Master of Computer Applications (MCA)

Advance Data Structure and Algorithms Analysis (DMCASE106T24)

Self-Learning Material (SEM 1)

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

PREFACE

In today's rapidly evolving technological landscape, the understanding and application of algorithms and data structures are paramount for any aspiring computer scientist or IT professional. This document aims to provide a comprehensive guide on these fundamental concepts, specifically tailored for students pursuing their MCA (Master of Computer Applications). Our objective is to break down complex topics into manageable sections, ensuring that learners grasp both the theoretical and practical aspects of data structures and algorithms.

This book begins with an introduction to the basic principles of data structures, exploring their importance in effective data management and memory utilization. We delve into various types of data structures such as arrays, linked lists, stacks, and queues, elucidating their individual characteristics and use cases. Each chapter is designed to build on the previous one, gradually increasing in complexity and covering more advanced topics like trees, graphs, and hash tables.

A significant portion of this book is dedicated to the analysis of algorithms, emphasizing the critical role of time and space complexity. By understanding concepts such as Big O notation, readers will be equipped to evaluate the efficiency of different algorithms, making informed decisions in their software development projects.

The practical applications of these concepts are highlighted through case studies and real-world examples, demonstrating how efficient data structures and algorithms can solve complex problems in various domains. Additionally, each chapter includes self-evaluation questions to reinforce learning and ensure that key concepts are well understood.

This preface sets the stage for a detailed exploration of algorithms and data structures, aiming to provide readers with the knowledge and skills necessary to excel in the field of computer science.

TABLE OF CONTENTS

UNIT: 1

INTRODUCTION - DATA STRUCTURES

Learning Objectives:

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

Structure:

1. Data Structures Introduction
2. Data Manipulation
3. Asymptotic Notation
4. Time/Space Trade-off
5. Overview
6. Key terms
7. Self- Evaluation Questions
8. Case Study
9. References

1.1 Introduction - Data Structures

The structured data of a computer program is a design for organising and save data in a way that able to be easily obtained or used. Data structures give for efficient searching, arranging, adding, and removing of data alongside for the oversight of large amounts of data. The data structure that is picked for a task built upon the variety and quantity of information that will be reclaimed, the tasks that is demands to be done to the information, and the program's performance needed. A schedule may execute more efficiently and quickly with less memory if data structures are used appropriately.

1.1.1 Importance:

Effective information processing: Data structures offer a technique to organise and holds data in a form that makes it possible for effective data retrieval, manipulate, and storage. 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 abstraction level that frees programmers from having to worry about the set of how data is saved and handled, allowing them to concentrate on the logical structure of the information and the procedures 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 conceptual understanding of data structures.

Primitive and non-primitive data structures are the two categories into which structured of Data can be divided.

1.2 Data Structure Operations

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

- **Traversing**a structured data is to go from element to element within it. It makes systematic trips to the data. Data Structure can be used to accomplish this. The programme used to demonstrate operation on array is shown below.

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

int main(){
        int arr[] = { 1, 2, 3, 4 };


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

        for (int i = 0; i < N; i++) {


        cout << arr[i] << ' ';
    }
    return 0;
}
```

- **Searching** is procedure of locating a certain element in a specified structured data . Whennecessary component is located, endeavour is supposed doing well. 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:

```
#include <iostream>
using namespace std;
void findElement(int arr[], int N, int K){
   for (int i = 0; i< N; i++) {

      if (arr[i] == K) {
cout<< "Element found!";
return;
      }
   }
cout<< "Element Not found!";
}

int main(){
   int arr[] = { 1, 2, 3, 4 };

   int K = 3;
   int N = sizeof(arr) / sizeof(arr[0]);


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

- **Insertion**: the procedure that is applicable to all data structures. Insertion refers to the act of addition of a new element to data structure. When the necessary is added to required data-structure, 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 inclusion 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 isprovided below
- **Deletion**: It is the procedure we use on all structured data. A data structure's element can be deleted using deletion. When the necessary componentsis removed from the data structure, the deletion operation is successful. A deletion in data structure such an array, linked list, graph, tree, etc. has 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 <iostream>
using namespace std;

void printArray(int arr[], int N)
{

	for (int i = 0; i < N; i++) {


		cout << arr[i] << ' ';
	}
}

int main()
{
		int arr[4];

		int N = 4;


	for (int i = 1; i < 5; i++) {
		arr[i - 1] = i;
	}

		printArray(arr, N);
	return 0;
}
```

```
#include <bits/stdc++.h>
using namespace std;
void printStack(stack<int> St)
{

    while (!St.empty()) {
            cout << St.top() << ' ';


        St.pop();
    }
}
int main()
{
        stack<int> St;


    St.push(4);
    St.push(3);
    St.push(2);
    St.push(1);



    printStack(St);

    cout << endl;

        St.pop();


            }
       }
       int main()
       {
                stack<int> St;


            St.push(4);
            St.push(3);
            St.push(2);
            St.push(1);



            printStack(St);

            cout << endl;

                St.pop();


            printStack(St);
            return 0;
       }
```

Some other methods

a) **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.

b) **Selection: -**

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

c) **Update**

The data in structured data is updated. By involving a condition in a loop, like the select approach, you mightchange any specific data.

d) **Sort**

Sort the information according to a specific order (descending or ascending). Different sorting techniques can be used to quickly sort data. For example, a bubble sort sorts data in O(n2) time. Numerous algorithms are available, such as "Selection sort, Rapid sort, Insertion sort, and merge sort."

e) **Merge**

It is possible to merge data from two diverse orders in an descending or ascending order. To combine sorts of data, we use “Merge Sort “.

f) **Split Data**

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

1.3 Asymptotic Notation

Based on the size of the insertion value, asymptotic notation can use to illustrate space or running time complexity of an algorithm. It is usually employed in complexity analysis to elaborate how an “Algorithm”act as the insert value size increases. “Big O”, Omega, and Theta are the three notations that are oftenly used.

- **Big O Notation (O):** The running time or storage needs of a procedure 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 a procedure execution time, for instance, is O(n), it signifies that it scales linearly with input size, n, or a smaller number.

- **Omega notation (Ω):**A lower bound on the growth rate of a procedure running time or space usage is allowed by this notation. It illustrates the best case, i.e., the less amount of time or space an algorithm requires to solve a task. For a minute, if an Algorithm's running time is (n), it symbolize that the algorithm's execution time increase linearly as the input size increases to n or more.

- **"Theta notation" (Θ):**Thissignsprovides upper and lower bound in the increase of time or physical location needed to execute an algorithm. It indicates a typical case, i.e., the time limite or space normally required for an algorithm to complete task. For instance, if an algorithm's execution time is (n), i.e. it increase steady growthas the size of the input n

1.4 Time/Space trade-off

- Method for resolve problems in: Either in fewer time limits and with good space, or in a small space with more of time.
- The best algorithm is one that has capability in solving a problem that take less memory and produces results quickly. However, it cannot often be possible to fulfil both requirements at once.
- The most common case is a lookup table-based algorithm. This conveys that it is Feasible to record the responses to some queries for each credible 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 higher time-efficient algorithms you have, the less space-efficient they would be.

1.4.1 Applications:

Compressed or Uncompressed data: The problem of storage of data can be proposed through a space-time trade-off. Uncompressed information grab up more space but loads faster. However, compressed data need less storage space but higher processing time to address the decompression method. Working closely with compressed data is possible in a several situations. It is faster to work with compression than without compression in that example of compressed bitmap indices

Rendering or Stored images: In this situation, saving the information and rendering it as an image would take up height memory but take limited amount of time; similarly, caching an image is quicker than rendering but consume more memory.

Smaller code or Loop Unrolling: Small code takes up lower memory, but it takes height time to compute because it have to move back to the start of the loop after each iteration. Loop unrolling can enhance binary size butbetter execution performance. Although it holds a good amount of memory space, it holds less time to process.

Lookup tables or Recalculation: A lookup table's implementation option allows for full table inclusion, which reduces computation time but needed good amount of memory. It can recalculate, or compute table entries, as critical, approachingcomputation times while shortening memory needs.

