College of Engineering & Applied Sciences

CSPB 2270

Data Structures

Class Notes

University Of Colorado

2024

Data Structures - Class Notes

1 C++ Review, Debugging, Unit Testing 4
C++ Review, Debugging, Unit Testing . . . 4
1.0.1 Activities . . . 4
1.0.2 Lectures . . . 4
1.0.3 Programming Assignment . . . 4
1.0.4 Chapter Summary . . . 5

2 Object Orientation in C++ & ADT 10
Object Orientation in C++ & ADT . . . 10
2.0.1 Activities . . . 10
2.0.2 Lectures . . . 10
2.0.3 Programming Assignment . . . 10
2.0.4 Chapter Summary . . . 11

3 Pointers & Lists 34
Pointers & Lists . . . 34
3.0.1 Activities . . . 34
3.0.2 Lectures . . . 34
3.0.3 Programming Assignment . . . 34
3.0.4 Chapter Summary . . . 35

4 Recursion & Trees 70
Recursion & Trees . . . 70
4.0.1 Activities . . . 70
4.0.2 Lectures . . . 70
4.0.3 Programming Assignment . . . 70
4.0.4 Chapter Summary . . . 71

5 Balanced Trees 96
Balanced Trees . . . 96
5.0.1 Activities . . . 96
5.0.2 Lectures . . . 96
5.0.3 Programming Assignment . . . 96
5.0.4 Chapter Summary . . . 97

6 Code Complexity, Binary Search Tree, Sorting Methods 101
Code Complexity, Binary Search Tree, Sorting Methods . . . 101
6.0.1 Activities . . . 101
6.0.2 Lectures . . . 101
6.0.3 Programming Assignment . . . 101
6.0.4 Chapter Summary . . . 102

7 B-Tree, Sets, Midterm 116
B-Tree, Sets, Midterm . . . 116
7.0.1 Activities . . . 116
7.0.2 Lectures . . . 116
7.0.3 Programming Assignment . . . 116
7.0.4 Exam . . . 116
7.0.5 Chapter Summary . . . 117

8 Priority Queue, Heap, Treap 130
Priority Queue, Heap, Treap . . . 130
8.0.1 Activities . . . 130
8.0.2 Lectures . . . 130
8.0.3 Programming Assignment . . . 130
8.0.4 Chapter Summary . . . 131

9 Hash Tables **136**
Hash Tables . . . 136
9.0.1 Activities . . . 136
9.0.2 Lectures . . . 136
9.0.3 Programming Assignment . . . 136
9.0.4 Chapter Summary . . . 137

10 Compression **148**
Compression . . . 148
10.0.1 Activities . . . 148
10.0.2 Lectures . . . 148
10.0.3 Programming Assignment . . . 148
10.0.4 Chapter Summary . . . 149

11 Graphs **151**
Graphs . . . 151
11.0.1 Activities . . . 151
11.0.2 Lectures . . . 151
11.0.3 Programming Assignment . . . 151
11.0.4 Exam . . . 151
11.0.5 Chapter Summary . . . 152

C++ Review, Debugging, Unit Testing

C++ Review, Debugging, Unit Testing

1.0.1 Activities

The following are the activities that are planned for Week 1 of this course.

- Take the C++ assessment
- Read the C++ refresher or access other resources to improve your skills (book activities are graded but the grades are not included in your final grade for this course)
- Read the zyBook chapter(s) assigned and complete the reading quiz(s) by next Monday
- Access the GitHub Classroom and get your Assignment-0 repository created, cloned, edited, and graded by next Tuesday
- Watch the videos for Cloning GitHub Classroom Assignments, Setting up an IDE in Jupytherhub, and Unit Testing

1.0.2 Lectures

Here are the lectures that can be found for this week:

- Course Concepts
- GitHub Classroom
- GitHub Security
- Accepting an Assignment
- Accessing Git Files
- Cloning Into JupyterHub
- VSCode in JupyterHub
- Multi File Programming
- Unit Testing Basics

1.0.3 Programming Assignment

The programming assignment for Week 1 is:

- **Programming Assignment 0 - Using GitHub and GitHub Classroom**

1.0.4 Chapter Summary

The first chapter of this week was **Chapter 1: Introduction to Data Structures**.

Section 1.1 - Data Structures

We define Data Structures to be the following:

- A data structure is a method of organizing, storing, and performing operations on data.
- Operations performed on data structures include accessing or updating stored data, searching for specific data, inserting new data, and removing data.
- Understanding data structures is crucial for effectively managing and manipulating data.

