. How can a stack be implemented using an array? Provide the necessary algorithms for push and pop operations.
3. Describe the application of stacks in converting infix expressions to postfix expressions. Provide an example to illustrate the conversion process.
4. Discuss the role of stacks in evaluating arithmetic expressions. Explain how a stack can be used to perform the evaluation.
5. What are the applications of stacks in real-world scenarios? Provide examples where stacks are used to solve specific problems or optimize certain operations.

3.6 Case study:

Stack Implementation in a Web Browser's Back and Forward Features

In modern web browsers such as Google Chrome or Mozilla Firefox, one of the most utilized features is the ability to navigate "back" and "forward" through the user's browsing history. These features can be easily understood as real-world implementations of the Stack data structure.

The browser maintains two stacks - the Back Stack and the Forward Stack. When a user navigates to a new page, the URL of the current page is pushed into the Back Stack, and the user is taken to the new page. If the user decides to hit the "Back" button, the URL of the current page is pushed into the Forward Stack, and the user is taken to the previous page (i.e., the top element of the Back Stack is popped and displayed).

When a user chooses to navigate "Forward," the browsing process undergoes a reversal: the URL of the current page is pushed into the Back Stack, while the top element of the Forward Stack is popped and presented. However, if the user moves to a new page without utilizing the "Back" button, the Forward Stack is cleared. This action occurs because the user has embarked on a new browsing path, diverging from the previous one.

This implementation effectively uses the push and pop operations of the Stack data structure and allows for a seamless and intuitive browsing experience.

Questions:

- Why are two separate stacks (Back Stack and Forward Stack) used in this scenario instead of a single stack?
- What would happen if a user navigates to a new webpage after using the "Back" button and why does the Forward Stack need to be cleared?
- How could this system be further optimized to reduce memory usage while still maintaining the same user experience?

3.7 References

1. “Data Structures and Algorithm Analysis in Java" by Mark Allen Weiss.
2. "Data Structures: Abstraction and Design Using Java" by Elliot B. Koffman and Paul A. T. Wolfgang.
3. Schaum Series, “Introduction to Data Structures”, TMH

`

Unit: 4

Queue

Learning Objectives:

1. Understand the concept of Queues
2. Know its implementation
3. Learn about its applications

Structure:

4.1 Queues

4.2 Implementation of queues using Array

4.3 Applications of a Linear Queue

4.4 Applications of a circular queue

4.5 Applications of a Double ended Queue (Deque)

4.6 Summary

4.7 Keywords

4.8 Self-Assessment Questions

4.9 Case Study

4.10 References

4.1 Queues

First-In-First-Out (FIFO) is a linear data structure that is adhered to by queues. It is made to keep items in a particular order, with the addition of an element marking the beginning of its removal. Enqueue and dequeue are the two main operations of queues.

i. **Enqueue:** An element is added to the end of the queue via this procedure. An element moves to the front of the queue when it is enqueued.

ii. **Dequeue:** The element at the head of the queue is eliminated during this process. The element in front of the queue moves to the front when an element is dequeued.

Apart from these primary operations, queues usually also provide additional operations and properties:

Peek/Front: This operation returns the element at the front of the queue without removing it. It allows you to examine the next element that will be dequeued.

Is Empty: This operation checks whether the queue is empty. If there are no elements in the queue, it returns true; otherwise, it returns false.

Queues can be implemented using various data structures, such as arrays or linked lists. In some implementations, there might be additional operations like size or clear, which provide information about the number of elements in the queue or remove all elements from the queue, respectively.

Queues are commonly used in scenarios where the order of processing is important, such as handling tasks in a multi-threaded system, scheduling processes, or managing print jobs in a printer spooler.

4.2 Implementation of queues using Array

Queues can be implemented using an array, where the elements are stored in a contiguous block of memory. Here is a basic implementation of a queue using an array in C language

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_SIZE 100

typedef struct {
int queue[MAX_SIZE];
int front;
int rear;
int size;
} Queue;

void initQueue(Queue* q) {
q->front = 0;
q->rear = -1;
q->size = 0;
}

bool is_empty(Queue* q) {
return q->size == 0;
}
bool is_full(Queue* q) {
return q->size == MAX_SIZE;
}

void enqueue(Queue* q, int item) {
if (is_full(q)) {
printf("Queue is full. Unable to enqueue.\n");
return;
}
q->rear = (q->rear + 1) % MAX_SIZE;
q->queue[q->rear] = item;
q->size++;
}int dequeue(Queue* q) {
if (is_empty(q)) {
printf("Queue is empty. Unable to dequeue.\n");
```

```
int peek(Queue* q) {
if (is_empty(q)) {
printf("Queue is empty. Unable to peek.\n");
return -1;
}
return q->queue[q->front];
}
int get_size(Queue* q) {
return q->size;
}

int main() {
Queue q;
initQueue(&q);

enqueue(&q, 10);
enqueue(&q, 20);
enqueue(&q, 30);

printf("Front: %d\n", peek(&q));