```
class MyClass {
    // 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. The volumeof auxiliary space is less. However, even though it takes up good amount of space, our focus on to make the process low time-wasting. The goal is to use Dynamic Programming to optimise the strategy by memorising the overlay sub problems.

```
class MyClass {
static int Fibonacci(int N){
int[] f= new int[N + 2];
   int i;

f[0] = 0;
f[1] = 1;

   for (i = 2; i<= N; i++) {
```

```
        f[i] = f[i - 1] + f[i - 2];
    }
        return f[N];}

public static void main (String[] args) {
    int N = 5;

System.out.println(Fibonacci(N));
}
}
```

1.5 Summary

- Structured data is a way of organising and holding information in a computer program so that it can be accessed and used properly.
- Structured data provide a way to organise and store information in a way that allows for efficient retrieval, manipulation, and storage of data.
- Structured informationcan be categorised into two parts: primitive data structures and non-primitive data structures
- Traversing a Structured informationmeans to visit the component holds in it. It visits information in a systematic manner.
- Searching implies to find specific components in the given data-structure. It is considered as successful when the required componentsis found.

1.6 Keywords

- **Code Reusability:**Structuredinformationcan be utilizedas building blocks in various algorithms and programs, making it way simpler to reuse code.

Asymptotic Notation: It is a manner to describe the running time or space complexity of an algorithm relies on the size of input. It is often used in analysis of complexity to explain how an algorithm functioning as the size of thinput grows.

*** Time/Space trade-off**: A trade off is a situation where one thing increases and another thing reduces, It is a way to solve a problem either in less time and by using more space, or In very little space by spending a long amount of time.

*** Lookup tables or Recalculation**: In this table, an implementation can added the whole table which decreases computing time but enhance the amount of memory critical. It can recounted i.e., compute table entries as needed, enhancing computing time but reducing memory needs.

1.7 Self- Evaluation Questions

1. Define 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 abstruse 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 data structureGraph, 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.andChattopadhyayMatangini., “Data Structure through C language”, BPB publications.

UNIT: 2

ARRAY AND LINKED LISTS

Learning Objectives:

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

Structure:

a) Linear Lists
b) Address Calculation in Single-Dimensional Arrays
c) Address Calculation in Multidimensional Arrays
d) Operations on Arrays
e) Sequential Search
f) Binary Search
g) Summary
h) Keywords
i) Self-Assessment Questions
j) Case Study
k) 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 dyanamic 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:

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 components 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;
      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.

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.

Deletion: This process removes an element from the array. Just like insertion, deletion may require shifting elements to fill the empty space.

Updating: Updating an element in an array involves modifying the value at a specific index.

Searching: Searching an array involves detect the index or value of a particular element within the array.General search algorithms include linear search and binary search.

Sorting: Sorting arranges the components of an array in a particular order, such as ascending or descending.General sorting algorithms contains bubble sort, insertion sort, merge sort, and quicksort.

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.

```
import java.util.Arrays;

public class WorkonArray {
   public static int[] mergingArr(int[] array1, int[] array2) {
          int len1 = array1.length;
          int len2 = array2.length;
          int[] mergedArray = new int[len1+len2];

          System.arraycopy(array1, 0, mergedArray, 0, len1)
          System.arraycopy(array2, 0, mergedArray, len1, len2)

          return mergedArray;
   }
   public static void main(String[] args) {
                int[] array1 = {1,3, 5};
                int[] array2 = {2, 4, 6};
                int[] mergedArray = mergingArr(array1, array2);
                System.out.println("Merged Array: "+
Arrays.toString(mergedArray));
   }
}
```

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 prior to a match is discovered or all the structure has been read across.

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

Start from the origin of the data structure.

Distinguish between the target element with the current element being examined.

If the present the target element, the search is successful, and the position or index of the element is returned.

If the current element does not match, move to the next component in the structie.

Redo steps 2-4 until a match is found or the end of the structure is reached.

If all the 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 difficulty of O(n), where n is the count of component in the collection, it may not be effective for larger collections. There can be different search methods like as binary search (applicable exclusively on sorted collections) or hash-based searching may be more effective for bigger data sets. Following is a code to carry out the operation of sequential search

```
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 sequential search method do a sequential search on the given integer array to find the objective value. It iterates through each components 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 objective value, calls the method, and prints the result accordingly.

2.6 Binary Search

The position or index of a target element inside a sorted collection of data can be found using the search process known as binary search. The field of search is divided in half continually until the element that is wanted is located or turns out to be non-existent.

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

Commence with the middle component of the sorted collection.

Compare the middle component with the target element.

If the middle component matches the aim, the search is successful, and the position or index is returned

If the middle element is greater than the target, repeat the search process on the left half of the collection.

If the middle component is less than the aim, repeat the search process on the right half of the collection.

Redo steps 2-5 until a match is found or the search space is empty.

The sorted nature of the collection is advantageous to a binary search, which at each step can eliminate half of the remaining search space. Compared to sequential search, this causes a search to be substantially faster.The time difficulty of binary search is logarithmic, denoted as O(log n), where n is the number of components in the

collection. This logarithmic complexity arises from the fact that the search space is divided in half at each step, resulting in a divide-and-conquer approach. As a result, binary search is highly efficient, especially for large collections.

Following is a code to represent the binary search operation.

```
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");
    }
  }
}
```

In this present code, the binary Search method performs a binary search on the given sorted integer array array to find the target value. It maintains two pointers, left and right, which represent the range of indices to search within. The method iteratively updates these pointers and distinguish the middle component with the aim until a match is found or the search space is empty.

The main method demonstrates an example usage of the binary Search formula. It initializes a sorted array and a objective value, calls the method, and prints the result accordingly.

2.7 Summary

- 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.
- 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 formula for calculating the address of an element in a single-dimensional array is: address = base_address + (element_size * index)
- 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.
- The formula for calculating the address of an element is: address = base_address + (element_size * (row * columns + column))
- Accessing each element in an array can be done by using their indices.
- Insertion involves inserting elements to an array.
- 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 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 till a match is discovered or the all structure has been read across.

2.8 Keywords

- Linear List: A linear list is a type of data structure that represents a sequence list of components stored in continuous memory locations, where each element is accessed by its position or index.
- 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.
- 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.

- 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.
- Sequential Search: Sequential search is a type of simple search algorithm used to find the presence and location of a objective component within a set of data. It involves systematically examining each component in a data structure till a match is found or all the 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 objective 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.

2.10Case 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 algorithm works by continuously dividing the sorted list into two halves parts 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

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

UNIT: 3

STACK AND QUEUE

Learning Objectives:

- Know the concept of Stacks
- Learn to implement them
- Understand their application

Structure:

1. Stacks
2. Application of Stacks
3. Summary
4. Keywords
5. Self-Assessment Questions
6. Case Study
7. References

3.1 Stacks

Stacks is a type of Data Structure that is linear and it is based on the concept of Last In First Out(LIFO). It is an abstract data type that use the fundamental of Last-In-First-Out (LIFO) principle wherein component can be added and removed from only one end, known as the height of the stack.

The two fundamental functions associated with a stack and they are:

Push: With this action, a component is added to at the peak of the stack. Already used components are moved one position forward and the new element rises to the peak of the stack.

Pop: The component at the peak of the stack is removed by this operation. This data structure refreshed after removing the highest element, which was added most recently.

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

Peek/Top: Without removing the component from the stack, this action returns the value of the peak element on stack. It enables you to look at the top element without changing this data structure .

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 components 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:

```
privateintstackArray
public Stack size)!
publicvodstv
topma
```

```
System.out.printing Stack is full. Cannot push element."
System.out.printinPushed element:" value
1 public int pop
System.out.printing Stack is empty. Cannot pop element."
return
toppedvaluestackray top-
System.out.printing Popped element:"poppedValue)
returnpoppedValues
publicint peek (
System.out.printing
tunstackmatop
publicbooleanisEmpty()
return (top)
Stack is empty. Cannot peek element."
public static void maintingang
Stack stack-new Stack
stack.push(20)
stack.push(20
stack.push(30
stack.pop element stack peek
```

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 components onto the stack, popping components from the stack, peeking at the top component, and analyze 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 components 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:

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

Infix to Postfix: Stacks can be used to transform an arithmetic expression from infix notation (operators between operands) to postfix notation (operators after operands). This conversion simplifies expression evaluation and helps in removing the ambiguity of operator precedence and parentheses.

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

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.

Arithmetic Expression Evaluation: Stack data structure is an instrumental in assesses 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 a vacant stack.
Scan the statement from left to right.
If an operand is encountered, push it onto the stack.
If an operator is encountered, pop the needed number of operands from the stack, perform the function, and push the output back onto the stack.
Redo steps c and d till 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 associatively of operators.

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