To summarize, data structures are methods of organizing, storing, and manipulating data, including arrays, linked lists, stacks, queues, trees, graphs, hash tables, and heaps.

Arrays

Arrays - Sequential collections of elements with efficient access and modification.

- Sequential collection of elements with unique indices. Indexes from zero.
- Efficient access and modification of elements at specific locations.
- Less efficient for inserting or removing elements in the middle.

Linked Lists

Linked Lists - Chain of nodes allowing efficient insertion and removal.

- Chain of nodes where each node contains data and a reference to the next node.
- Efficient insertion and removal of elements.
- Sequential traversal required for access specific elements.

Stacks

Stacks - Follows Last-In-First-Out (LIFO) principle for efficient insertion and removal from the top.

- Follows Last-In-First-Out (LIFO) principle.
- Insert and remove elements from the top of the stack.
- Useful for tasks like function class and undo operations.

Queues

Queues - Follows First-In-First-Out (FIFO) principle for efficient insertion, and removal from the front and rear.

- Follows First-In-First-Out (FIFO) principle.
- Insert elements at the rear and remove elements from the front.
- Useful for tasks like process scheduling.

Queues

Trees

Trees - Hierarchical structure for enabling efficient searching, insertion, and deletion.

- Hierarchical structure consisting of nodes connected by edges.
- Efficient searching, insertion, and deletion operations.
- Suitable for organizing file systems or representing hierarchical relationships.

Basic Data Structures

Graphs - Collection of nodes connected by edges, useful for representing complex relationships.

- Collection of nodes connected by edges.
- Each node can have multiple connections.
- Used to represent complex relationships like social networks or computer networks.

Basic Data Structures

Hash Tables - Data structure that uses hashing for efficient insertion, retrieval, and deletion of key-value pairs.

- Efficient data structure using hashing for fast key-value pair operations.
- Uses a hash function to convert keys into indices.
- Provides quick access to elements and handes collisions for proper storage.

Heaps

Heaps - Binary-tree based structure that ensures efficient retrieval of the minimum or maximum element.

- Binary tree-based structure for efficient retrieval of minimum or maximum element.
- Maintains a partial order property, such as the min-heap or max-heap property.
- Supports fast insertion and deletion of elements while preserving the heap property.

In the study of data structures, we explore various methods of organizing, storing, and manipulating data.

Arrays are sequential collections of elements, allowing efficient access and modification. Linked Lists from a chain of nodes, facilitating efficient insertion and removal. Stacks follow the Last-In-First-Out principle and are useful tasks like function calls and undo opertions. Queues follow the First-In-First-Out principle and are suitable for process scheduling.

Trees, consisting of nodes connected by edges, provide a hierarchical organization, enabling efficient searching, insertion, and deletion. Graphs are collections of nodes connected by edges and represent complex relationships like social networks or computer networks.Hash Tables emply hashing for efficient insertion, retrieval, and deletion of key-value pairs. They use a hash function to convert keys into indices, providing fast access to elements while handling collisions. Heaps, based on binary trees, allow efficient retrieval of the minimum or maximum element. They maintain a partial order property and support fast insertion and deletion while preserving the heap property.

Understanding these data structures and their characteristics is essential for problem-solving and designing efficient algorithms in data-oriented scenarios.

Section 1.2 - Abstract Data Types

Abstract Data Types (ADT) can be summarized as:

- Abstract Data Types (ADTs) define a set of operations and behavior for manipulating data without specifying the implementation details.
- ADTs provide a logical representation of data and operations, focusing on the "what" rather than the "how" of data manipulation.
- ADTs promote code abstraction and modularity, allowing for reusable and maintable code by encapsulating dtat and providing a clear interference for interaction.

To summarize, Abstract Data Types (ADTs) provide a high-level, logical representation of data and operations, focusing on the "what" rather than the "how", enabling code abstraction and modularity for reusable and maintable programming.

List

List - A basic data structure that represents an ordered collection of elements, allowing for efficient insertion, deletion, and retrieval operations.

- Lists are a versatile data structure that can store elements of any type and maintain their order, allowing for easy access and modification.
- They offere efficient insertion and deletion operations at both ends, making them suitable for scenarios where elements need to be dynamically added or removed.
- Lists can be implemented using various techniques such as arrays or linked lists, each with its own trade-offs in terms of memory usage and performance.