while (!is_empty(&q)) {
printf("Dequeued: %d\n", dequeue(&q));
}

return 0;
```

In this C implementation, we define a struct Queue that contains an array to store integer variables for front, rear,the elements, and size of the queue.

- The init Queue() function initializes the queue by setting the front to 0, rear to -1, and size to 0.
- The is_empty() and is_full() functions check if the queue is empty or full, respectively.
- The enqueue() function adds an element to the rear of the queue. If the queue is full, it displays an error message.
- The dequeue() function removes and returns the front element of the queue. If the queue is empty, it displays an error message and returns -1.
- The peek() function returns the front element without removing it.

- The get size() function returns the current size of the queue.

- In the main() function, we demonstrate the usage of the queue by enqueuing three elements, peeking at the front element, and dequeuing all elements.

4.3 Applications of a Linear Queue

- Linear queues, also known as simple queues, have several applications in various domains. Here are some common applications of linear queues:

- **Printers:** In printer spooling systems, a linear queue is often used to manage print jobs. When multiple users send print requests simultaneously, the jobs are added to the queue and processed in the order they arrived. The first job in the queue is printed first, followed by the next job, and so on.

- **Operating Systems:** Linear queues are frequently employed in operating systems for process scheduling. In a multiprogramming environment, processes are added to a queue and executed in a sequential manner. The CPU is allocated to the process at the front of the queue, and once it completes, it is dequeued, allowing the next process in line to execute.

- **Call Centers:** Call centers often use linear queues to handle incoming calls. When callers contact the center, their calls are placed in a queue based on their arrival time. Agents handle the calls in a first-come-first-served manner, ensuring fairness in addressing customer inquiries or concerns.

- **Breadth-First Search (BFS)**: BFS is "a graph traversal algorithm that explores all the vertices of a graph in breadth-first order". It uses a queue to store the vertices being visited. Starting from a given vertex, the algorithm visits its neighbouring vertices, adds them to the queue, and continues the process until all reachable vertices have been visited.

- **Buffer Management**: Linear queues are utilized in buffer management systems, where data packets or messages are stored temporarily in a queue

before being processed or transmitted. For example, in network communication protocols, packets arriving at a node are placed in a queue for processing, ensuring proper handling and orderly transmission.

These are just a few examples of how linear queues find applications in various domains. The FIFO behavior of queues makes them suitable for scenarios that require ordering and sequential processing of elements.

4.4 Applications of a circular queue

Circular queues, also known as circular buffers or ring buffers, have several applications in different fields. Here are some common applications of circular queues:

- Memory Management: Circular queues are often used in memory management systems, such as embedded systems or operating systems. In these systems, circular queues can be used as fixed-size buffers to efficiently manage data transfers between different modules or components. For example, in a device driver, a circular queue can be used to hold incoming or outgoing data packets.

- Data Streaming: Circular queues are useful in applications that involve continuous data streaming or real-time processing. By using a circular queue, data can be buffered or processed in a circular manner, allowing a continuous flow of data. This is particularly valuable in applications such as audio and video processing, where a constant stream of data needs to be processed in real time.

- Producer-Consumer Problem: Circular queues are commonly used to solve the producer-consumer problem, which involves coordinating the interaction between multiple producer and consumer threads or processes. The circular queue acts as a buffer between the producers and consumers, allowing efficient and synchronized data transfer. Producers enqueue data into the circular queue, while consumers dequeue and process the data.

- Task Scheduling: Circular queues find applications in task scheduling algorithms, particularly in cyclic scheduling or round-robin scheduling. In these algorithms, tasks are scheduled in a circular order, where each task gets a fixed time slice for execution before moving to the next task in the queue. The circular queue helps maintain the order and rotation of tasks.

- Event Handling: Circular queues are useful for event handling systems, such as in graphical user interfaces (GUIs) or event-driven programming. Events, such as user inputs or system notifications, can be enqueued into a circular queue. The event loop then dequeues and processes these events in the order they were received, ensuring proper event handling and responsiveness.

- CPU Cache Management: Circular queues can be used in CPU cache management strategies. Cache replacement policies, such as the Least Recently Used (LRU) algorithm, use circular queues to keep track of the most recently accessed cache lines. The circular queue helps in efficiently managing the cache and deciding which cache lines to evict when the cache is full.

These are just a few examples of the applications of circular queues. Circular queues are preferred in scenarios where efficient data buffering, ordered processing, or continuous data streaming is required, making them valuable in a wide range of systems and applications.

4.5 Applications of a "Double ended Queue" (Deque)

A double-ended queue, also known as a "deque", is "a versatile data structure that supports insertion and deletion of elements from both ends". Due to its flexibility, a deque finds applications in various scenarios. Here are some common applications of a double-ended queue:

- **Palindrome Checking**: A deque can be used to check whether a given string or sequence is a palindrome. By inserting the characters from both ends of the string into a deque, you can compare the elements at the front and rear of the

deque iteratively. If they match for all corresponding positions, the string is a palindrome.

- **Sliding** Window Problems: Deques are helpful in solving sliding window problems efficiently. Sliding window problems involve processing a set of elements in a window that slides through a larger data structure. A deque allows easy insertion and deletion of elements from both ends, making it suitable for maintaining the elements within the sliding window as it moves.

- **Job Scheduling**: Deques can be used in job scheduling systems, where tasks or jobs need to be scheduled and executed. By allowing insertion and deletion of jobs from both ends, a deque can be used to implement priority-based or time-based job scheduling algorithms, where new jobs can be added dynamically and executed based on priority or scheduled time.

- **Cache Implementation**: Deques can be utilized in cache implementation for efficient memory management. The most recently used or frequently accessed items can be stored at the front of the deque, while the least used items are stored at the rear. This allows quick access to frequently used items and efficient eviction of less used items when the cache is full.

- **Undo/Redo Operations**: In applications that require undo and redo functionality, a deque can be employed to store the state of operations. Each time an operation is performed, the state is stored in the deque. The undo operation removes the most recent state from the front, while the redo operation adds a previously undone state to the front, enabling stepwise undo and redo functionality.

- **Implementing Data Structures**: Deques can serve as a building block for implementing other data structures. For example, double-ended queues are used in the implementation of double-ended priority queues, where elements have both a priority and a value. The flexibility of a deque allows efficient insertion, deletion, and access to elements based on their priority and value.

These are just a few examples of how a double-ended queue (deque) can be applied in various scenarios. The ability to efficiently insert and delete elements from both ends makes it a useful data structure in situations that require flexibility and dynamic operations.

4.6 Summary

A queue is “a linear data structure that follows the First-In-First-Out (FIFO) principle”. It is designed to hold elements in a specific order, where the element added first is the one that will be removed first.Queues have two primary operations: enqueue and dequeue.IsEmpty checks whether the queue is empty.Queues can be implemented using various data structures, such as arrays or linked lists.

Queues can be implemented using an array, where the elements are stored in a contiguous block of memory.

The enqueue() function adds an element to the rear of the queue. If the queue is full, it displays an error message.

The peek() function returns the front element without removing it.

Linear queues, also known as simple queues, have several applications in various domains.

In printer spooling systems, a linear queue is often used to manage print jobs. When multiple users send print requests simultaneously, the jobs are added to the queue and processed in the order they arrived.

Linear queues are frequently employed in operating systems for process scheduling.

Call centers often use linear queues to handle incoming calls. When callers contact the center, their calls are placed in a queue based on their arrival time.

Linear queues are utilized in buffer management systems, where data packets or messages are stored temporarily in a queue before being processed or transmitted.

Circular queues, also known as circular buffers or ring buffers, have several applications in different fields.

Circular queues are commonly used to solve the producer-consumer problem, which involves coordinating the interaction between multiple producer and consumer threads or processes.

Cache replacement policies, such as the Least Recently Used (LRU) algorithm, use circular queues to keep track of the most recently accessed cache lines.

A double-ended queue, also known as a deque, is a versatile data structure that supports insertion and deletion of elements from both ends.

Deques can be used in job scheduling systems, where tasks or jobs need to be scheduled and executed.

The ability to efficiently insert and delete elements from both ends makes it a useful data structure in situations that require flexibility and dynamic operations.

4.7 Keywords

Queue: A queue is a "linear data structure that follows the First-In-First-Out (FIFO) principle". It is designed to hold elements in a specific order, where the element added first is the one that will be removed first.

Enqueue: Enqueue is an operation in a queue that adds an element to the rear (end) of the queue. If the queue is full, it displays an error message.

Dequeue: Dequeue is an operation in a queue that removes the front (first) element from the queue. The removed element is no longer accessible in the queue.

Is Empty: Is Empty is a method or function used to check whether a queue is empty or not. It returns true if the queue contains no elements, and false otherwise.

Linear Queue: A linear queue, also known as a simple queue, is a "type of queue where elements are added at one end (rear) and removed from the other end (front) following the FIFO principle". It finds applications in various domains, such as printer spooling systems, process scheduling in operating systems, call centers, and buffer management systems."

4.8 Self-Assessment Questions

What is a queue, and what is the significance of the First-In-First-Out (FIFO) principle in its operation?

How can a queue be implemented using an array? Explain the steps involved in enqueue and dequeue operations.

Discuss some applications of a linear queue in real-world scenarios. Provide examples of domains where linear queues are commonly used.
What are circular queues, and what are their advantages over linear queues? Give examples of real-world applications where circular queues are used.
Explain the concept of a double-ended queue (deque) and its benefits compared to other data structures. Provide examples of situations where deques are useful in solving problems or improving efficiency.

4.9 Case study:

Netflix's Use of Queues in Content Delivery

Netflix, one of the world's largest video streaming platforms, faces the enormous task of providing seamless and consistent content delivery to millions of subscribers globally. To achieve this, the company employs data structures such as queues in its engineering and data architecture.

One specific application is the use of queues in handling customer requests and streaming content. When a user clicks on a video, the request is placed in a queue and processed in a 'First In, First Out' (FIFO) manner. This ensures a fair and systematic handling of requests.

A more nuanced application is observed in Netflix's content delivery network (CDN), named Open Connect. The CDN servers use queues to handle incoming requests for chunks of videos, ensuring optimal delivery by managing traffic during peak hours. A linear queue is used in this scenario, serving requests as they arrive and maintaining the quality of streaming service.

Moreover, the use of circular queues is prominent in handling video buffering. By implementing circular queues, Netflix ensures that once a video is fully buffered, the system doesn't have to clear and rebuffer if a user decides to replay or rewind the video, thus improving user experience.

Netflix also employs double-ended queues (deques) in its recommendation algorithm. These deques allow quick addition and removal of movie recommendations from both

ends (front - the most recommended, and rear - the least), enhancing the speed and efficiency of the recommendation engine.

Questions:

- How does the use of linear queues in Netflix's CDN servers contribute to optimal content delivery?

- Can you explain how circular queues contribute to the user experience in video streaming services like Netflix?

- What is the role of double-ended queues in Netflix's recommendation algorithm and how does it enhance the platform's efficiency?

4.10 References

1. "Data Structures and Algorithm Analysis in Java" by Mark Allen Weiss.
2. "Data Structures: Abstraction and Design Using Java" by Elliot B. Koffman and Paul A. T. Wolfgang.
3. Schaum Series, "Introduction to Data Structures", TMH
4. R.B. Patel, "Expert Data Structures with C", Second Edition, Khanna Book publishing Co (P) Ltd.

Unit: 5
Linked list

Learning Objectives:

1. Understand the concept and importance of data structures and algorithms in computing.
2. Familiarize with the definition, use, and types of linked lists as a data structure.
3. Learn about the components of a linked list: nodes and pointers.
4. Explore the memory representation of different types of linked lists (singly, doubly, and circular).
5. Grasp the concept of dynamic memory allocation in the context of linked lists.

Structure:

5.1 Linked Lists
5.2 Types of Linked Lists
5.3 Summary
5.4 Keywords
5.5 Self-Assessment Questions
5.6 Case Study
5.7 References

5.1 Linked List

A linked list is a "data structure consisting of a sequence of nodes, where each node contains a data element and a reference (or link) to the next node in the sequence". Unlike arrays, linked lists do not require contiguous memory allocation. Each node in a linked list can be scattered in different memory locations, and the links between nodes allow traversal through the list.

The basic components of a linked list are:

- Node: Each node in a linked list contains two fields: a data field to store the actual data element and a link field (or pointer) to the next node in the sequence.
- Head: The head of a linked list is a reference (or pointer) to the first node in the list.
- Tail: The tail of a linked list is a reference (or pointer) to the last node in the list.

To perform operations on a linked list, common operations include:

1. Insertion: Adding a new node to the list.
2. Insertion at the beginning: Create a new node, update its link to point to the current head, and update the head to point to the new node.
3. Insertion at the end: Create a new node, update the link of the last node to point to the new node, and update the tail to point to the new nod
4. Insertion at a specific position: Traverse the list to the desired position, create a new node, update the links of the adjacent nodes to include the new node.
5. Deletion: Removing a node from the list.
6. Deletion from the beginning: Update the head to point to the next node and free the memory of the removed node.
7. Deletion from the end: Traverse the list to find the second-to-last node, update its link to NULL, update the tail to point to the second-to-last node, and free the memory of the removed node.
8. Deletion from a specific position: Traverse the list to the desired position, update the links of the adjacent nodes to exclude the node to be deleted, and free the memory of the removed node.

- Searching: Finding a specific value in the list by traversing through the nodes and comparing the data elements.
- Traversing: Visiting each node in the list sequentially to perform some operation.

Linked lists are dynamic data structures, allowing efficient insertion and deletion operations, but they require extra memory for storing the links between nodes.

Additionally, random access to elements is not as efficient as in arrays because accessing a specific element in a linked list requires traversing the list from the beginning or the previous nodes.

5.2 Types of Linked List

Singly Linked List - It is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. The node containing a pointer to the next node means that the node stores the address of the next node in the sequence. A single linked list allows the traversal of data only in one way.

```
class MyLinkedList {
    // Structure of Node
    static class Node {
        int data;
        Node next;
    };
    // Function to print the content of linked list starting
from the given node
    static void printList(Node n)
    {
        // Iterate till n reaches null
        while (n != null) {
            // Print the data
            System.out.print(n.data + " ");
            n = n.next;
        }
    }
```

```
    "// Driver Code
    public static void main(String[] args)
    {
        Node head = null;
        Node second = null;
        Node third = null;

        // Allocate 3 nodes in
        // the heap
        head = new Node();
        second = new Node();
        third = new Node();

        // Assign data in first
        // node
head.data = 1;

        // Link first node with
        // second
head.next = second;

        // Assign data to second
        // node
second.data = 2;
second.next = third;
```

`

Circular Linked List

A “circular linked list is that in which the last node contains the pointer to the first node of the list. While traversing a circular linked list, we can begin at any node and traverse the list in any direction forward and backward until we reach the same node we started”. Thus, a circular linked list has no beginning and no end.