```
import java.util.Stack;

public class InfixToPostfix {
    public static int getPrecedence(char operator) {
        switch (operator) {
            case '+':
            case '-':
                return 1;
            case '*':
            case '/':
                return 2;
            case '^':
                return 3;
            default:
                return -1;
        }
    }
    public static String convertToPostfix(String infix) {
     StringBuilder postfix = new StringBuilder();
     Stack<Character> stack = new Stack<>();
```

```
For (char ch: infix.toCharArray()) {
// If the character is an operand, append it to the postfix expression
encountered
if (Character.isLetterOrDigit(ch)){
        postfix.append(ch):
// If the character is an opening parenthesis, push it to the stack
else if (ch == "") {
        stack.push(ch);
// If the character is a closing parenthesis, pop operators from the stack and append them to
the postfix expression until an opening parenthesis is
else if (ch")){
        while (!stack.isEmpty() &&stack.peek() != '() {
                postfix.append(stack.pop());
if (!stack.isEmpty() &&stack.peek() != "") {
        throw new IllegalArgumentException("Invalid infix expression");
        stack.pop():
else {

while (!stack.isEmpty() &&getPrecedence(ch) <= getPrecedence(stack.peek())) {
postfix.append(stack.pop());
        stack.push(ch):
while (!stack.isEmpty()) {
        if (stack.peek()==')'){
                throw new IllegalArgumentException("Invalid infix expression");
```

```
            postfix.append(stack.pop());
returnpostfix.toString();
public static void main(String[] args) {
        String infixExpression = "A+B (C-D)";
        String postfixExpression = convertToPostfix(infixExpression);
System.out.println("Infix Expression: "+ infixExpression); System.out.println("Postfix Expression: "+postfixExpression);
```

In this example, the convertToPostfix() method takes an infix expression as insertion value and converts it to postfix notation using a stack. The method iterates over each character of the infix expression and performs the necessary conversions based on the character's type (operand, operator, parentheses).

The getPrecedence() method assigns precedence values to the operators (+, -, *, /, ^) to determine their order of evaluation.
The main method demonstrates the conversion by passing an infix expression "A + B * (C - D)" and printing the resulting postfix expression.

3.3 Summary

- Stack is a type of Data Structure behave linear and it is based on the principle of Last In First Out(LIFO).
- It is an abstract data type that is based on the principle of Last-In-First-Out (LIFO) principle wherein components can be added and removed from only one from one side end, known as the top of the stack.
- The component at the peak of the stack is removed by the pop operation.
- IsEmptyfunction inform us whether the stack is empty. It gives back a Boolean result that indicates whether or not the stack has any entries.
- The array is used to hold the components of the stack, and an additional variable called top holds the data of the index of the topmost components in the stack.
- Stacks are used to transform arithmetic expressions from one form to another.
- Stacks can also be used to convert an arithmetic expression from infix notation to prefix notation.
- 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 getPrecedence() method assigns precedence values to the operators (+, -, *, /, ^) to determine their order of evaluation.

3.4 Keywords

- **Stack:** It is a data structure that work on Last-In-First-Out (LIFO) principle, where components are added and removed from only one side end, known as the top of the stack.
- **LIFO:** Last-In-First-Out, working principle of stacks where the last component inserted into the stack is the first component to be removed.
- **Pop:** An operation performed on a stack data structure to remove the component at the top of the stack.
- **IsEmpty:** A process that determines whether a stack is vacant or not. It provides a Boolean result indicating whether the stack has any entries.
- **Array:** A data structure used to hold the elements of a stack, where the additional variable called "top" frequently trace the index of the topmost component in the stack.

3.5 Self-Assessment Questions

- Describe the concept of a stack and its main operations.
- How can a stack be implemented using an array? Provide the necessary algorithms for push and pop operations.
- Describe the application of stacks in converting infix expressions to postfix expressions. Provide an example to illustrate the conversion process.
- Discuss the role of stacks in evaluating arithmetic expressions. Explain how a stack can be used to perform the evaluation.
- What are the supplication 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 applicant 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 applicant is taken to the previous page (i.e., the top element of the Back Stack is popped and displayed).

If the applicant decides to go "Forward," the process is reversed: the URL of the current page is pushed into the Back Stack, and the top element of the Forward Stack is popped and displayed. However, if the user navigates to a new page without hitting the "Back" button, the Forward Stack is cleared since the user has branched off in a new direction of browsing.
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

TREES

Learning Objectives:

- Understand the concept of Queues
- Know its implementation
- Learn about its applications

Structure:

1. Queues
2. Implementation of queues using Array
3. Applications of a Linear Queue
4. Applications of a circular queue
5. Applications of a Double ended Queue (Deque)
6. Summary
7. Keywords
8. Self-Assessment Questions
9. Case Study
10. References

4.1 Queues

A linear data structure that used the concept of First-In-First-Out (FIFO) principle is called queue. It is designed to hold elements in a particular order, which means the element which is added first, will be removed first. Queues have two primary operations: dequeue and enqueue.

- Dequeue: This operation removes the component at the front of the queue. next element in line becomes the new front of the queue when you use this operation.
- Enqueue: This operation is used for adding an component to the end of the queue. It becomes the last component in the queue when you enqueue an element,

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

Peek/Front: Without removing it, this operation returns the element at the front of the queue and allows you to examine the next element that will be dequeued.

IsEmpty: This method is used to check whether the queue has no elements. If there are no elements, it returns true; otherwise, it returns false.

Queues can be used in 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

An array can be used to implement queues, with each element being kept in a single, 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
typedefstruct {
int queue[MAX_SIZE];
int front;
int rear;
int size;
} Queue;
voidinitQueue(Queue" q) {
q->front=0;
q->rear=-1;
q->size = 0;
boolis_empty(Queue" q) {
}
return q->size == 0;
boolis_full(Queue* q) {
return q->size == MAX_SIZE;
voidenqueue(Queue* q, int item) {
if (is_full(a)) {
}
printf("Queue is full. Unable to enqueue.\n");
return;
q-rear (q-rear + 1) % MAX_SIZE;
q->queue[q->rear] = item;
q->size++;
Jintdequeue(Queue* q) {
if (is empty(a)) {
}
printf("Queue is empty. Unable to dequeue.\n");
return -1;
int item = q->queue[q->front];
q->queue[q->front]=0;
q->front (q->front + 1) % MAX_SIZE;
q->size--;
return item;
```

```
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 describe a struct Queue that contains integer variables for the front, rear, and size of the queue and an array to store the components

The initQueue() 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 has no element or full, respectively.

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

The dequeue() function removes and returns the front component of the queue. If it displays an error message and returns -1 it means queue is empty.

Without removing it, the peek () function returns the front element.

The get_size() function is used to return the current size of the queue.

By enqueuing three elements, peeking at the front element, and dequeuing all elements, we demonstrate the usage of the queue in the main () function

4.3 Applications of a Linear Queue

Linear queues, also called simple queues, are required in various domains. Some common applications of linear queues are mentioned below:

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 aloted 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 the timings of their call. Agents handle the calls in a first-come-first-served manner, ensuring fairness in addressing customer inquiries or concerns.

Breadth-First Search (BFS): It is a graph traversal algorithm. Its function is to explore each and every 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

There are numerous uses for circular queues in various fields. They are often referred to as circular buffers or ring buffers. The following are some typical uses for circular queues:

- Memory Management: Memory management systems, such as embedded systems or operating systems frequently use circular queues. 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 volume of data needs to be processed in real time.

- Producer-Consumer Problem: Circular queues are commonly used to solve the producer-consumer problem, which include 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. Producersenqueue 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. Some common applications of a double-ended queue are given below:

- 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 insertion as well as

deletion of elements from both ends, which makes 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. A deque can be used to implement priority-based or time-based job scheduling algorithms by allowing insertion and deletion of jobs from both ends so that 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 instances 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 linear data structure that use the concept of First-In-First-Out (FIFO) principle is a queue. It holds component in a particular order, where the element added first is the one that will be removed first.
- Two primary operations of queue: enqueue and dequeue.
- IsEmpty checks whether the queue is empty.
- By using different type of data structures, such as linked lists or arrays, queues can be implemented.
- Queues can be implemented using an array, where the elements are stored in a contiguous block of memory.
- The enqueue () function is used to add an component to the back of the queue. If error message appears it means the queue is full.
- The peek () function giving back the front element without removing it.
- Linear queues, also known as simple queues, have several applications in various domains.
- A linear queue is frequently used in printer spooling systems to handle print jobs. Print requests sent by many users at the same time are queued and handled in the order they are received.
- 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 positioned in a queue based on the 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 includes 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 called deque, is known for its versatile data structure. It 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 type of data structure that using the concept of the First-In-First-Out (FIFO) principle. It is designed to hold elements in a particular order. It means that the element which is added first will be removed first.
- **Enqueue:** Enqueue is an operation in a queue that adds an component to the end (rear) of the queue. It will display an error message if the queue is full.
- **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.
- **IsEmpty:** IsEmpty is a method or function used to check whether a queue is empty or not. If the queue contains no elements it will return true, and vice versa.
- **Linear Queue:** A linear queue, also known as a simple queue, is a type of queue where components 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. Clicking on a video adds the request to a queue, which is then handled in a 'First In, First Out' (FIFO) manner. This assures a systematic and unbiased processing 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 deletion of movie recommendations from both ends (rear - the least and front - the most recommended), 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

GRAPHS

Learning Objectives:

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

Structure:

- Linked Lists
- Types of Linked Lists
- Summary
- Keywords
- Self-Assessment Questions
- Case Study
- References

5.1 Linked List

It is a type of data structure that is built of a series of nodes, each of which has a reference (or link) to the node after it in the sequence as well as a data element. Unlike arrays, this data structure do not require memory allocation. Each node in a linked list can be dispersed 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 this data structure contains two fields: a data field to store the actual data element and a pointer (or link field) to the next node in the sequence.
- Head: The head of a this list is a pointer (or reference) to the first node in the list.
- Tail: The tail of a linked list is a pointer (or reference) to the last node in the list.

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

- Insertion: Adding a new node to the list.
- Insertion at the starting: Make a new node, update its link to point to the current head, and update the head to point to the new node.
- Insertion at the end: Make 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 node.
- Insertion at a specific position: Traverse the list to the desired position, make a new node, and update the links of the adjacent nodes to include the new node.

Deletion: To remove a node from the list.

- Deletion from the beginning: Update the head to point to the next node and free the memory of the removed node.
- 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.

  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.

This is a dynamic data structures, allowing effectual 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 evaluate a specific component in this data structure requires traversing the list from the starting or the previous nodes.

5.2 Types of Linked List

Singly Linked List

This is the most elementary kind of linked list, where each node has a pointer to the following node with the same data type and some data. When a node has a pointer to the next node, it indicates that it has the address of the node after it in the sequence stored on it. A single linked list allows the traversal of data only in one direction.

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

  static void printList(Node n)
  {
          while (n != null) {
        // Print the data
        System.out.print(n.data + " ");
        n = n.next;
     }
  }
```

```
public static void main(String[] args)
  {
     Node head = null;
     Node second = null;
     Node third = null;
     head = new Node();
     second = new Node();
     third = new Node();
 head.data = 1;
 head.next = second;
 second.data = 2;
  second.next = third;
 third.data = 3;
  third.next = null;

  printList(head);
   }
}
```