Dynamic Array

Dynamic Array - A dynamic array is a resizable data structure that provides the flexibility to dynamically adjust its size to accommodate the changing needs of a program.

- Dynamic arrays are resizable data structures that can grow or shrink in size based on the program's needs, allowing for efficient memory management.
- They provide the benefits of random access like traditional arrays, enabling constant-time access to elements using indices.
- Dynamic arrays allocate contiguous memory blocks, and when the array size exceeds its capacity, a larger memory block is allocated, and elements are copied over, ensuring efficient insertion and deletion operations while maintaining order.

Stack

Stack - A stack is a Last-In-First-Out (LIFO) data structure that allows efficient insertion and removal of elements from one end, commonly used in scenarios involving function calls, memory management, and undo operations.

- A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle, where the last element added is the first one to be removed.
- It supports two primary operations, push, which adds an element to the top of the stack, and pop, which removes the top most element from the stack.
- Stacks are commonly used in tasks that require tracking function calls, managing memory, and undo operations, providing efficient insertion and deletion of elements from a single end.

Queue

Queue - A queue is a Firs-In-First-Out (FIFO) data structure that enables efficient insertion at one end and removal at the other, commonly used for managing processes, task scheduling, and breadth-first search algorithms.

- Queues follow the First-In-First-Out (FIFO) principle, ensuring that the element inserted first is the first one to be removed.
- They support two primary operations: enqueue, which adds an element to the rear of the queue, and dequeue, which removes the element from the front.
- Queues are frequently utilized for process management, task scheduling, and breadth-first search algorithms, as they maintain the order of elements and provide efficient insertion and removal at both ends.

Deque

Deque - A deque (double-ended queue) is a data structure that allows efficient insertion and removal of elements at both ends, providing flexibility in managing data from the front or the rear.

- Deques support insertion and removal of elements at both ends, allowing for efficient operations at the front and rear of the data structure.

- They provide flexibility in managing data by enabling operations like push and pop at both ends, as well as accessing elements from either end.
- Deques are useful in scenarios where elements need to be added or removed from both ends, such as implementing algorithms like breadth-first search, implementing a queue with additional functionalities, or managing a sliding window in algorithms like dynamic programming.

Bag

Bag - A bad, also known as a multiset or a collection, is an unordered data structure that allows storing multiple occurrences of elements, providing efficient insertion and retrieval operations.

- Bags allow for the insertion of elements without enforcing any particular order, making them suitable for scenarios where maintaining the order is not necessary.
- Unlike other data structures, bags can store duplicate elements, allowing for multiple occurrences of the same item.
- Bags are commonly used when it is important to count or track the frequency of elements, such as in data analytics text processing, or certain types of machine learning algorithms.

Set

Set - A set is an unordered data structure that stores a collection of unique elements, providing efficient membership testing and set operations.

- Sets contain only unique elements, ensuring that duplicates are automatically removed, making the suitable for tasks that require uniqueness, such as maintaining a distinct list of items.
- Sets provide efficient membership testing, allowing for quick checks to determine if an element is present or absent.
- Sets support common set operations like union, intersection, and difference, enabling efficient manipulation and comparison of multiple sets, often used in tasks like data deduplication, finding common elements, or checking for similarities across multiple datasets.

Priority Queue

Priority Queue - A priority queue is an abstract data type that stores elements with associated priorities, allowing efficient retrieval of the highest priority element.

- Priority queues store elements with priorities, where the element with the highest priority can be efficiently retrieved.
- Elements in a priority queue are typically ordered on their priority, allowing for operations such as insertion and removal according to their priority level.
- Priority queues are commonly used in various applications like task scheduling, event-driven simulations, graph algorithms, and data compression, where efficient handling of elements based on their priority is essential for optimizing performance.

Dictionary (Map)

Dictionary (Map) - A dicitionary, also known as a map or associative array, is a data structure that stores key-value pairs, providing efficient lookup and retrieval of values based on their associated keys.

- Dicitionaries store key-value pairs, allowing efficient retrieval of values based on their associated keys.
- Keys in a dictionary are unique, enabling fast and direct access to the corresponding values.
- Dicitionaries are commonly used in situations that require fast lookup, such as data indexing, caching, symbol tables, and implementing algorithms like graph traversal or dynamic programming.