```
importjava.util.*;
class MyLinkedList {

    // Structure for a node
    static class Node {
        int data;
        Node next;
    };

    // Function to insert a node at the beginning of Circular
    // LL
    static Node push(Node head_ref, int data)
    {
        Node ptr1 = new Node();
        Node temp = head_ref;
        ptr1.data = data;
        ptr1.next = head_ref;

        // If linked list is not null then set the next
        // of last node
        if (head_ref != null) {
            while (temp.next != head_ref) {
                temp = temp.next;
            }
temp.next = ptr1;
        }
  // For the first node
        else
            ptr1.next = ptr1;

head_ref = ptr1;
        return head_ref;
    }
static void printList(Node head)
    {
        Node temp = head;
        if (head != null) {
            do {
                // Print the data
System.out.print(temp.data + " ");
                temp = temp.next;
            } while (temp != head);
        }
    }
```

```
 // Driver Code
    public static void main(String[] args)
    {
        // Initialize list as empty
        Node head = null;

        // Created linked list will
        // be 11.2.56.12
        head = push(head, 12);
        head = push(head, 56);
        head = push(head, 2);
        head = push(head, 11);

System.out.print("Contents of Circular"
                         + " Linked List\n ");

        // Function call
printList(head);
    }
}
```

Doubly Circular linked list

A Doubly Circular linked list or a circular two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in the sequence. The difference between the doubly linked and circular doubly list is the same as that between a singly linked list and a circular linked list. The circular doubly linked list does not contain null in the previous field of the first node.

```
 static class Node {
        int data;
        Node next;
        Node prev;
    };

    // Start with the empty list
    static Node start = null;

    // Function to insert Node at the beginning of the List
    static void insertBegin(int value)    {
            if (start == null) {
            Node new_node = new Node();
new_node.data = value;
new_node.next = new_node.prev = new_node;
            start = new_node;
            return;
```

```
 Node new_node = new Node();
            new_node.data = value;

        // Update the previous and next of new node
new_node.next = start;
new_node.prev = last;
      last.next = (start).prev = new_node;
start = new_node;
    }
static void display()
    {
        Node temp = start;
System.out.printf("\nTraversal in"+" forward direction \n”) ;
while (temp.next != start) {
System.out.printf("%d ", temp.data);
            temp = temp.next;
        }
System.out.printf("%d ", temp.data);

System.out.printf("\nTraversal in "
                          + "reverse direction \n");
        Node last = start.prev;
        temp = last;

        while (temp.prev != last) { System.out.printf("%d ",
temp.data);
            temp = temp.prev;
        }
System.out.printf("%d ", temp.data);
    }
public static void main(String[] args)  {
insertBegin(5);
insertBegin(4);
insertBegin(7);
System.out.printf("Created circular doubly"
                          + " linked list is: ");
display();
    }
}
```

5.3 Summary

❖ A linked list is a “data structure consisting of a sequence of nodes, where each node contains a data element and a reference (or link) to the next node in the sequence”.

- Each node in a linked list can be scattered in different memory locations, and the links between nodes allow traversal through the list.
- The head of a linked list is a reference (or pointer) to the first node in the list.
- Each node in a linked list contains two fields: “a data field to store the actual data element” and “a link field (or pointer) to the next node in the sequence”.
- Linked lists are dynamic data structures, allowing efficient insertion and deletion operations, but they require extra memory for storing the links between nodes.
- A single linked list allows the traversal of data only in one way.
- A circular linked “list is that in which the last node contains the pointer to the first node of the list”.
- A “Doubly Circular linked list” or a c”ircular two-way linked list” is a more complex type of linked list that contains a pointer to the next as well as the previous node in the sequence.

5.4 Keywords

1. **Linked List:** A data structure consisting of a sequence of nodes, where each node contains a data element and a reference to the next node in the sequence. Each node can be located in different memory locations, and the links between nodes allow traversal through the list .
2. **Head:** The reference or pointer to the first node in a linked list. It points to the beginning of the list and allows access to the entire list by following the links between nodes.
3. **Node:** A fundamental component of a linked list that stores a data element and a link to the next node. Each node contains a data field to hold the actual data and a link field (or pointer) to connect it with the next node in the sequence”.
4. **Circular Linked List:** A type of linked list in which the last node in the list contains a pointer that loops back to the first node, forming a circular structure . This allows traversal from any node to any other node in a continuous loop.
5. **Doubly Circular Linked List:** A more advanced type of linked list that contains a pointer to both the next and the previous node in the sequence. Each node in a doubly circular linked list has links in both directions, enabling traversal in both forward and backward directions. The last node's pointer connects with the first node, forming a circular structure .

5.5 Self-Assessment Questions

1. How does a linked list differ from other data structures, such as arrays, in terms of memory representation and flexibility?
2. What are the key components required to represent a linked list in memory, and how are they interconnected to maintain the structure of the list?
3. Explain the process of traversing a linked list and accessing individual nodes, highlighting any techniques or optimizations that can be employed.
4. How would you insert a new node into a linked list at a specific position, and what steps are necessary to update the pointers and maintain the integrity of the list?
5. Describe the process of deleting a node from a linked list, considering various scenarios such as deleting the head or tail node, and explain how the pointers are adjusted to preserve the structure of the list.

5.6 Case study:

Implementation of Linked Lists in Spotify's Recommendation Algorithm

Spotify, a leading music streaming service, handles millions of songs in its database and serves over 365 million active users worldwide. Their recommendation system plays a vital role in suggesting songs to users based on their listening history and preferences. Spotify uses data structures like linked lists and algorithms for managing and manipulating this vast array of data efficiently.

The streaming giant employs a data structure called "playlist-linked-list", a variation of doubly-linked lists, to manage playlists. Each node represents a song, containing data like song ID, artist, genre, and other metadata, along with two pointers to the previous and next song in the playlist. This structure enables seamless navigation, allowing users to shuffle songs, repeat a song, or play the previous or next song.

Moreover, when a new song is added to a user's playlist, it's dynamically inserted into the playlist-linked-list. Deletion operation also comes into play when a user removes a song. This shows how insertion and deletion operations of linked lists are employed in real-world applications.

This data structure is also pivotal for Spotify's "Discover Weekly" feature. The linked list structure enables the recommendation algorithm to sequence songs that would flow well together, providing a better user experience.

However, the scale of Spotify's operations does pose challenges. Large user bases can lead to longer traversal times, and managing memory usage becomes critical. But with a carefully optimized approach, Spotify successfully leverages the linked list data structure to its advantage.

Questions

- What is the "playlist-linked-list" data structure implemented by Spotify and how does it function?

- How do the operations of linked lists (insertion and deletion) come into play in the functionality of a user's playlist in Spotify?

- Given the scalability issues, how can the efficiency of operations on large linked lists, like those in Spotify, be improved?

5.7 References

1. "Data Structures and Algorithm Analysis in Java" by Mark Allen Weiss.
2. "Data Structures: Abstraction and Design Using Java" by Elliot B. Koffmanand Paul A. T. Wolfgang.
3. Schaum Series, “Introduction to Data Structures”, TMH
4. R.B. Patel, “Expert Data Structures with C”, Second Edition, Khanna Book publishing Co (P) Ltd.

Unit: 6
Linked List Operations

Learning Objectives:

1. Understand the fundamentals of traversing a linked list
2. Learn to insert new nodes into linked lists
3. Apply different deletion strategies to remove nodes from a linked list, considering time complexities
4. Recognize the advantages and use cases of header linked lists and two-way linked lists

Structure:

6.1 Linked List Operations
6.2 Header Linked List
6.3 Two-way Linked List
6.4 Summary
6.5 Keywords
6.6 Self-Assessment Questions
6.7 Case Study
6.8 References

6.1 Linked List Operations

6.1.1 Traversing a Linked List

To traverse a linked list, you need to start from the head node and visit each node in the list sequentially until you reach the end (i.e., the node with a null reference in its link field)

Set a temporary node pointer to the head of the linked list.

While the temporary pointer is not null, perform the following steps:

a. Process the data of the current node.

b. Move the temporary pointer to the next node by updating it to the value in thecurrent node's link field.

Repeat steps 2a and 2b until the temporary pointer becomes null, indicating the end of the list.

The following code will help in understanding the traversal

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

// Definition of a Node
struct Node {
    int data;
    struct Node* next;
};

// Function to traverse the linked list
void traverseLinkedList(struct Node* head) {
    struct Node* current = head;

    while (current != NULL) {
        // Process the data of the current node (print, perform an
operation, etc.)
printf("%d ", current->data);

        // Move to the next node
        current = current->next;
    }
```

```
int main() {
    // Create a linked list: 1 -> 2 -> 3 -> 4 -> NULL

    struct Node* head = (struct Node*)malloc(sizeof(struct Node));
    head->data = 1;
    head->next = (struct Node*)malloc(sizeof(struct Node));
    head->next->data = 2;
    head->next->next = (struct Node*)malloc(sizeof(struct Node));
    head->next->next->data = 3;
    head->next->next->next = (struct Node*)malloc(sizeof(struct
Node));
    head->next->next->next->data = 4;
    head->next->next->next->next = NULL;

    // Traverse the linked list and print the data of each node
traverseLinkedList(head);

    // Free the allocated memory
    struct Node* current = head;
    struct Node* next;
    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }

    return 0;
}
```

This code defines a linked list structure (struct Node) and implements the traverse Linked List function to traverse the list and print the data of each node. In the main function, a linked list with four nodes is created, and then the traverseLinkedList function is called to perform the traversal and print the data. Finally, the dynamically allocated memory for the linked list is freed to prevent memory leaks.