Circular Linked List

Circular Linked List

In this list the last node holds the pointer to the first node of the list. This list can be traversed starting at any node and going forward or backward until it ends at the node where we started. This list, thus, has no beginning and no end.

```
import java.util.*;
class MyLinkedList {
 static class Node {
      int data;
      Node next;
   };
 static Node push(Node head_ref, int data)
   {
      Node ptr1 = new Node();
      Node temp = head_ref;
      ptr1.data = data;
      ptr1.next = head_ref;
 if (head_ref != null) {
         while (temp.next != head_ref) {
            temp = temp.next;
         }
         temp.next = ptr1;
      }
      else
         ptr1.next = ptr1;

      head_ref = ptr1;
      return head_ref;
   }
static void printList(Node head)
   {
      Node temp = head;
      if (head != null) {
         do {
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

This list, also known as a circular two-way linked list, is a more complicated kind of linked list that has pointers to both the prior and succeeding nodes in the sequence. The differentiation that exists between a singly list and a circular list also applies to doubly linke and circular doubly lists. The circular doubly linked list does not enclose 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 last = (start).prev;
```

```
Node new_node = new Node();
new_node.data = value;
new_node.next = start; new_node.prevlast;
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;
templast;
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"
display();
+"linked list is: ");
```

Summary

- A linked list is a kind of data structure built of a series of nodes, each of which has a reference (or link) to the node after it in the sequence as well as a data component.
- A linked list allows traversal of the list despite the fact that each node may be scattered among many memory locations.
- The head of a linked list is a pointer (or reference) to the first node in the list.
- Each node in a linked list have two fields: a link field (or pointer) to the next node in the sequence, and a data field to store the actual data element.
- Linked lists are dynamic data structures, that allow both deletion and Insertion operations, but they require more memory for storing the links between nodes.
- A single linked list allows the traversal of data only in one direction.
- In a circular linked list, last node contains the pointer to the first node of the list.

- A doubly circular linked list, also known as a circular two-way linked list, is a more complicated kind of linked list that has pointers to both the prior and succeeding nodes in the sequence.

5.4 Keywords

- Linked List: A data structure consists of a sequence of nodes, in which each node contains a data element and a reference to the next node in the sequence. Each and every node can be located in different memory locations, and the links between nodes allow traversal through the list.
- Head: The pointer or reference 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.
- Node: A fundamental component of a linked list that stores a data element and a link to the next node. Each node contains a link field (or pointer) to connect it with the next node in the sequence, and a data field to hold the actual data.
- Circular Linked List: A type of linked list in which the last node contains a pointer that loops back to the first node, creating a circular structure. This allows traversal from one node to any other node in a continuous loop.

Doubly Circular Linked List: A more superior type of linked list that contains a pointer to both the next and the previous node in the sequence. Each node has links in both directions which enables traversal in both forward and backward directions. The last node's pointer connects with the first node, creating a circular structure.

5.5 Self-Assessment Questions

- How does a linked list differ from other data structures, such as arrays, in terms of memory representation and flexibility?
- 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?
- Explain the process of traversing a linked list and accessing individual nodes, highlighting any techniques or optimizations that can be employed.
- 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?

- 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 imply 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. 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 : 6

HASHING AND HASH TABLES

Learning Objectives:

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

Structure:

1. Linked List Operations
2. Header Linked List
3. Two-way Linked List
4. Overview
5. Key terms
6. Self- Evaluation Questions
7. Case Study
8. References

6.1 Linked List Operations

6.1.1 Traversing a Linked List

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

Set a temporary node pointer to the linked list head .

While the temporary pointer is void , perform the following steps:

a. Process the current node data.

b. Move the temporary pointer to the succeeding code by updating it to the value in the current node's link field.

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

The following code will help to understanding the traversal

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

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

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);

        current = current->next;
  }
}
```

```
int main() {

   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;

   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 explains a linked list structure (struct Node) and implements the traverse Linked List function to traverse the list and print the information of each node. 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 printing the data. Finally, the dynamically provided memory for the linked list is free 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: placing at the beginning, placing at the end, and placing at a specific position within the list. Here's an example of insertion at the beginning:

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

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

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;
}


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 insertAtBeginning function and demonstrates how to insert nodes at the beginning of a “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 at the starting of the “linked list” , and then the linked list is printed to confirm the insertion.

6.1.3 Deleting from a “Linked List”

To remove 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 removal of the last node. Here is an example implementation in C that covers the case of removing a node from the middle.