Lists provide ordered collections of elements with efficient insertion, deletion, and retrieval operations. They offer flexibility in managing data and are widely used in various applications that require maintaining a specific order. Dynamic arrays, on the other hand, offer resizable storage that adjusts to the needs of the program. They provide random access to elements and efficient memory management by reallocating memory blocks as the array

size changes.

Stacks adhere to the Last-In-First-Out (LIFO) principle and are commonly used for tracking function calls, managing memory, and implementing undo operations. They offer efficient insertion and removal of elements from one end. Queues, on the other hand, follow the Firs-In-First-Out (FIFO) principle. They are employed for process management, task scheduling, and breadth-first search algorithms. Queues enable efficient insertion at one end and removal at the other.

Bags are a type of data structure that stores unordered elements. They allow duplicates and enable frequency tracking. Bags are useful in scenarios where maintaining a distinct collection of items is not necessary, but counting occurrences or tracking frequency is essential. Sets, on the other hand, maintain unique elements. They provide efficient membership testing and support common set operations like union, intersection, and difference. Sets are utilized in various applications that require distinct elements and set manipulation.

Priority queues are data structures that store elements with associated priorities. They allow efficient retrieval of the highest priority element. Priority queues are commonly used in tasks like task scheduling, event-driven simulations, and graph algorithms. Finally, dictionaries (maps) store key-value pairs and provide efficient lookup and retrieval based on keys. They are extensively used for data indexing, symbol tables, and implementing algorithms that require fast access to values based on their associated keys.

Sec. 1.3 - Applications of ADTs

Abstract Data Types (ADTs) find applications across various domains, offering versatile solutions to address computational challenges. One common application of ADTs is in data storage and retrieval. ADTs like lists, arrays, and dictionaries (maps) provide flexible structures that enable efficient organization and access to data with different requirements for ordering, uniqueness, or key-value associations. These ADTs are used in databases, file systems, and data-driven applications to store and retrieve information in a structured and optimized manner.

ADTs play a crucial role in algorithm design. They are fundamental in solving computational problems efficiently. Stacks and queues, for example, are essential for managing program flow and data manipulation. They are used in areas such as compiler design, expression evaluation, and depth-first or breadth-first traversals. Priority queues are particularly useful in optimization algorithms and event-driven simulations, where elements with associated priorities need to be processed in a specific order.

Memory management in programming languages relies on ADTs for efficient memory allocation and deallocation. Dynamic arrays, for instance, are used to dynamically allocate and resize memory blocks as needed. Stacks are instrumental in tracking function calls and managing runtime memory, ensuring efficient resource utilization. ADTs help manage memory effectively, preventing issues like memory leaks or excessive memory fragmentation.

ADTs have applications in various fields, including simulation modeling, task scheduling, and graph manipulation. Simulation models often rely on ADTs for modeling and analyzing complex systems. Queues are used for process scheduling, while bags and sets assist in statistical analysis, data sampling, and randomness generation. In task scheduling, ADTs like queues help manage process execution and prioritize tasks based on their priority levels or time constraints. In graph manipulation and network analysis, ADTs such as dictionaries (maps) provide efficient storage and retrieval of graph elements and properties, while priority queues can aid in graph algorithms like Dijkstra's algorithm for finding the shortest path.

Object Orientation in C++ & ADT

Object Orientation in C++ & ADT

2.0.1 Activities

The following are the activities that are planned for Week 2 of this course.

- Read the zyBook chapter(s) assigned and complete the reading quiz(s) by next Tuesday (usually Monday but it's a holiday).
- Read the C++ refresher or access other resources to improve your skills.
- Watch the videos on C++ Classes and Abstract Data Types.
- Watch the videos on Object-Oriented Thinking and Debugging your Assignments.
- Implement the examples In week videos for yourself on Jupytherhub machine.
- Access the GitHub Classroom to get your Assignment-1 repository (assignment due next Tuesday).

2.0.2 Lectures

Here are the lectures that can be found for this week:

- C++ Classes Basics
  - Source Files
- Abstract Data Type (ADT)
- Notes for Assignment 1 - Vector10
- Objected Oriented Thinking
- Object Lifestyle
- My Code is Not Working

The lecture notes for this week are:

- Abstract & Concrete Data Types Lecture Notes

2.0.3 Programming Assignment

The programming assignment for Week 2 is:

- Programming Assignment 1 - Vector10

2.0.4 Chapter Summary

The first chapter of this week is **Chapter 2: Objects and Classes**.