6.1.2 Insert into a Linked List

To insert a new node into a linked list, you need to consider three cases: insertion at the beginning, insertion at the end, and insertion at a specific position within the list. Here's an example of insertion at the beginning:

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

// Definition of a Node
struct Node { int data;  struct Node* next; };

// Function to insert a new node at the beginning of the linked
list
void insertAtBeginning(struct Node** headRef, int data) {
    // Create a new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
newNode->data = data;
    // Set the next pointer of the new node to the current head
newNode->next = *headRef;
    // Update the head pointer to the new node
    *headRef = newNode;
}”

“// Function to print the linked list
void printLinkedList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
printf("%d ", current->data);
        current = current->next;
    }
printf("\n");
}

int main() {
    struct Node* head = NULL;
    // Insert nodes into the linked list at the beginning
insertAtBeginning(&head, 3);
insertAtBeginning(&head, 2);
insertAtBeginning(&head, 1);
printLinkedList(head);  // Output: 1 2 3

    return 0;
}”
```

This code focuses on the insert At Beginning function and demonstrates how to insert nodes at the beginning of a linked list. The print Linked List function is provided to print the elements of the linked list for verification. In the main function, three nodes with values 1, 2, and 3 are inserted at the beginning of the linked list, and then the linked list is printed to confirm the insertion.

6.1.3 Deleting from a Linked List

To delete a node from a linked list, you need to consider three cases: deletion of the head node, deletion of a node in the middle of the list, and deletion of the last node. Here is an example implementation in C that covers the case of deleting a node from the middle.

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

// Definition of a Node
struct Node {
    int data;
    struct Node* next;
};

// Function to delete a node with a specific value from the linked
list
void deleteNode(struct Node** headRef, int value) {
    struct Node* current = *headRef;
    struct Node* prev = NULL;

    // Traverse the list to find the node to delete
    while (current != NULL && current->data != value) {
prev = current;
        current = current->next;
    }"
"// If the node is not found, return
    if (current == NULL) {
        return;
    }

    // If the node to delete is the head node
    if (current == *headRef) {
        *headRef = (*headRef)->next;
    } else {
prev->next = current->next;
    }
    free(current);
}"

"// Function to print the linked list
void printLinkedList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
printf("%d ", current->data);
        current = current->next;
    }
printf("\n");
}"
```

```
"int main() {
    struct Node* head = NULL;

    // Insert nodes into the linked list
    head = (struct Node*)malloc(sizeof(struct Node));
    head->data = 1;
    head->next = (struct Node*)malloc(sizeof(struct Node));
    head->next->data = 2;
    head->next->next = (struct Node*)malloc(sizeof(struct Node));
    head->next->next->data = 3;
    head->next->next->next = NULL;

printLinkedList(head);  // Output: 1 2 3

    // Delete nodes from the linked list
deleteNode(&head, 2);
printLinkedList(head);  // Output: 1 3

    return 0;
}"
.
```

In this code, the focus is on the deleteNode function, which deletes a node with a specific value from the linked list. The printLinkedList function is provided to print the elements of the linked list for verification. In the main function, three nodes with values 1, 2, and 3 are inserted into the linked list. Then, the node with value 2 is deleted from the middle of the linked list, and the updated linked list is printed to confirm the deletion.

6.2 Header Linked List

If you are referring to a header node in a linked list, it means adding a special node at the beginning of the list that serves as a placeholder or header, rather than containing actual data. The header node helps in simplifying certain operations and provides a convenient starting point for traversing and manipulating the list.

Here's an example of how you can implement a linked list with a header node in C:

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

// Definition of a Node
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new header node
struct Node* createHeaderNode() {
    struct Node* header = (struct Node*)malloc(sizeof(struct Node));
    header->data = -1; // Placeholder value for the header node
    header->next = NULL;
    return header;
}

// Function to insert a new node at the beginning of the linked list
void insertAtBeginning(struct Node* header, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
newNode->data = data;
newNode->next = header->next;
    header->next = newNode;
}
// Function to print the linked list
void printLinkedList(struct Node* header) {
    struct Node* current = header->next;
    while (current != NULL) {
printf("%d ", current->data);
        current = current->next;
    }
printf("\n");
}"

"int main() {
    struct Node* header = createHeaderNode();

    // Insert nodes into the linked list
insertAtBeginning(header, 3);
insertAtBeginning(header, 2);
insertAtBeginning(header, 1);

printLinkedList(header);  // Output: 1 2 3

    return 0;
}"
```

In this code, a header node is created using the createHeaderNode function. The header node acts as the starting point for the linked list and does not contain actual data. The insert At Beginning function is modified to take the header node as a parameter and inserts new nodes after the header node. The printLinkedList function

is also modified to start printing the list from the node after the header. In the main function, nodes with values 1, 2, and 3 are inserted at the beginning of the linked list, and the elements of the list (excluding the header) are printed.

6.3 Two-way Linked List

A doubly linked list, also known as a two-way linked list, is a type of linked list where each node contains two pointers: one pointing to the previous node and one pointing to the next node. This allows for traversal in both directions, making it easier to perform operations such as insertion, deletion, and searching in both forward and backward directions. Here's an example of how you can implement a doubly linked list in C

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

// Definition of a Node
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
newNode->data = data;
newNode->prev = NULL;
newNode->next = NULL;
    return newNode;
}
// Function to insert a new node at the beginning of the doubly
linked list
void insertAtBeginning(struct Node** headRef, int data) {
    struct Node* newNode = createNode(data);
newNode->next = *headRef;

    if (*headRef != NULL) {
        (*headRef)->prev = newNode;
    }

    *headRef = newNode;
}"
```

```
"// Function to print the doubly linked list in forward direction
void printForward(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
printf("%d ", current->data);
        current = current->next;
    }
printf("\n");
}

// Function to print the doubly linked list in backward direction
void printBackward(struct Node* tail) {
    struct Node* current = tail;
    while (current != NULL) {
printf("%d ", current->data);
        current = current->prev;
    }
printf("\n");
}

int main() {
    struct Node* head = NULL;

    // Insert nodes into the doubly linked list at the beginning
insertAtBeginning(&head, 3);
insertAtBeginning(&head, 2);
insertAtBeginning(&head, 1);

printForward(head);   // Output: 1 2 3
printBackward(head);  // Output: 3 2 1

    return 0;
}
```

In this code, the Node structure is modified to include a prev pointer in addition to the next pointer. The create Node function creates a new node with the specified data and initialises the prev and next pointers to NULL. The insert At Beginning function inserts a new node at the beginning of the doubly linked list, ensuring that the prev and next pointers are properly updated. The print Forward function prints the elements of the doubly linked list in the forward direction by following the next pointers, and the print Backward function prints the elements in the backward direction by following the prev pointers.

In the main function, nodes with values 1, 2, and 3 are inserted at the beginning of the doubly linked list, and the elements of the list are printed both in forward and backward directions to demonstrate the traversal capabilities of the doubly linked list.

6.4 Summary

To traverse a linked list, you need to start from the head node and visit each node in the list sequentially until you reach the end.

In the main function, a linked list with four nodes is created, and then the traverse Linked List function is called to perform the traversal and print the data.

The print Linked List function is provided to print the elements of the linked list for verification. In the main function, three nodes with values 1, 2, and 3 are inserted at the beginning of the linked list, and then the linked list is printed to confirm the insertion.

To delete a node from a linked list, you need to consider three cases: deletion of the head node, deletion of a node in the middle of the list, and deletion of the last node.

The header node helps in simplifying certain operations and provides a convenient starting point for traversing and manipulating the list.

A doubly linked list, also known as a two-way linked list, is a type of linked list where each node contains two pointers: one pointing to the previous node and one pointing to the next node.

The print Forward function prints the elements of the doubly linked list in the forward direction by following the next pointers, and the print Backward function prints the elements in the backward direction by following the prev pointers.

6.5 Keywords

Traversal: The process of visiting each node in a linked list sequentially, starting from the head node and progressing through the links until reaching the end. It allows access to each node's data and performs operations on the elements as needed.

Creation: The process of constructing a linked list by allocating memory for nodes and setting up the appropriate data values and links between nodes. It involves initialising the head node and linking subsequent nodes to create the desired sequence.

Insertion: The process of adding a new node to a linked list at a specific position. It can involve inserting a node at the beginning (head) of the list, at the end, or in the middle, depending on the desired location and the list's current structure.

Deletion: The process of removing a node from a linked list. There are three common cases to consider: deleting the head node, deleting a node in the middle of the list, and deleting the last node. It involves updating the links between nodes to maintain the integrity and connectivity of the remaining list.

Doubly Linked List: a particular kind of linked list in which every node has two pointers: one that points to the node before it and another that points to the node after it. This makes it possible to traverse both forward and backward. It provides additional flexibility compared to a singly linked list but requires more memory to store the extra pointers.

6.6 Self-Assessment Questions

1. How does traversal of a linked list work and what is its time complexity?
2. Explain the process of inserting a node into a linked list and how it affects the rest of the list.
3. What are the common techniques for deleting a node from a linked list, and how do they differ in terms of time complexity?
4. What is a header linked list, and what advantages does it offer over a regular linked list implementation?
5. What is a two-way linked list, and what are its main characteristics and use cases compared to a regular linked list?

6.7 Case study:

Implementation of a Double Linked List in a Music Streaming App

In the booming era of digital music, streaming applications have revolutionized the way we listen to our favourite tunes. Our subject, Let'sPlay, a startup music streaming app, faced a critical challenge in optimizing its playlist management system.

The developers initially implemented the playlists as arrays, which seemed to be the simplest data structure. However, they soon ran into problems. Insertion and deletion operations became expensive in terms of time complexity when songs were added or removed in the middle of playlists, negatively impacting the app's performance.

Moreover, navigating between songs (back and forth) wasn't smooth.
The development team decided to switch to using a two-way or double linked list to overcome these challenges. Each song was stored as a node in the linked list, with 'next' and 'previous' pointers leading to the subsequent and preceding songs. This provided a smooth and efficient user experience when navigating between songs.

The decision to implement a two-way linked list brought significant performance improvements. Adding or removing songs in the middle of a playlist, which was previously a time-consuming operation, became much more efficient. The app's overall performance improved noticeably, and users found navigating their playlists much smoother, enhancing the overall user experience. This successful implementation led to a surge in app ratings and user retention, crucial metrics for any startup.

Questions:

1. What were the limitations of using arrays for playlist management in the Let'sPlay app?
2. How did the implementation of a two-way linked list address the issues faced by the Let'sPlay app?
3. Discuss the improvements in app performance and user experience brought by the two-way linked list implementation.

6.8 References

1. "Data Structures and Algorithm Analysis in Java" by Mark Allen Weiss.
2. "Data Structures: Abstraction and Design Using Java" by Elliot B. Koffman and Paul A. T. Wolfgang.
3. Schaum Series, “Introduction to Data Structures”, TMH
4. R.B. Patel, “Expert Data Structures with C”, Second Edition, Khanna Book publishing Co (P) Ltd.

Unit: 7

Tree

Learning Objectives:

1. Understand the concept of a tree as a hierarchical data structure
2. The definition and properties of a binary tree
3. Key terms related to binary trees, such as root, parent, child, leaf, subtree, etc.
4. Different tree traversal techniques

Structure:

7.1 Tree Data Structure
7.2 Implementation of Tree
7.3 Binary Tree
7.4 Summary
7.5 Keywords
7.6 Self-Assessment Questions
7.7 Case Study
7.8 References

7.1 Tree Data Structure

A tree is a common hierarchical data structure in algorithms and data structures. It is a group of nodes with edges connecting them, and each node may have one or more child nodes. A tree's root node is at the top, and each child node has the potential to generate children of its own, creating subtrees.