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

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

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 (current == *headRef) {
      *headRef = (*headRef)->next;
   } else {
prev->next = current->next;
   }

   free(current);
}

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;

   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

      deleteNode(&head, 2);
   printLinkedList(head);  // Output: 1 3

   return 0;
}
.
```

In this code, the focus is on the delete Node function, which deletes a node with a specific value from the linked list. The " printLinkedList " function is provided to print the components 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 starting 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:

```c
include <stdio.h>
#include <stdlib.h>
 struct Node {
   int data;
   struct Node* next;
};

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;
}

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;
}
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();
   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 "insertAtBeginning" 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 every node contains two pointers: one point to the previous node and one point to the next node. This allows to navigate 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>

struct Node {
   int data;
   struct Node* prev;
   struct Node* next;
};

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;
}
void insertAtBeginning(struct Node** headRef, int data) {
   struct Node* newNode = createNode(data);
   newNode->next = *headRef;

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

   *headRef = newNode;
}
```

```
void printForward(struct Node* head) {
   struct Node* current = head;
   while (current != NULL) {
      printf("%d ", current->data);
      current = current->next;
   }
   printf("\n");
}
 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;
 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 “CreateNode” method built a new node with the specified data and initialises the prev and next pointers to NULL. The insert At Beginning method inserts a new node at the beginning of the doubly linked list, ensuring that the prev and next pointers are properly updated. The printForward function prints theelements of the “Doubly linked list” in the forward direction by following the next pointers, and the “printBackward” 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 backward and forward directions to demonstrate the traversal capabilities of the “Doubly linked list” .

6.4 Overview

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 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.
- The printLinkedList function is provided to print the components 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 eliminated a node from a linked list, you need to consider three cases: deletion of the last node, elimination of a node in the middle of the list, and elimination of the head 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 next node and one pointing to the previous node.
- The printForward function prints the components of the doubly linked list in the forward direction by following the next pointers, and the printBackward function prints the elements in the backward direction by following the prev pointers.

6.5 Key terms

- **Traversal:** The process of visiting each node in a "linked list" sequentially, start 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 add a new node to a linked list at a particular position. It can involve inserting a node at the head of the list, at the end of list, 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 from the middle

list, and deleting the last node of list . It involves updating the links between nodes to maintain the integrity and connectivity of the remaining list.

- **“ Doubly Linked List “:** A type of linked list where every node contains two pointers: a) pointing to the previous node . b) one pointing to the next node. This permit traversal in both forward and backward directions. It provides additional flexibility compared to a “Singly linked list” but requires more memory to store the extra pointers.

6.6 Self- Evaluation Questions

- How does traversal of a linked list work and what is its time complexity?
- Define the process of inserting a node into a linked list and how it affects the rest of the list.
- What are the common techniques for removing a node from a linked list, and how do they differ in terms of “time complexity “?
- Define a header linked list, and what advantages does it offer over a regular linked list implementation?
- Define 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 favorite 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:

- What were the limitations of using arrays for playlist management in the Let'sPlay app?
- How did the implementation of a two-way linked list address the issues faced by the Let'sPlay app?
- 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

SORTING ALGORITHMS

Learning Objectives:

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

Structure:

1. Tree Data Structure
2. Implementation of Tree
3. Binary Tree
4. Overview
5. Key terms
6. Self- Evaluation Questions
7. Case Study
8. References

7.1 Tree Data Structure

It is a common hierarchical data structure in 2datastructuresand algorithms. It is a group of nodes joined by edges, with any node having one or more child nodes. The highest node is referred to as the root node, and each child node has the ability to create subtrees by bearing children of its own.

Trees are an important data structure in computer science and have various applications, such as represent hierarchical relationships, organising data effective 中 sorting and searching, as well as the use of different algorithms including balanced search trees and binarysearch.

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 tree node .
- Parent: A node with offspring nodes
- Child: A node linked to its parent node.
- Siblings: Nodes that have the same parent.
- Leaf: A node without children.
- Edge: A link between nodes.
- Depth: The distance along the path that leads from the root to a specific node
- Height: The deepest of any node in the tree.
- Subtree: A tree rooted at a child node.

7.1.2 Types of Trees:

- Binary Tree: In this tree where every node can have maximum two children.
- Binary Search Tree (BST): A tree where the left child of a node have a value smaller than the node, and the right child have a value larger than the node.
- AVL Tree: A self-balancing binary search tree where the heights of the left and right subtrees of any node differ by at most one.
- Red-Black Tree: A self-balanced binary search tree that assured logarithmic time complexity of searching , inserting , and deleting operations.

- 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 tree-like data structure used to store a dynamic set or an associative array in which strings are keys.

7.1.3 Tree Traversal

- Depth-First Traversal: Visit the nodes in a tree depth-wise before visiting siblings. Pre-order, in-order, and post-order traversals are the three typical forms of depth-first traversals
- 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: Examine each node in turn, working your way from left to right.
- Trees are fundamental for solving complex problems efficiently, and understanding tree data structures and algorithms is essential for many applications in computer science and software development.

7.2 Implementation of Tree

Using pointers, the nodes can be dynamically constructed to create the tree data structure.

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
  int data;
  struct Node* left;
  struct Node* right;
};
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;
}
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);
    }
  }
}
```

```
void inOrderTraversal(struct Node* root) {
  if (root != NULL) {
    inOrderTraversal(root->left);
    printf("%d ", root->data);
    inOrderTraversal(root->right);
  }
}int main() {
  struct Node* root = NULL;

  insertNode(&root, 4);
  insertNode(&root, 2);
  insertNode(&root, 1);
  insertNode(&root, 3);
  insertNode(&root, 6);
  insertNode(&root, 5);
  insertNode(&root, 7);
 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 shows a node in the binary tree. The Node method is used to built a new node with the specified data value. The insertNode function inserts a new node into the binary tree via the insertNode function 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 in Order Traversal method do an in-order traversal of the binary tree, printing the information 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

A binary tree is a kind of tree data structure where each node can have at maximum two children, referred to as the left, thild and the right child. The binary tree can be empty (null) or can consist of a rodunode with left and/or right subtrees. The left subtree have nodes that have values less than the root node, while the right subtree have nodes with values greater

than the root node. This property makes binary trees useful for efficient searching and sorting operations.

7.3.1 Binary Tree Operations:

- Insertion: 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 methods to traverse a binary tree to traverse all its nodes. Common traversal methods have 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 differentiate the aim 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 right and left subtrees are approximately equal, improving theefficiency of search and other operations. Examples of balanced binary trees are AVL trees and Red-Black trees.

```
struct Node* findMinNode(struct Node* root) {
  while (root->left != NULL) {
    root = root->left;
  }
  return root;
}
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 {
```

```
        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;
        }
    struct Node* temp = findMinNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
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);
```

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

      root = deleteNode(root, 4);
      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 shown data value. There are three examples to consider when deleting a node:

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

Case 2: Two children: We locate the smallest value node in its right subtree (which will be the leftmost node in the right subtree) if the node to be eliminated has two children. The data of the least value node is substituted for the node's data, and the minimum value node is then recursively removed from the appropriate 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

In data structures and algorithms, a tree is a mostly used hierarchical data structure. It is a group of nodes connected by edges, where every node can have zero or more child nodes. The peakset node in a tree is called the root node, and each child node can have its own children, forming subtrees.

Trees are an important 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.

Children are nodes connected to their parent .
A Binary Tree is a tree in which each node can have at most two children.
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 precise 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 insertNode function.
A binary tree is a type of tree data structure in which each node can have at most two children, referred to as the left child and the right child. The binary tree can be empty (null) or can consist of a root node with left and/or right subtrees.

7.5 Keywords

- **Tree:** A hierarchical data structure collected of nodes linked by edges, where each node can have zero or more child nodes. 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:** In this type of tree data structure each node can have at most two children: a right child and a left child. It is a primary tree variant used in a variety of algorithms and data structures.
- **AVL Tree:** It maintains its balance during insertions and deletions, ensuring efficient search, insertion, and deletion operations.
- **Node:** A regular component of a tree that have 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

- What is the definition of a tree in the context of data structures?
- How would you define a binary tree? What are its distinguishing characteristics?
- What are some common terms used in binary trees and their definitions? (e.g., root, parent, leaf, subtree)
- Can you explain the difference between depth and height in a binary tree?
- 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 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:

- How does the use of binary search trees improve the efficiency of data retrieval operations in the database system of the e-commerce company?