Section 2.1 - Objects: Introduction

Objects

Objects are fundamental concepts in object-oriented programming (OOP) that represent real-world entities or abstract concepts. They encapsulate both data (attributes) and behavior (methods), allowing for modular and reusable code, enhanced code oganization, and modeling of complex systems. Objects promote the principles of encapsulation, inheritence, and polymorphism, facilitating efficient and modular software development.

Objects Example

Here is a simple example of objects in C++:

```cpp
class Person {
private:
    std::string name;
    int age;

public:
    Person(const std::string& name, int age) : name(name), age(age) {}

    void displayInfo() {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};

int main() {
    Person person("John", 25);
    person.displayInfo();
    return 0;
}

```

In this example, the Car class represents a car with a make, model, and year. An object of type Car named car is created in the main function, and its displayInfo method is called to print the car's make, model, and year.

Abstraction

Abstraction is a core concept in computer programming, helping to simplify complex systems by focusing on important aspects and hiding unnecessary details. It involves representing real-world objects or systems in a generalized way using classes and objects. Through abstraction, we create abstract classes that define a common interface and behavior for related objects while hiding implementation specifics. This allows us to manage system complexity, improve code organization, and promote reusability. Abstraction is crucial for creating modular, scalable, and maintainable software systems, allowing us to work at higher levels of abstraction without getting caught up in implementation intricacies.

Abstraction Example

An example of abstraction can be seen below:

```cpp
#include <iostream>

// Abstract class
class Shape {
public:
    virtual void draw() = 0;  // Pure virtual function

    void printName() {
        std::cout << "Shape" << std::endl;
    }
};

// Concrete class
class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a circle." << std::endl;
    }
};
```

```
20
21  // Concrete class
22  class Rectangle : public Shape {
23  public:
24      void draw() override {
25          std::cout << "Drawing a rectangle." << std::endl;
26      }
27  };
28
29  int main() {
30      // Creating objects of concrete classes
31      Circle circle;
32      Rectangle rectangle;
33
34      // Using the abstract class pointer to achieve abstraction
35      Shape* shapePtr = nullptr;
36
37      // Polymorphic behavior
38      shapePtr = &circle;
39      shapePtr->draw();  // Calls draw() of Circle class
40
41      shapePtr = &rectangle;
42      shapePtr->draw();  // Calls draw() of Rectangle class
43
44      shapePtr->printName();  // Calls printName() of Shape class
45
46      return 0;
47  }
48
```

This code demonstrates the concept of abstraction and polymorphism using an example of shapes. It defines an abstract class called "Shape" with a pure virtual function "draw()" and a non-virtual function "printName()". Two concrete classes, "Circle" and "Rectangle", inherit from the abstract class and provide their own implementations of the "draw()" function.

In the main function, objects of the concrete classes are created. An abstract class pointer, "shapePtr", is used to achieve abstraction. The pointer is assigned the address of the "Circle" object, and the "draw()" function is called, resulting in the message "Drawing a circle." Similarly, the pointer is assigned the address of the "Rectangle" object, and the "draw()" function is called, resulting in the message "Drawing a rectangle." This demonstrates polymorphic behavior, where the appropriate "draw()" function is called based on the object type.

Additionally, the "printName()" function of the abstract class is called using the abstract class pointer. This function is not overridden in the concrete classes, so the implementation in the abstract class is invoked, printing the message "Shape".

Overall, this code illustrates the use of abstract classes, pure virtual functions, inheritance, and polymorphism to achieve abstraction and enable the flexible handling of different objects through a common interface. It showcases the power of using abstract classes and polymorphism to create modular and extensible code for working with related objects in a data structures context.

Section 2.2 - Using a Class

Public Member Functions

Public member functions in object-oriented programming allow objects to interact with each other and provide functionality to the outside world. They define the behavior and operations that objects of a class can perform, encapsulating the logic and operations related to the class. Public member functions serve as an interface through which users can interact with objects, accessing and utilizing the functionality provided without exposing the internal implementation details. They promote code reusability, encapsulation, and maintainability, ensuring controlled access to object behavior and enabling modular design in object-oriented programming.