Trees are an essential data structure in computer science and have various applications, such as representing hierarchical relationships, organising data for efficient searching and sorting, and implementing various algorithms like binary search and balanced search trees.

Function is called to perform the traversal and print the data. Finally, the dynamically allocated memory for the linked list is freed to prevent memory leaks.

7.1.1 Key Terminology in Trees:

- **Root:** The topmost node of the tree.
- **Parent**: A node that has child nodes.
- **Child**: A node connected to its parent node.
- **Siblings**: Nodes that have the same parent.
- **Leaf**: A node with no children.
- **Edge**: A connection between nodes.
- **Depth:** The distance along the path that leads from the root to a certain node.
- **Height**: the deepest point that a node in the tree can go.
- **Subtree:** A tree rooted at a child node.

7.1.2 Types of Trees:

- **Binary Tree:** A tree in which each node can have at most two children.

- **Binary Search Tree (BST):** a binary tree where a node's right child has a value larger than the node and its left child has a value less than the node.

- **AVL Tree:** a self-balancing binary search tree where a node's left and right subtrees have height differences of no more than one.

- **Red-Black Tree:** A self-balancing binary search tree that ensures insert, remove, and search operations have logarithmic temporal complexity.

- **B-Tree:** A self-balancing search tree designed to optimise disk read/write operations by minimising the number of disk accesses required.

- **Trie (Prefix Tree):** a data structure that resembles a tree and is used to hold dynamic sets or associative arrays with strings for keys.

7.1.3 Tree Traversal:

- Depth-First Traversal: Visit the nodes in a tree depth-wise before visiting siblings. Three common types of depth-first traversal are pre-order, in-order, and post-order traversal.

- Pre-order Traversal: Visit the current node, then traverse the left subtree, and finally traverse the right subtree.

- In-order Traversal: Traverse the left subtree, visit the current node, and then traverse the right subtree. For binary search trees, this traversal gives nodes in sorted order.

- Post-order Traversal: Traverse the left subtree, traverse the right subtree, and visit the current node.

- Breadth-First Traversal: Visit the nodes level by level, from left to right.

- Trees are fundamental for solving complex problems efficiently, and understanding tree data structures and algorithms is crucial for many applications in computer science and software development.

`

7.2 Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers.

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

// Definition of a Node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = data;
newNode->left = NULL;
newNode->right = NULL;
    return newNode;
}

// Function to insert a new node into the binary tree
void insertNode(struct Node** rootRef, int data) {
    if (*rootRef == NULL) {
        *rootRef = createNode(data);
    } else {
        if (data <= (*rootRef)->data) {
insertNode(&(*rootRef)->left, data);
        } else {
insertNode(&(*rootRef)->right, data);
        }
    }
}

// Function to perform an in-order traversal of the binary tree
void inOrderTraversal(struct Node* root) {
    if (root != NULL) {
inOrderTraversal(root->left);
printf("%d ", root->data);
inOrderTraversal(root->right);
    }
}int main() {
    struct Node* root = NULL;

    // Insert nodes into the binary tree
insertNode(&root, 4);
insertNode(&root, 2);
insertNode(&root, 1);
insertNode(&root, 3);
insertNode(&root, 6);
insertNode(&root, 5);
insertNode(&root, 7);

    // Perform in-order traversal
printf("In-order traversal: ");
inOrderTraversal(root);  // Output: 1 2 3 4 5 6 7
printf("\n");

    return 0;
}
```

In this code, the Node structure is defined to represent a node in the binary tree. The Node function is used to create a new node with the specified data value. The insertNode function inserts a new node into the binary tree based on the value of the node. If the value is less than or equal to the current node's value, it is inserted into the left subtree; otherwise, it is inserted into the right subtree. The inOrderTraversal function performs an in-order traversal of the binary tree, printing the data of each node in ascending order.

In the main function, nodes with values 4, 2, 1, 3, 6, 5, and 7 are inserted into the binary tree using the insertNode function. Then, the in-order traversal is performed, resulting in the nodes being printed in ascending order (1, 2, 3, 4, 5, 6, 7).

7.3 Binary Tree

The left child and the right child are the two children that each node can have, at most, in a binary tree, a type of tree data structure. The root node of the binary tree may have left or right subtrees, or it may be empty (null). The left subtree contains nodes whose values are fewer than the root node, whereas the right subtree contains nodes whose values are greater than the root node. This property makes binary trees useful for efficient sorting and searching.

7.3.1 Binary Tree Operations:

- Insertion: To add a new node to the binary tree, it is typically inserted as a leaf node according to certain rules (e.g., smaller values to the left, larger values to the right).

- Traversal: There are different ways to traverse a binary tree to visit all its nodes. Common traversal methods include pre-order, in-order, post-order, and level-order traversal.

- Searching: Binary trees provide an efficient way to search for a specific value. It can be performed by comparing the target value with the values of nodes and traversing left or right based on the comparison result.

- Deletion: Removing a node from a binary tree involves rearranging the tree structure while maintaining the binary tree properties.

- Balancing: Balancing a binary tree ensures that the heights of the left and right subtrees are approximately equal, improving the efficiency of search and other operations. Examples of balanced binary trees include AVL trees and Red-Black trees.

Here's an example of how you can delete a node from a binary tree in C

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

// Definition of a Node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
newNode->data = data;
newNode->left = NULL;
newNode->right = NULL;
    return newNode;
}

// Function to insert a new node into the binary tree
void insertNode(struct Node** rootRef, int data) {
    if (*rootRef == NULL) {
        *rootRef = createNode(data);
    } else {
        if (data <= (*rootRef)->data) {
insertNode(&(*rootRef)->left, data);
        } else {
insertNode(&(*rootRef)->right, data);
        }
    }
}"
```

```
“// Function to find the minimum value node in a binary tree
struct Node* findMinNode(struct Node* root) {
    while (root->left != NULL) {
        root = root->left;
    }
    return root;
}

// Function to delete a node from the binary tree
struct Node* deleteNode(struct Node* root, int data) {
    if (root == NULL) {
        return root;
    } else if (data < root->data) {
        root->left = deleteNode(root->left, data);
    } else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    } else {
        // Case 1: No child or only one child
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        // Case 2: Two children
        struct Node* temp = findMinNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// Function to perform an in-order traversal of the binary tree
void inOrderTraversal(struct Node* root) {
    if (root != NULL) {
inOrderTraversal(root->left);
printf("%d ", root->data);
inOrderTraversal(root->right);
    }
}”
```

```
"int main() {
    struct Node* root = NULL;

    // Insert nodes into the binary tree
insertNode(&root, 4);
insertNode(&root, 2);
insertNode(&root, 1);
insertNode(&root, 3);
insertNode(&root, 6);
insertNode(&root, 5);
insertNode(&root, 7);


    // Perform in-order traversal
printf("In-order traversal before deletion: ");
inOrderTraversal(root);
printf("\n");

    // Delete a node (e.g., delete node with value 4)
    root = deleteNode(root, 4);
    // Perform in-order traversal after deletion
printf("In-order traversal after deletion: ");
inOrderTraversal(root);
printf("\n");

    return 0;
}"
```

In this code, the deleteNode function is added to delete a node from the binary tree based on the given data value. There are three cases to consider when deleting a node:

Case 1: No child or only one child: In this case, the node is simply removed from the tree by rearranging the child pointers.

Case 2: Two children: If the node to be deleted has two children, we find the minimum value node in its right subtree (which will be the leftmost node in the right subtree). We replace the node's data with the data of the minimum value node, and then recursively delete the minimum value node from the right subtree.

In the main function, nodes with values 4, 2, 1, 3, 6, 5, and 7 are inserted into the binary tree using the insertNode function. Then, the in-order traversal is performed

before and after deleting a node (in this case, node with value 4). The in-order traversal demonstrates that the node has been successfully deleted from the binary tree.

7.4 Summary

- A tree is a common hierarchical data structure in algorithms and data structures. It is a group of nodes with edges connecting them, and each node may have one or more child nodes. A tree's root node is at the top, and each child node has the potential to generate children of its own, creating subtrees.
- Trees are an essential data structure in computer science and have various applications, such as representing hierarchical relationships, organising data for efficient searching and sorting, and implementing various algorithms like binary search and balanced search trees.
- Child is a node connected to its parent node.
- A Binary Tree is a tree where a node can have a maximum of two offspring.
- The AVL Tree is a self-balancing binary search tree where a node's left and right subtrees have height differences of no more than one.
- Trees are fundamental for solving complex problems efficiently, and understanding tree data structures and algorithms is crucial for many applications in computer science and software development.
- The Node function is used to create a new node with the specified data value.
- In the main function, nodes with values 4, 2, 1, 3, 6, 5, and 7 are inserted into the binary tree using the insert Node function.
- A binary tree is a kind of tree data structure where the left child and the right child are the two children that each node can have, at most. The binary tree can have a root node with left and/or right subtrees, or it can be empty (null).

7.5 Keywords

Tree: a node-based, edge-connected hierarchical data structure in which a node may have zero or more children. It is widely used in data structures and algorithms to represent hierarchical relationships and organise data efficiently.

Root Node: The topmost node in a tree, serving as the starting point for traversing or accessing the entire tree structure. It is the only node in the tree that does not have a parent.

Binary Tree: A type of tree data structure in which each node can have at most two children: a left child and a right child. It is a fundamental tree variant used in various algorithms and data structures.

AVL Tree:a self-balancing binary search tree where a node's left and right subtrees have height differences of no more than one.

It maintains its balance during insertions and deletions, ensuring efficient search, insertion, and deletion operations.

Node: A fundamental component of a tree that contains a data value and connects to other nodes through edges. Each node can have child nodes, forming subtrees. In the context of binary trees, a node typically has a left child and a right child, along with the data value it holds. The Node function is used to create new nodes with specific data values.

7.6 Self-Assessment Questions

1. What is the definition of a tree in the context of data structures?
2. How would you define a binary tree? What are its distinguishing characteristics?
3. What are some common terms used in binary trees and their definitions? (e.g., root, parent, leaf, subtree)
4. Can you explain the difference between depth and height in a binary tree?
5. What is the significance of the terms "pre-order," "in-order," and "post-order" when discussing tree traversal in binary trees?