- What problem did the company face with the growth of the database, and how does an unbalanced tree affect the efficiency of a BST?

- 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

SEARCHING ALGORITHMS

Learning Objectives:

- Understand about the types of trees
- Application and advantages of binary tree
- Understand the tree traversal techniques

Structure:

1. Binary Tree
2. Tree Traversal
3. Summary
4. Keywords
5. Self-Assessment Questions
6. Case Study
7. References

8.1 Binary Tree

Binary trees are those that have two children or fewer at any given node. There are numerous varieties of binary trees.

8.1.1 Types of Binary Tree

Full Binary Tree

If each node has 0 or 2 children, a binary tree is fully formed. Here are a few explain 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

It is a degenerate tree where the left or right nodes are more prevalent. Therefore, there exist two distinct binary trees: left-skewed binary trees.

8.1.2 Application of Binary Tree

Binary trees are those that have two children or fewer at any given node. There are numerous types of binary trees.

- **Search Algorithms**:

It make optimal use of the binary tree's structure to look for a particular member. This search's complexity can be represented as O(log n), where n is the tree's node count.

- **Sorting Algorithms**:

Heap sort and binary sort are two efficient sorting algorithms that can be created with binary trees.

- **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 quantity of data.

- **File Systems:**

It can be used to apply file systems where each node is a directory or file. This makes file system searching and navigation efficient.

- **Compression Algorithms:**

It can be used to perform the Huffman coding compression algorithm, which gives letters variable-length codes dependent on how frequently they appear in the input data.

- **Decision Trees:**

Decision trees are one application of machine learning method, used for differentiation and regression analysis. Binary trees can be utilised to create decision trees.

- **Gaming AI:**

Game AI can be implemented using binary trees, where every node shows 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**
- Given that each node can only contain two child nodes, binary trees work incredibly well for elemental searching. This allows for binary search approaches. This suggests that the complexity of searches can be completed in O(log n) time.
- **Ordered Traversal**

  Because of their structural nature, binary trees can be traversed in a particular order, including in-order, pre-order, and post-order. This available certain actions to be performed on the nodes, such as sorting and printing the nodes in a specific order.
- **Memory Efficient**

  Comparing binary trees to other tree designs, they are more memory-efficient since each node only requires two child references. This translates into the capacity to carry out efficient searches even while handling large amounts of data in memory.
- **Faster Insertion and Deletion**

  Binary trees provide for O(log n) time complexity for inserts and deletions. For applications like database systems that require dynamic data structures, they are therefore a great choice.
- **Easy to implement**

  These Binary trees are known for a choice of a variety of applications since they are simple to create and comprehend.
- **Useful for Sorting**

  These trees are used to build two important sorting algorithms: heap sort and binary search tree sort.

8.2 Tree Traversal

8.2.1 Introduction

Our goal is to process a binary tree by "visiting" every node and acting accordingly, like publishing the node's contents, each time. Any process that requires visiting each node in a specific order is called a traversal.

Regardless of the order in which they are visited, applications simply require that each node be visited exactly once. It is necessary to access nodes in a way that maintains some relationship with other applications.

Preorder Traversal

A binary tree traversal that goes through the root, left child, right child, and left child once again. The following is an illustration of the preorder traversal algorithm:

```
Preorder(root):

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

End loop
```

The preorder traversal's code implementation is shown below.

This code defines a Node struct, which is used to represent each node in the binary tree. With the provided data, a new node is created using the newNode function. The preorder traversal is carried out recursively via the preorderTraversal function. After printing the current node's data, it recursively calls itself for the left and right subtrees. This is how the output will look:

Preorder binary tree traversal: 1 2 4 5 3

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

We could like to visit every node after viewing its progeny (and their subtrees). For example, We would like to remove every child of a node before deleting it. But to do that, you have to eliminate the children's children first, and so on.

Binary Tree Postorder Traversal Algorithm:

This displayed 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. To represent a node in a binary tree, we first build a structure called Node. To build a new node with the supplied data, use the createNode function.

```
#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;
}
```

In order to complete the post order traversal recursively, the postorder Traversal function visits the current node first, the left subtree, and then the right subtree. To print the nodes in post order traversal, we build a binary tree in the main function and call the postorderTraversal method. The binary tree's post-order traversal will produce the following output: Post-order binary tree traversal: 4 5 2 3 1

Inorder Traversal

In this traversal go to the node and finally the left child. The binary search tree uses this traversal to print every node in ascending value order.