Public Member Functions Example

Below is an example of Public Member Functions in C++:

```
1  #include <iostream>
2
3  class Rectangle {
4  private:
```

```
    int width;
    int height;

public:
    void setDimensions(int w, int h) {
        width = w;
        height = h;
    }

    int calculateArea() {
        return width * height;
    }

    void printInfo() {
        std::cout << "Width: " << width << ", Height: " << height << std::endl;
    }
};

int main() {
    Rectangle rect;

    rect.setDimensions(5, 3);
    int area = rect.calculateArea();
    std::cout << "Area: " << area << std::endl;

    rect.printInfo();

    return 0;
}

```

This code example demonstrates the concept of public member functions in C++. It defines a Rectangle class with private member variables for width and height. The class provides three public member functions: setDimensions(), calculateArea(), and printInfo().

The setDimensions() function allows users to set the width and height of the rectangle by passing the values as parameters. The calculateArea() function performs the calculation of the rectangle's area by multiplying the width and height and returns the result. Finally, the printInfo() function prints the width and height of the rectangle to the console.

In the main() function, an object of the Rectangle class is created, and the public member functions are utilized to set the dimensions of the rectangle, calculate its area, and print the information. This example showcases how public member functions provide an interface for interacting with objects, allowing users to manipulate data, perform computations, and retrieve information in a controlled manner, promoting encapsulation and modular design in C++ programming.

Section 2.3 - Defining a Class

In object-oriented programming (OOP), private data members are a fundamental concept that allows for encapsulation and data hiding. Private data members are variables declared within a class that can only be accessed or modified by member functions within the same class. By designating data members as private, they are shielded from direct access by code outside the class, ensuring that the internal state and implementation details of an object are protected. This encapsulation promotes data integrity, enhances code maintainability, and prevents external code from inadvertently modifying or corrupting the object's data. Private data members facilitate information hiding and abstraction, allowing objects to maintain their integrity while providing controlled access to their functionality through public member functions.

Private Data Members Example

Here is an example of private data members in C++:

```
#include <iostream>

class BankAccount {
private:
    std::string accountNumber;
    double balance;

public:
    void deposit(double amount) {
        balance += amount;
    }

    void withdraw(double amount) {
```

```
14          if (amount <= balance) {
15              balance -= amount;
16          } else {
17              std::cout << "Insufficient balance." << std::endl;
18          }
19      }
20
21      void displayBalance() {
22          std::cout << "Account balance: " << balance << std::endl;
23      }
24  };
25
26  int main() {
27      BankAccount myAccount;
28
29      myAccount.deposit(1000.0);
30      myAccount.displayBalance();
31
32      myAccount.withdraw(500.0);
33      myAccount.displayBalance();
34
35      myAccount.withdraw(800.0);
36      myAccount.displayBalance();
37
38      return 0;
39  }
40
```

In the main() function, an object of the BankAccount class is created. Public member functions are used to deposit an amount, display the balance, withdraw amounts, and display the updated balance.

By making the accountNumber and balance private, they cannot be directly accessed or modified from outside the class. This ensures the encapsulation and data hiding of sensitive information. Users can interact with the bank account object through the public member functions, maintaining data integrity and preventing unauthorized access to or modification of the private data members.

This example illustrates how private data members in C++ provide encapsulation and data hiding. By hiding the internal implementation details, the class enforces controlled access to the data and protects it from unauthorized manipulation. Private data members facilitate proper data management and security within an object, ensuring that only the intended interface, defined by public member functions, is used to interact with and modify the object's state.

Private Data Members

Section 2.4 - Inline Member Functions

Inline Member Functions

An inline member function in C++ is a function that is defined within a class declaration and is marked with the inline keyword. When a member function is declared as inline, it suggests to the compiler that the function should be expanded at the point of its call instead of being invoked through a function call. This expansion replaces the function call with the actual code of the function, eliminating the overhead of the function call itself. Inline member functions are typically used for small and frequently used functions to improve performance by reducing the function call overhead. They provide a mechanism for code optimization and are especially useful when the function body is simple, making it more efficient to replace the function call with the actual code.

Inline Member Function Example

Below is an example of inline member functions:

```
1   #include <iostream>
2
3   class Rectangle {
4   private:
5       int width;
6       int height;
7
8   public:
9       void setDimensions(int w, int h) {
10          width = w;
11          height = h;
```

```
    }

    // Inline member function
    inline int calculateArea() {
        return width * height;
    }
};

int main() {
    Rectangle rect;

    rect.setDimensions(5, 3);
    int area = rect.calculateArea();

    std::cout << "Area: " << area << std::endl;

    return 0;
}

```

In this example, we have a Rectangle class with private data members width and height. The class provides two member functions: setDimensions() and calculateArea(). The setDimensions() function sets the width and height of the rectangle, while the calculateArea() function calculates the area of the rectangle by multiplying its width and height.