7.7 Case study:

Implementation of Binary Search Trees in Database Indexing

In a leading e-commerce company, managing large volumes of data efficiently is a crucial requirement. One major aspect of their database management system is its indexing mechanism, which significantly speeds up data retrieval operations.

The e-commerce company uses binary search trees (BSTs) as an underlying data structure for database indexing. Each node in the BST represents a database record, with the node key being a unique identifier (say, a product ID). Records with smaller keys are found in the left child node while records with bigger keys are found in the right child node of a BST node. This arrangement allows the database management

system to bypass a large portion of the records during search operations, enabling faster retrieval of data.

For instance, when a customer queries for a specific product, the system doesn't need to scan through the entire database. Instead, it traverses through the BST. If the queried product ID is less than the one at the current node, it moves to the left child; if greater, it moves to the right child. This process continues until the system finds the queried product or confirms the product does not exist.

However, as the business scaled up, the database records increased exponentially. Consequently, the BST became unbalanced, leading to inefficient search operations as the tree height increased. To solve this issue, the company implemented a self-balancing BST (like an AVL or Red-Black Tree), ensuring that the tree remained balanced after every insertion or deletion operation, thus maintaining search operation efficiency.

Questions:

1. How does the use of binary search trees improve the efficiency of data retrieval operations in the database system of the e-commerce company?
2. What problem did the company face with the growth of the database, and how does an unbalanced tree affect the efficiency of a BST?
3. Discuss the solution implemented by the company to maintain the efficiency of search operations as the number of records grew. What are the properties of the data structure they chose?

7.8 References

1. "Data Structures and Algorithm Analysis in Java" by Mark Allen Weiss.
2. "Data Structures: Abstraction and Design Using Java" by Elliot B. Koffman and Paul A. T. Wolfgang.
3. Schaum Series, “Introduction to Data Structures”, TMH
4. R.B. Patel, “Expert Data Structures with C”, Second Edition, Khanna Book publishing Co (P) Ltd.

Unit: 8
Binary tree

Learning Objectives:

1. Understand about the types of trees
2. Application and advantages of binary tree
3. Understand the tree traversal techniques

Structure:

8.1 Binary Tree

8.2 Tree Traversal

8.3 Summary

8.4 Keywords

8.5 Self-Assessment Questions

8.6 Case Study

8.7 References

8.1 Binary Tree

Any tree with "two offspring or less for any of its nodes is said to be binary". Binary trees come in a variety of types.

8.1.1 Types of Binary Tree

Full Binary Tree

If every node has 0 or 2 children, a binary tree is fully formed. Here are a few illustrations of complete binary trees. A full binary tree is a binary tree in which all nodes have two offspring, apart from leaf nodes.

Degenerate Binary Tree

A tree with one child at each internal node. Such trees perform similarly to linked lists in terms of speed. A tree that only has one child, whether it be left or right, is considered degenerated.

Skewed Binary Tree

"A degenerate tree known as a skewed binary tree is one in which the right or left nodes predominate". Thus, "left-skewed binary trees" and "right-skewed binary trees" are the two different types of skewed binary trees.

8.1.2 Application of Binary Tree

- **Search Algorithms**:
  Binary search algorithms make optimal use of the binary tree's structure to look for a particular member. The complexity of the search can be expressed as "O(log n)", where "n is the number of nodes in the tree".
- **Sorting Algorithms**:
  Binary trees can be utilised to build effective sorting algorithms, including heap sort and binary search tree sort.
- **Database systems**:
  Data can be stored in binary trees, where each node corresponds to a record. As a result, search operations may be carried out more effectively, and the database system can handle vast amounts of data.
- **File Systems:**
  File systems can be implemented using binary trees, where each node is a directory or file. This makes file system searching and navigation efficient.
- **Compression Algorithms:**
  The Huffman coding compression algorithm, which assigns variable-length codes to letters based on their frequency of occurrence in the input data, can be implemented using binary trees.
- **Decision Trees:**
  Decision trees are a form of machine learning method used for classification and regression analysis. Binary trees can be utilised to create decision trees.
- **Gaming AI:**

Game AI can be implemented using binary trees, where each node represents a potential move in the game. To discover the optimum move, the AI programme can search the tree.

8.1.3 Advantages of Binary Tree

- **Efficient Searching**

  When looking for a specific element, binary trees perform very well since each node can only have two child nodes, which makes binary search techniques possible. This indicates that searches can be carried out in "O(log n)" time complexity.

- **Ordered Traversal**

  Binary trees "can be traversed in a certain order, such as in-order, pre-order, and post-order, thanks to their structural design". This makes it possible to do operations on the nodes in a certain order, like printing the nodes in sorted order.

- **Memory Efficient**

  Binary trees only need "two child references per node, which makes them comparatively memory-efficient when compared to other tree designs". This translates into the ability to keep a lot of data in memory and still perform effective searches.

- **Faster Insertion and Deletion**

  Insertions and deletions using binary trees can be done in O(log n) time complexity. They are thus an excellent option for applications like database systems that demand dynamic data structures.

- **Easy to implement**

  Binary are simple to create and comprehend.

- **Useful for Sorting**

  Binary search tree sort and heap sort are two effective sorting algorithms that can be implemented using binary trees.

8.2 Tree Traversal

8.2.1 Introduction

We want to process a binary tree by "visiting" each of its nodes and then taking some action on each occasion, such publishing the node's contents. A traversal is any procedure that involves going to each node in some order. An “enumeration of the tree's nodes is any traversal that lists each node in the tree exactly once”. Some applications only demand that “each node be visited exactly once, regardless of the sequence in which they are visited”. Nodes must be visited in a manner that preserves some relationship for other applications.

Preorder Traversal

A traversal in a binary tree that visits the root, the left child, the right child, and then the left child again. The algorithm for preorder traversal is shown as follows:

“Preorder(root):

- Follow step 2 to 4 until root != NULL
- Write root -> data
- Preorder (root -> left)
- Preorder (root -> right)

End loop”

Below is the code implementation of the preorder traversal.

Each node in the binary tree is represented by a Node struct that is defined by this code. With the provided data, a new node is created using the new Node function. The preorder traversal is carried out recursively via the preorder Traversal function. It then recursively calls itself for the left and right subtrees after printing the data of the current node. The Output will be like this: Preorder traversal of binary tree: 1 2 4 5 3

```
“#include <stdio.h>
#include <stdlib.h>
// Structure for a binary tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};”
“// Function to create a new node
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;
}”
“// Preorder traversal function
void preorderTraversal(struct Node* root) {
    if (root == NULL)
        return;
printf("%d ", root->data);  // Print the current node's data
preorderTraversal(root->left);  // Recursively traverse the left
subtree
preorderTraversal(root->right);  // Recursively traverse the right
subtree
}
int main() {
    // Constructing a sample binary tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

printf("Preorder traversal of binary tree: ");
preorderTraversal(root);

    return 0;
}”
```

Postorder Traversal

It's possible that we'd want to visit each node after seeing its offspring (and their subtrees). This would be necessary, for instance, if we wanted to put all of the tree's nodes back in free storage. Prior to deleting a node, we would like to remove all of its children. However, in order to achieve that, the children's children must first be erased, and so on.

Algorithm for Postorder Traversal of Binary Tree:
The algorithm for postorder traversal is shown as follows:

"Postorder(root):

- Follow step 2 to 4 until root != NULL
- Postorder (root -> left)
- Postorder (root -> right)
- Write root -> data

End loop"
Let us look at the below code. We first define a structure Node to represent a node in a binary tree. The createNode function is used to create a new node with the given data.

`

```
“#include <stdio.h>
#include <stdlib.h>
// Structure for a binary tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};”
“// Function to create a new node
struct Node* createNode(int data)
{
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    if (newNode == NULL) {
printf("Memory allocation failed!");
exit(1);
    }
newNode->data = data;
newNode->left = newNode->right = NULL;
    return newNode;
}”

“// Function to perform postorder traversal of a binary tree
void postorderTraversal(struct Node* root){
    if (root == NULL)
        return;

    // Visit left subtree
postorderTraversal(root->left);

    // Visit right subtree
postorderTraversal(root->right);

    // Visit current node
printf("%d ", root->data);
}”
`
```

```
 int main(){
    // Create the binary tree
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

printf("Postorder traversal of the binary tree: ");
postorderTraversal(root);

    return 0;
}
```

The postorder Traversal function performs the postorder traversal recursively by first visiting the left subtree, then the right subtree, and finally the current node. In the main function, we create a binary tree and call the postorder Traversal function to print the nodes in postorder traversal.Output the postorder traversal of the binary tree will be: Postorder traversal of the binary tree: 4 5 2 3 1

Inorder Traversal

In an inorder traversal, the left child and its entire subtree are visited first, followed by the node and then “right child” and its “entire subtree”. The binary search tree utilizes this traversal method to display all nodes in ascending order based on their values. Algorithm of the inorder traversal in a tree can be like this:

“Traverse the left subtree, i.e., call Inorder(left->subtree)
Visit the root.
Traverse the right subtree, i.e., call Inorder(right->subtree)”

Below is the code to show how the algorithm is applied. the TreeNode structure represents a node in the binary tree. The createNode function is used to create a new node with a given value.