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)
```

The code that demonstrates how the algorithm is used is shown below. A node in the binary tree is represented by the TreeNode structure. To build a new node with a specified value, use the createNode function.

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

// Definition for a binary tree node
struct TreeNode {
   int val;
   struct TreeNode* left;
   struct TreeNode* right;
};
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 inorder traversal is carried out recursively via the inorderTraversal function. It go to the current node, traverses the left subtree first, and then moves on to the right subtree. The inorder traverse is carried out and a sample binary tree is generated in the main function.

8.3 Summary

- A basic data structure in computer science, binary trees are employed in many different domains. They are made up of nodes, each of which has two pointers to its left and right children in addition to a value.
- 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.
- An algorithm called a tree traversal is used to travel through and interact with every node in a binary tree. The three primary traversal types—pre-order, in-order, and post-order—each specifying a distinct sequence for viewing the nodes.
- The root node is visited first by pre-order traversal, which then moves on to its left and right subtrees. It is frequently used to create prefix notation, evaluate expressions, and copy trees.
- Sorting binary search trees and getting the elements in a sorted order are common uses for it.
- The left subtree is visited first by post-order traversal, followed by the right subtree and the root node. It can be used to determine a binary tree's height, evaluate postfix expressions, and remove a tree.

8.4 Keywords

- **Binary Tree:**a hierarchical data structure for effective data storage and retrieval that is made up of nodes with a maximum of two offspring (left and right) on each.
- **Tree Traversals:**algorithms that visit and process a tree's nodes in a predetermined order.
- **Pre-order Traversal:**an algorithm for traversing trees that go to the root node first, then the left and right subtrees.
- **In-order Traversal:**a technique for traversing trees that go to the root node, the left subtree, and the right subtree in that order.

- **Post-order Traversal:**an approach to tree traversal that visits the root node following the left and right subtrees.
- **Binary Search Tree**: a kind of binary tree where the nodes are arranged in a certain sequence to facilitate effective searching and sorting (e.g., values in the left subtree are more than the root, and values in the right subtree are less).

8.5 Self-Assessment Questions

- What is the primary distinction between a binary search tree and a binary tree?
- How does a pre-order traversal algorithm work in binary trees? Provide a step-by-step explanation.
- When compared to pre-order or post-order traversals, under what circumstances would you prefer an in-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 the benefits and drawbacks of employing binary trees as opposed to alternative data structures for data archiving and retrieval?

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.

Following a thorough investigation, we suggested utilizing a Binary Search Tree (BST) design to rebuild their database. Because of its advantageous O(log n) time complexity for search, insertion, and deletion operations, the binary tree structure was selected.

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:

- What were the primary issues the client was experiencing with their initial linear data arrangement?
- How did the implementation of Binary Search Tree architecture improve database management for the client?
- 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

DYNAMIC PROGRAMMING

Learning Objectives:

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

Structure:

1. Threaded Binary Tree
2. Binary Search Tree
3. Heap Data Structure
4. Heap Sort
5. General Trees
6. Summary
7. Keywords
8. Self Assessment Questions
9. Case Studies
10. References

9.1 Threaded Binary Trees

The binary trees known as threaded binary trees are those in which we should use the left child pointer, the right child pointer, or both of the leaf nodes to accomplish or optimise 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 good amount of memory will be used with time complexity very high.
- It is obviously 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.

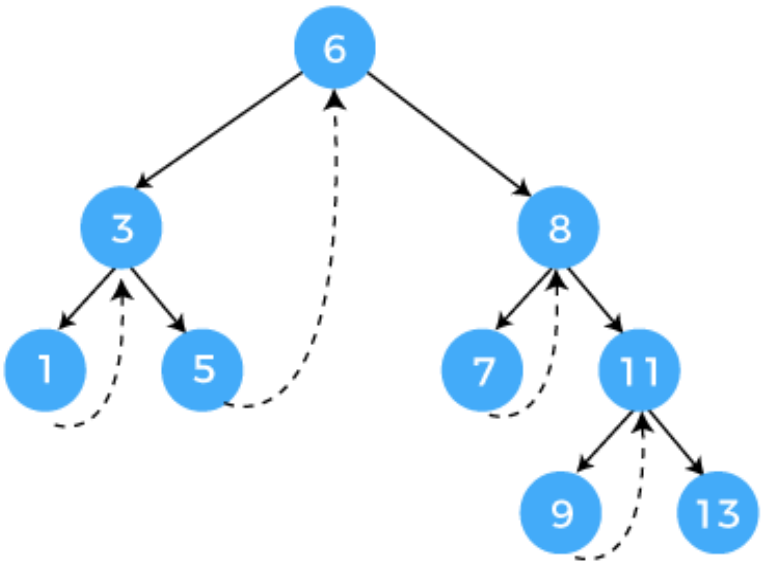
Recursive and iterative implementations of a tree traversal require stack space proportionate of the height of a tree (without employing the threaded binary tree notion). It takes up O(log n) space for a tree with n components if the tree is adequately balanced. In the worst scenario, when a tree assumes the shape of a chain, the tree's height is n, requiring O(n) space 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 thread pointers, it is impossible to return to the remainder of the tree.

9.1.2 Types of Threaded Binary Trees

Single Threaded Binary Trees

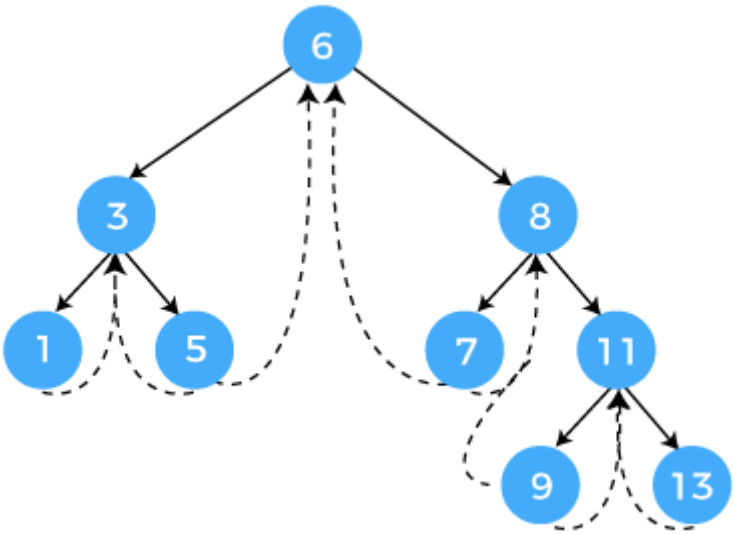
When adding information to a single threaded binary tree, generally we use either left or right child node pointers of the leaf nodes.The unique threaded linkages are depicted in the diagram below by dashed arrows. Each leaf node's right child pointer in this diagram is pointing to the leaf node's in-order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



A thread will get inserted in either the right or left link field of a node in one-way threaded binary trees. It will be pointing to the following node which 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 be pointing 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 rather left threaded binary trees are utilised least frequently. The right child's link field of the last node and the left child's link field of the first node in one-way threaded binary trees both contain the NULL. So to distinguish normal links from threads threads are represented by dotted lines.

Double threaded Binary Tree

In doubly threaded binary trees, any additional information is stored in both, the left child node link and the right child node link pointer of the leaf node. 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 in-order predecessor, while the right child pointer points to the leaf node's in-order successor.



In a two way threaded binary tree, a node's right side link field gets replaced by thread which links to the node's in-order successor.

When you have access to the root of binary tree, this is simple to navigate the entire tree; however, if you only have a pointer to one node, it may take some time or be impossible to identify the node that comes after it. By way of illustration, no other nodes may be accessed given simply a pointer to the leaf node because leaf nodes have no children (or successors). In order to make it easier to locate the "next" node (successor), the threaded tree includes additional information in some or all its nodes. Additionally, it may be traversed without using recursion and the additional storage (proportionate to the average tree's depth and proportionate to the total number of the nodes in the worst case when the given tree is skewed or linked) which it necessitates

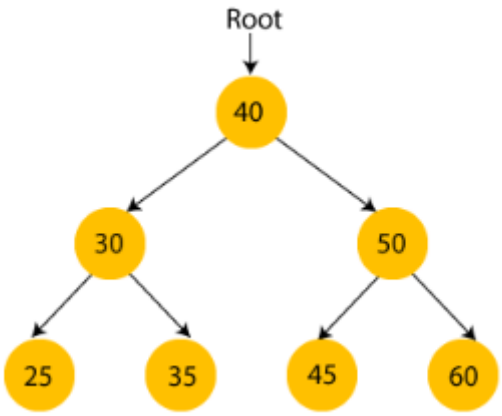
9.2 Binary Search Tree

This technique is 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 lesser than the key of the node is present in the left side sub-tree 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 the famous binary tree, where each node may have at most 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 manages the components in a particular order. The left side's node value in a binary search tree should be less than the parent's node value. Similarly the right node's value should be greater than the parent's node value. This rule applied recursively to the root's left as well as right sub-trees.



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

The left side child of the root node is bigger than its left side child and smaller than its right side child, as it is visible clearly. As a result, it is also meeting the binary search tree's property. So the result is that 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 an element in the Binary search tree.

Insertion operations and deletions in BST are quicker than those in array and linked lists.

9.2.3 Searching in BST

Key value comparison is a necessary logarithmic step in searching the BST structure. If the key value is equal to the root key, the search is successful. If the key value is lesser than root key value, the search is considered successful. Otherwise in case of key value larger than the root value, the search is considered to be successful.

Algorithm for the searching technique of BST is as follows

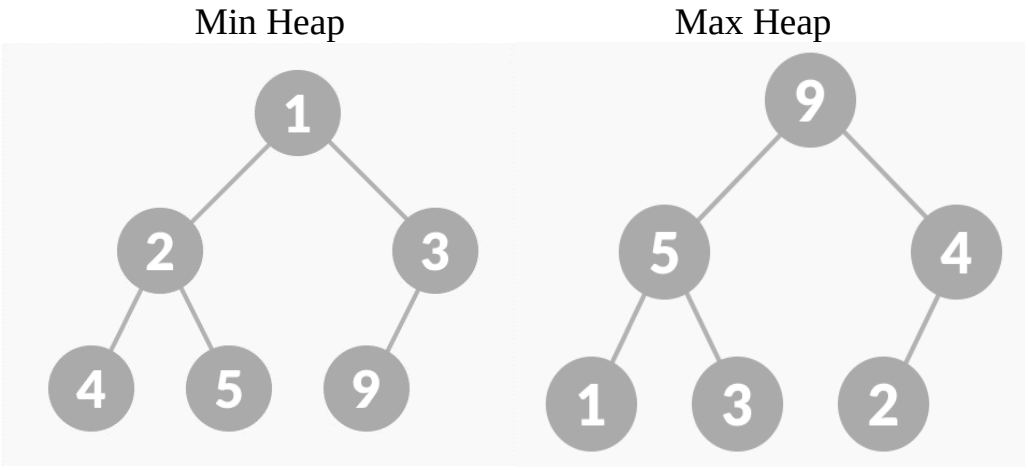
- Verify if the tree is pointing to NULL; if it is not pointing to NULL, proceed to the next step.
- Comparison between value of the search key and the value of the BST's root is made.
- Return back to the search screen to report "search successful" if the values of key matches the value of the root.
- Follow now the steps 3, 4, or 5 again to get the sub-tree.

9.2.4 Applications of the BST

- Indexing can be performed using BSTs.
- it can also be utilized to develop different types of searching methods.
- To implement several many other types data structures.
- Data storage and can be retrieved easily using BSTs in various types of decision support systems.
- In computer simulations, BSTs are used to store and retrieve data very quickly.
- Fast auto complete systems are developed using BSTs.
- Decision trees, used in machine learning based systems and artificial intelligence to arrive at decisions and forecast outcomes, can be implemented using BSTs.

9.3 Heap Data Structure

A heap is a unique tree-based data structure type. A heap is a fully complete binary tree. A heap is a type of structure where every node is often larger than any of its children and where the value of root node's key is the largest in all the nodes. This is called as the max heap. The key of the root node’s value is the smaller in all the nodes, and any given node can always be smaller than the child node or nodes. This is called the minimum heap property.



The min heap is also called binary heap.

9.3.1 Operations on Heap

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

Heapify

The process of converting a binary tree into a heap data structure is called Heapify. 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.

Find-max (or Find-min)

Also known as Peek/Top. The root node of the heap contains both the highest and the lowest element in the max-heap and min-heap, respectively.

Extract Min/Max

The maximum or minimum element in max-heap and min-heap, respectively, is returned and deleted by this operation. The root node contains the largest element.

9.4 Heap Sort

By using the components of the given array to create the min-heap or max-heap, heap sort processes the elements. The ordering of an array is known as a min-heap or max-heap, with the root member serving as the array's minimum or maximum element. It is an effective sorting algorithm. The idea behind a heap sort is to take each element out adding them to the sorted section. It is known to be the in-place sorting algorithm.

Basically, there are two stages to the sorting of components in a heap sort. They can be identified using the heap sort technique as follows:

- The first step involves rearranging the array's items to create a heap.
- After the heap has been created, the root element must now be repeatedly removed by shifting it to the end of the array, and the heap structure must then be stored with the other components.

The steps can be implemented into the following code. It sorts an array using the heap sort algorithm. The heapify function is used to build and maintain the heap, and the heapSort function orchestrates the overall sorting process.

The output will be:*Sorted array: 5 6 7 11 12 13*

```
#include <stdio.h>
void heapify(int arr[], int, int){ int largest
}
int left 2+1
int right=21+2:
if (left <n && arr[left]> arr[largest]) { largest left;
1
if (right <n && arr[right]> arr[largest]){
1
if
1
largest right;
(largest != 1) {
int temp = arr[0: arr[= arr[largest); arr[largest]= temp; heapify(arr, n, largest);
void heapSort(int arr[], int n) { // Build a max-heap
for (int i=n/2-1;>=0;-){ heapify(arr, n, i
1
// Extract elements one by one
for (int i = -1;10;-){
int temp = arr[0]: arr[0] = arr[i]: arr[]=temp; heapify(arr, 10:
1
1
int main(){
int arr (12, 11, 13, 5, 6, 7)
int n = sizeof(arr)/sizeof(arr[0]): heapSort(arr, n):
printf("Sorted array: "); for (int i = 0; i < n; i++){
1
printf("%d", arr[
1
printf("\n");
return 0;
```

9.5 General Tree

A General tree is a type of data structure made up of several nodes, each of which may have any number of child nodes. It is a hierarchical data structure, unlike a binary tree, that is not required to contain a certain number of child nodes.

Each node in a generic tree can have zero or more child nodes, and the nodes are joined by edges. However, unlike a binary tree where each node can only have a maximum of two child nodes, there is no constraint on the number of child nodes a node can have.

Linked lists and arrays are two examples of different data structures that can be used to describe general trees. Each node normally includes a value as well as several references or pointers to the nodes that make up its children.

Each tree node in this C implementation is represented by the structTreeNode, which includes two pointers: firstChild to the first child node and nextSibling to the next sibling node, as well as a value field to record the node value.

The addChild function adds a child node to an existing parent node, and the createNode function creates a new node with the supplied value.

The printTree function traverses the tree in preorder and outputs the node values. Finally, using the example you supplied, we create the nodes and construct the tree structure in the main function. The printTree function is then used to print the tree. The output of the code: Tree: A B D E C F

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

struct TreeNode {
   char value;
   struct TreeNode* firstChild;
   struct TreeNode* nextSibling;
};

struct TreeNode* createNode(char value) {
   struct TreeNode* newNode = (struct
TreeNode*)malloc(sizeof(struct TreeNode));
newNode->value = value;
newNode->firstChild = NULL;
newNode->nextSibling = NULL;
   return newNode;
}

void addChild(struct TreeNode* parent, struct TreeNode* child) {
   if (parent->firstChild == NULL) {
      parent->firstChild = child;
   } else {
      struct TreeNode* sibling = parent->firstChild;
      while (sibling->nextSibling != NULL) {
         sibling = sibling->nextSibling;
      }
      sibling->nextSibling = child;
   }
}

void printTree(struct TreeNode* node) {
   if (node == NULL) {
```

```
return;
    }

printf("%c ", node->value);

    struct TreeNode* child = node->firstChild;
    while (child != NULL) {
printTree(child);
        child = child->nextSibling;
    }
}
int main() {
    // Create nodes
    struct TreeNode* root = createNode('A');
    struct TreeNode* node_b = createNode('B');
    struct TreeNode* node_c = createNode('C');
    struct TreeNode* node_d = createNode('D');
    struct TreeNode* node_e = createNode('E');
    struct TreeNode* node_f = createNode('F');

    // Build tree structure
addChild(root, node_b);
addChild(root, node_c);
addChild(node_b, node_d);
addChild(node_b, node_e);
addChild(node_c, node_f);

    // Print the tree
printf("Tree: ");
printTree(root);
printf("\n");

    return 0;
}
```

9.6 Summary

- A threaded tree is a binary tree that has additional threads, or links, to efficiently traversing the tree without using any recursion or any stack. The threads connect certain nodes to their in-order predecessors or successors, allowing for quick navigation between nodes.
- A binary search tree is a special binary tree where the key value of a node is greater than all the key values in its subtree of left side and less than all the key values in sub-tree of right side. BSTs provide efficient deletion, searching, and insertion of

elements, with the average time complexity of O(log n) in the category of balanced trees.

- A heap is a fully complete binary tree here the key value of each node is either greater or equal to (max-heap) or less than or equal to (min-heap) the key values of its children. Heaps are commonly implemented using arrays and are used to efficiently find the maximum value or minimum value element (root) of the tree.
- Heap sort is categorised as omparison-based sorting algorithm that utilises the properties of a heap data structure.Heap sort is particularly useful when a sorted array is required, but the input size is too large to fit in memory.
- General trees are hierarchical data structures where each node can have an arbitrary number of child nodes.Unlike binary trees, there are no restrictions on the number of child nodes per parent node.

9.7 Keywords

- **Optimization:** Threaded trees optimize traversal operations, such as in-order traversal, by establishing additional threads or links between nodes. This eliminates the need for recursion or an explicit stack, leading to more efficient tree navigation.
- **Ordered:** A binary search tree is a hierarchical data structure where each node's value is ordered in relation to its child nodes. The left child holds a smaller value, while the right child holds a greater value. This property enables efficient searching, insertion, and deletion operations.
- **Priority:** A heap is a complete binary tree which can satisfy the heap property. It provides a priority-based data structure where the highest (or lowest) priority element is always at the root. Heaps are commonly used in priority queues and algorithms that require efficient access to the maximum (or minimum) element.
- **Sorting:** Heap sort is categorised as comparison-based sorting algorithm that utilizes the heap data structure. Heap Sort repeatedly extracts the maximum (or minimum) element from the heap, placing it in the sorted portion of the array. Heap sort has a time complexity of O(n log n) and is particularly useful for large data sets.
- **Hierarchy:** General trees are hierarchical structures where each node can have an arbitrary number of child nodes. They are used to represent relationships with varying degrees of branching, such as file systems, organizational hierarchies, or family trees. General trees provide a flexible and dynamic way to organize data.

9.8 Self-Assessment Questions

- How does a threaded tree optimize in-order traversal compared to a regular binary tree?
- How can the balance of a binary search tree be maintained to ensure optimal performance?
- Explain the process of heapifying an array and converting it into a heap.
- Describe the steps involved in the heap sort algorithm.
- 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:

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

9.10References

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.