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

"// Definition for a binary tree node
struct TreeNode {
    int val;
    struct TreeNode* left;
    struct TreeNode* right;
}"
```

```
 "// Function to create a new node
struct TreeNode* createNode(int value) {
    struct TreeNode* newNode = (struct
TreeNode*)malloc(sizeof(struct TreeNode));
    if (newNode == NULL) {
printf("Memory allocation failed!\n");
exit(1);
    }
newNode->val = value;
newNode->left = NULL;
newNode->right = NULL;
    return newNode;
}"

"// Function to perform inorder traversal of the binary tree
void inorderTraversal(struct TreeNode* root) {
    if (root == NULL)
        return;

inorderTraversal(root->left);    // Recursively traverse left
subtree
printf("%d ", root->val);        // Visit the current node
(printing value here)
inorderTraversal(root->right);   // Recursively traverse right
subtree
}"

int main() {
    // Creating a sample binary tree
    struct TreeNode* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

printf("Inorder traversal of the binary tree: ");
inorderTraversal(root);
printf("\n");

    return 0;
}
```

The function for inorder traversal conducts the process recursively. It starts by traversing the left subtree, then visits the current node, and finally traverses the right subtree. In the main function, a sample binary tree is constructed, and the inorder traversal is executed.

8.3 Summary

- ❖ Binary trees are a fundamental data structure used in computer science and are applied in various fields. They consist of nodes, each containing a value and two pointers to its left and right children.
- ❖ The application of binary trees extends to areas such as data storage, searching, and sorting algorithms. Their hierarchical structure allows for efficient operations like insertion, deletion, and retrieval.
- ❖ Tree traversals are algorithms used to visit and process each node in a binary tree. The three main types of traversals are pre-order, in-order, and post-order, each defining a different order of visiting the nodes.
- ❖ Pre-order traversal visits the root node first, followed by its left subtree and then its right subtree. It is commonly used for copying a tree, evaluating expressions, and constructing prefix notation .
- ❖ In-order traversal visits the left subtree first, then the root node, and finally the right subtree. It is often used for sorting binary search trees and retrieving the elements in a sorted order .
- ❖ Post-order traversal visits the left subtree first, then the right subtree, and finally the root node. It is useful for deleting a tree, evaluating postfix expressions, and calculating the height of a binary tree .

8.4 Keywords

- **Binary Tree:** A hierarchical data structure consisting of nodes, each having at most two children (left and right), used for efficient storage and retrieval of data.
- **Tree Traversals:** Algorithms for visiting and processing the nodes of a tree in a specific order.
- **Pre-order Traversal:** A tree traversal algorithm that visits the root node first, followed by its left and right subtrees.
- **In-order Traversal:** A tree traversal algorithm that visits the left subtree, then the root node, and finally the right subtree.
- **Post-order Traversal:** A tree traversal algorithm that visits the left and right subtrees first, and then the root node.

- **Binary Search Tree**: A type of binary tree in which the nodes are organized in a specific order (e.g., values in the left subtree are less than the root, and values in the right subtree are greater), enabling efficient searching and sorting operations.

8.5 Self-Assessment Questions

What is the main difference between a binary tree and a binary search tree?

How does a pre-order traversal algorithm work in binary trees? Provide a step-by-step explanation.

In what scenarios would you choose an in-order traversal over a pre-order or post-order traversal?

Can you explain the concept of a post-order traversal in binary trees? What are some practical applications of this traversal algorithm?

What are some advantages and disadvantages of using binary trees compared to other data structures for storing and retrieving data?

8.6 Case study:

Using Binary Trees in a Database Management System

Our client, an international online marketplace, was experiencing significant issues with their database management. The system handled millions of products, each identified by a unique ID, and with several associated attributes such as price, category, and seller information.

The database was initially structured using a linear data arrangement. However, as the volume of data increased, search operations became increasingly slower. Inefficient data retrieval was not just an internal issue; it also affected end-users who experienced delays in product searches and transaction processing.

After a comprehensive analysis, we proposed to restructure their database using a Binary Search Tree (BST) architecture. The binary tree structure was chosen due to its favorableO(log n) time complexity for search, insertion, and deletion operations. Each product node in the BST contained the product ID as the key and other product attributes as associated data.

The implementation led to significant improvements. Query times for searching products were dramatically reduced, improving the overall user experience. Additionally, operations like insertion of new products and deletion of old products were more efficient.

Beyond the initial improvements, the BST structure also provided a solid foundation for future scalability. As the marketplace continues to grow and add more products, we can confidently maintain efficient database operations thanks to the chosen binary tree structure.

Questions:

1. What were the primary issues the client was experiencing with their initial linear data arrangement?
2. How did the implementation of Binary Search Tree architecture improve database management for the client?
3. In what ways does the Binary Search Tree structure support future scalability of the client's marketplace?

8.7 References

1. "Data Structures and Algorithm Analysis in Java" by Mark Allen Weiss.
2. "Data Structures: Abstraction and Design Using Java" by Elliot B. Koffman and Paul A. T. Wolfgang.
3. Schaum Series, “Introduction to Data Structures”, TMH
4. R.B. Patel, “Expert Data Structures with C”, Second Edition, Khanna Book publishing Co (P) Ltd.

Unit: 9

Heaps

Learning Objectives:

1. Understand the concept of a tree as a hierarchical data structure
2. The definition and properties of a binary tree
3. Key terms related to binary trees, such as root, parent, child, leaf, subtree, etc.
5. Different tree traversal techniques

Structure:

9.1 Threaded Binary Tree
9.2 Binary Search Tree
9.3 Heap Data Structure
9.4 Heap Sort
9.5 General Trees
9.6 Summary
9.7 Keywords
9.8 Self-Assessment Questions
9.9 Case Study
9.10 References

9.1 Threaded Binary Trees

The binary trees known as threaded binary trees are those in which we use either the left child pointer, the right child pointer, or both leaf nodes to accomplish or optimize a particular type of tree traversal.

9.1.1 Advantages of Threaded Binary Tree

•There is no need for a stack because nodes in a threaded binary tree can be traversed quickly and linearly. If the stack is employed, a lot of memory and time are used.

It is more general since by merely following the thread and links, one can quickly ascertain the successor and predecessor of any node. It works quite similarly to a circular linked list.

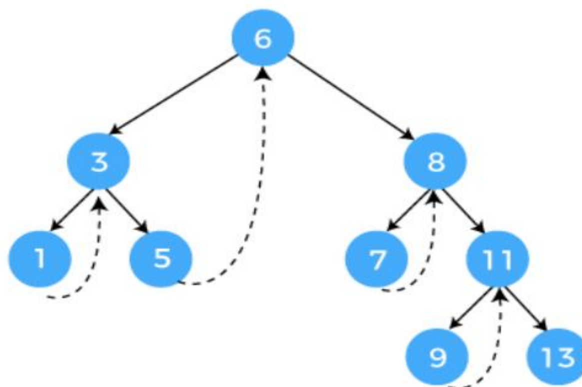
Both recursive and iterative implementations of tree traversal necessitate stack space, which scales proportionally to the height of the tree, provided threaded binary tree concepts are not employed. For a well-balanced tree with n nodes, this stack space consumption is O(log n). However, in the worst-case scenario, where the tree resembles a chain, its height becomes n, resulting in O(n) space requirement for the algorithm.

Another issue is that since nodes only hold pointers to their offspring, all traversals must start at the root. It is normal to have a pointer to a certain node, but without further information, such as thread pointers, it is impossible to return to the remainder of the tree.

9.1.2 Types of Threaded Binary Tree

Single Threaded Binary Tree

When adding information to a single threaded binary tree, we use either the left or right child node pointers of the leaf nodes.The unique threaded linkages are depicted in the diagram below by dashed arrows. Each leaf node's right child pointer in this diagram is pointing to the leaf node's inorder successor node.

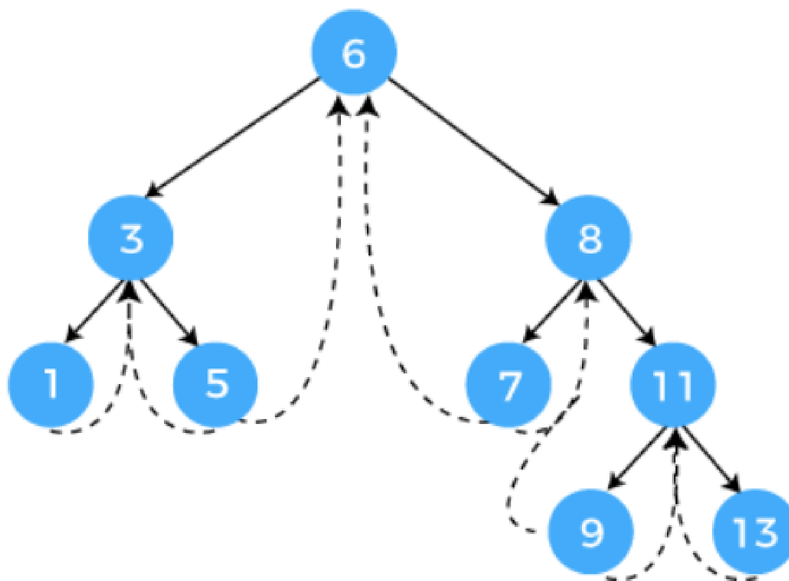


A thread will show up in either the right or left link field of a node in one-way threaded binary trees. It will point to the following node that will appear after

executing an order traversal if it exists in the right link field of a node. Right threaded binary trees are these types of trees. A thread will point to the node's predecessor if it occurs in the left field of the node. Left threaded binary trees are these types of trees. Given that they lack the final benefits of right threaded binary trees, left threaded binary trees are utilised less frequently. The right link field of the last node and the left link field of the first node in one-way threaded binary trees both contain a NULL. In order to distinguish threads from normal links they are represented by dotted lines.

Double threaded Binary Tree

In a double threaded binary tree, any additional information is stored in both the left child node pointer and the right child node pointer of the leaf nodes.The unique threaded linkages are depicted in the diagram below by dashed arrows. The left child pointer of each leaf node in this diagram points to the leaf node's inorder predecessor, while the right child pointer points to the leaf node's inorder successor.



In two-way threaded binary trees, a node's right link field is replaced by a thread that links to the node's inorder successor, and a node's left link field is replaced by a thread that points to the node's inorder predecessor.

When only a pointer to a single node within a binary tree is available, navigating the entire tree can be challenging or even impossible. For instance, reaching the

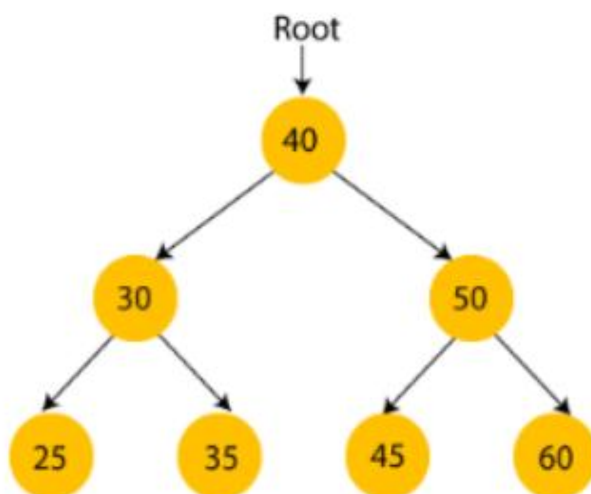
succeeding node from a leaf node solely with its pointer might be unattainable, as leaf nodes lack children or successors.To facilitate the identification of the "next" node, threaded trees incorporate supplementary information in some or all of their nodes. This additional data enables traversal without recursion and eliminates the need for additional storage proportional to the average tree depth or total number of nodes, which becomes necessary in cases where the tree is heavily skewed or linked.

9.2 Binary Search Tree

The binary search technique can be used to quickly search for any key in a binary search tree (BST), which is a sorted binary tree. The BST must possess the following qualities in order to be sorted: Only a key that is smaller than the node's key is present in the left subtree of the node.

Binary Search and Trees make up the two primary programming components of Binary Search Trees. One of the quickest and best-optimized search algorithms is binary search. The most popular type of tree is a binary tree, in which each node can have a maximum of two children. To make things simpler, all nodes—including the root node—will have a maximum of two children, not more. Binary Search Trees are unique varieties of Binary Trees with a few unique characteristics. Like the traditional Binary Search algorithm, the concept. The fact that the data collection is represented using Binary Trees is the only minor variation.

9.2.1 Working:



A binary search tree organizes its elements in a specific order. According to this structure, the value of a left node in a binary search tree must be less than the value of its parent node, while the value of a right node must be greater than that of its parent node. This rule is recursively applied to both the left and right subtrees of the root node.

The root node in the above graphic is 40, and all the nodes in the left and right subtrees, respectively, are smaller than and larger than the root node.

The left child of the root node is bigger than its left child and smaller than its right child, as can also be seen. As a result, it also meets the binary search tree's property. As a result, the tree in the above image is what is known as a binary search tree.

9.2.2 Advantages:

We always have an indication as to which subtree has the desired element, making it simple to search for an element in the Binary search tree.

Insertion and deletion operations in BST are quicker than those in arrays and linked lists.

9.2.3 Searching in BST

In the search process of a binary search tree (BST), comparing key values is a crucial step. Here's an algorithm outlining the technique:

1. Check if the tree is NULL; if not, proceed.
2. Compare the search key with the key of the BST's root.
3. If the key is smaller than the root, search in the left subtree.
4. If the key is larger than the root, search in the right subtree.
5. If the key matches the root, report "search successful."
6. Repeat steps 3, 4, or 5 to explore the subtree further.

9.2.4 Applications of the BST

BSTs are instrumental in indexing data and facilitating various search methods. They serve as a foundation for implementing diverse data structures and enabling efficient data storage and retrieval in decision support systems. Additionally, BSTs find application in computer simulations, where they offer swift data retrieval capabilities. These trees are also integral to the implementation of fast autocomplete systems.

Moreover, BSTs are utilized in constructing decision trees, pivotal in machine learning and artificial intelligence for decision modeling and outcome forecasting.

9.3 Heap Data Structure

In a heap, a distinctive tree-based data structure, the tree conforms to the characteristics of a full binary tree. In a max heap, every node is larger than any of its children, and the root node holds the largest key among all nodes. Conversely, in a min heap, the root node's key is the smallest, and any node is smaller than its child nodes, adhering to the minimum heap property.

9.3.1 Operations on Heap

The algorithms for a few of the significant operations carried out on a heap are detailed below.

Heapify

Heapify is the process of converting a binary tree into a heap data structure. A Min-Heap or a Max-Heap can be made with it.

There are two ways to apply heapify.

up_heapify(): It utilizes a bottom-up strategy. By moving in the direction of the rootNode, we can determine if the nodes are adhering to the heap property, and if not, we may do a number of actions to force the tree to do so.

down_heapify(): It utilizes a top-down strategy. By moving in the direction of the leaf nodes, we can determine if the nodes are adhering to the heap property, and if not, we can do certain operations to force the tree to do so

Insertion

The following are the steps for applying insertion. At the bottom of the pile, add the new element. Since the newly added element has the potential to alter the Heap's attributes. In order to maintain the heap's properties from the bottom up, we must execute the up_heapify() function.

`

Deletion

The following are the steps for applying deletion. The last element in the heap takes the place of the element that will be destroyed.Remove the last thing from the pile.
Now that the final piece has been added to the heap, it might not follow its properties, therefore the down_heapify() action is necessary to preserve the heap's structure. The top-bottom heapification is carried out by the down_heapify() function.The element that is located at the heap's root node is deleted as per regular procedure.

In heap data structures, operations like Find-max (or Find-min), also known as Peek/Top, involve accessing the root node, which holds the highest or lowest element in a max-heap or min-heap, respectively.
Similarly, the Extract Min/Max operation returns and removes the maximum or minimum element in a max-heap or min-heap, respectively, with the root node containing the largest element.
Heap sort, an efficient sorting algorithm, utilizes the properties of a heap to process the elements of an array. Initially, the array is rearranged to form a heap, where the root node represents the minimum or maximum element, depending on the type of heap.
The sorting process involves repeatedly removing the root element from the heap and appending it to the sorted section of the array. This process continues until all elements are sorted.
The steps involved in heap sort can be implemented using functions like heapify to construct and maintain the heap, and heapSort to orchestrate the sorting process.

9.5 General Tree

A general tree is a hierarchical data structure composed of nodes, each of which can have any number of child nodes. Unlike a binary tree, there is no restriction on the number of child nodes a node can have.
Nodes in a general tree are interconnected by edges, and each node may possess zero or more child nodes. Unlike binary trees where each node can have a maximum of two child nodes, there is no such constraint in a general tree.
Various data structures such as linked lists and arrays can be employed to represent general trees. Typically, each node contains a value and references or pointers to its child nodes.

In a C implementation, each tree node is represented by the struct TreeNode, which includes pointers to the first child node (firstChild) and the next sibling node (nextSibling), along with a value field to store the node's value.
The addChild function appends a child node to an existing parent node, while the createNode function generates a new node with the provided value.
The printTree function traverses the tree in a preorder manner and outputs the values of the nodes.
In the main function, utilizing the provided example, nodes are created and the tree structure is constructed. Subsequently, the printTree function is invoked to print the tree, resulting in the output: Tree: A B D E C F.

9.6 Summary

- A threaded tree is a binary tree enriched with additional threads or links, enabling efficient traversal without the need for recursion or a stack. These threads establish connections between specific nodes and their in-order predecessors or successors, facilitating rapid navigation within the tree.
- A binary search tree (BST) is a binary tree where each node's value is greater than all values in its left subtree and less than all values in its right subtree. BSTs offer efficient searching, insertion, and deletion operations, boasting an average time complexity of O(log n) for balanced trees.
- A heap is a complete binary tree where each node's value is either greater than or equal to (max-heap) or less than or equal to (min-heap) the values of its children. Typically implemented using arrays, heaps are utilized to efficiently locate the maximum or minimum element (root) of the tree.
- Heap sort is a comparison-based sorting algorithm leveraging the properties of a heap data structure. It is particularly beneficial when a sorted array is needed, but the input size is too large to fit in memory.
- General trees are hierarchical data structures in which each node can possess an arbitrary number of child nodes. Unlike binary trees, general trees impose no limitations on the number of child nodes per parent node.

9.7 Keywords

- Optimization: Threaded trees streamline traversal operations, like in-order traversal, by establishing extra threads or links among nodes. This obviates the

necessity for recursion or an explicit stack, leading to more efficient tree traversal.

- Ordered: A binary search tree organizes data hierarchically, with each node's value ordered concerning its child nodes. The left child holds a lesser value, while the right child holds a greater one. This organization facilitates efficient searching, insertion, and deletion.
- Priority: A heap is a complete binary tree adhering to the heap property, providing a priority-based data structure where the highest (or lowest) priority element resides at the root. Heaps are extensively employed in priority queues and algorithms necessitating efficient access to the maximum (or minimum) element.
- Sorting: Heap sort, a comparison-based sorting algorithm, leverages the heap data structure. It iteratively extracts the maximum (or minimum) element from the heap, placing it in the sorted portion of the array. With a time complexity of O(n log n), heap sort is particularly advantageous for sorting large data sets.
- Hierarchy: General trees exhibit hierarchical structures wherein each node can accommodate an arbitrary number of child nodes. They are instrumental in representing relationships with varying degrees of branching, such as file systems, organizational hierarchies, or family trees. General trees offer a dynamic and flexible approach to organizing data.

9.8 Self-Assessment Questions

1. How does a threaded tree optimize in-order traversal compared to a regular binary tree?
2. How can the balance of a binary search tree be maintained to ensure optimal performance?
3. Explain the process of heapifying an array and converting it into a heap.
4. Describe the steps involved in the heap sort algorithm.
5. What are some real-world applications of general trees?

9.9 Case study:

Using Binary Trees in a Database Management System

Our client, an international online marketplace, was experiencing significant issues with their database management. The system handled millions of products, each identified by a unique ID, and with several associated attributes such as price, category, and seller information.

The database was initially structured using a linear data arrangement. However, as the volume of data increased, search operations became increasingly slower. Inefficient data retrieval was not just an internal issue; it also affected end-users who experienced delays in product searches and transaction processing.

After a comprehensive analysis, we proposed to restructure their database using a Binary Search Tree (BST) architecture. The binary tree structure was chosen due to its favorableO(log n) time complexity for search, insertion, and deletion operations. Each product node in the BST contained the product ID as the key and other product attributes as associated data.

The implementation led to significant improvements. Query times for searching products were dramatically reduced, improving the overall user experience. Additionally, operations like insertion of new products and deletion of old products were more efficient.

Beyond the initial improvements, the BST structure also provided a solid foundation for future scalability. As the marketplace continues to grow and add more products, we can confidently maintain efficient database operations thanks to the chosen binary tree structure.

Questions:

1. What were the primary issues the client was experiencing with their initial linear data arrangement?
2. How did the implementation of Binary Search Tree architecture improve database management for the client?
3. In what ways does the Binary Search Tree structure support future scalability of the client's marketplace?

9.10 References

1. "Data Structures and Algorithm Analysis in Java" by Mark Allen Weiss.
2. "Data Structures: Abstraction and Design Using Java" by Elliot B. Koffman and Paul A. T. Wolfgang.
3. Schaum Series, “Introduction to Data Structures”, TMH
4. R.B. Patel, “Expert Data Structures with C”, Second Edition, Khanna Book publishing Co (P) Ltd.