The calculateArea() function is declared as an inline member function by using the inline keyword before the function declaration. This suggests to the compiler that the function should be expanded at the point of its call. In this case, when calculateArea() is called, the compiler replaces the function call with the actual code of the function, eliminating the overhead of the function call.

In the main() function, an object of the Rectangle class is created. The setDimensions() function is called to set the width and height of the rectangle. Then, the calculateArea() function is invoked to calculate the area of the rectangle, and the result is printed to the console.

This example demonstrates how inline member functions can be used to optimize code performance by reducing the overhead of function calls. By marking the calculateArea() function as inline, the compiler expands the function call at the point of invocation, avoiding the function call overhead and providing direct access to the function's code. Inline member functions are particularly useful for small and frequently used functions, where the expansion at the call site can lead to performance improvements.

Section 2.5 - Mutators, Accessors, & Private Helpers

Mutators & Accessors

Mutators and accessors are two types of member functions commonly used in object-oriented programming to manipulate and retrieve the values of private data members of a class. Mutators, also known as setter functions or modifiers, are used to modify the values of private data members by accepting parameters and updating the internal state of the object. They provide a controlled way to change the values of the object's attributes while enforcing any necessary validation or business rules. Accessors, also known as getter functions or inspectors, are used to retrieve the values of private data members without allowing direct access to them. They return the values of private data members, allowing users to access the object's attributes in a read-only manner. Mutators and accessors play a crucial role in encapsulation, providing an interface to manipulate and retrieve the object's state while maintaining data integrity, encapsulation, and abstraction. They allow for controlled interaction with the object's data and facilitate modular design and code maintainability by separating the implementation details from the external interface of the class.

Mutators & Accessors Example

Below is an example of mutators & accessors in C++:

```
#include <iostream>

class Circle {
private:
    double radius;

public:
    // Mutator
```

```
9       void setRadius(double r) {
10          if (r >= 0) {
11              radius = r;
12          }
13      }
14
15      // Accessor
16      double getRadius() const {
17          return radius;
18      }
19
20      double calculateArea() const {
21          return 3.14 * radius * radius;
22      }
23  };
24
25  int main() {
26      Circle myCircle;
27
28      myCircle.setRadius(5.0);
29      double radius = myCircle.getRadius();
30      double area = myCircle.calculateArea();
31
32      std::cout << "Radius: " << radius << std::endl;
33      std::cout << "Area: " << area << std::endl;
34
35      return 0;
36  }
37
```

In this example, we have a Circle class with a private data member radius. The class provides two member functions: setRadius() and getRadius().

The setRadius() function is a mutator that allows users to set the value of the radius data member. It accepts a parameter r and updates the radius only if the value is non-negative.

The getRadius() function is an accessor that returns the value of the radius data member. It allows users to retrieve the value of radius without directly accessing the private data member.

In the main() function, an object of the Circle class is created. The setRadius() mutator is called to set the radius of the circle to 5.0. The getRadius() accessor is then used to retrieve the value of the radius, and the calculateArea() function is invoked to calculate the area of the circle. Finally, the radius and area are printed to the console.

This example demonstrates how mutators and accessors provide a controlled interface for manipulating and retrieving the values of private data members. The mutator setRadius() allows users to set the radius of the circle, while the accessor getRadius() allows them to retrieve the radius. By encapsulating the private data member and providing these member functions, the class ensures data integrity and abstraction. Users can interact with the object through the mutators and accessors without direct access to the private data member, promoting encapsulation and modular design in C++ programming.

Sec. 2.6 - Initialization & Constructors

Data Member Initialization

Data member initialization in C++ allows you to assign initial values to the data members of a class when objects are created. It provides a convenient way to ensure that data members have valid initial values and avoids the need for separate initialization steps. Data member initialization can be done using two approaches: member initialization list and default member initializer. Member initialization list initializes data members directly in the constructor's initialization list, while default member initializer assigns values to data members directly in the class declaration. By initializing data members during object creation, you can ensure that the object starts in a consistent state and avoid potential bugs or undefined behavior caused by uninitialized data. Data member initialization enhances code readability, simplifies object construction, and promotes good programming practices in C++.

Data Member Initialization Example

Here is an example of data member initialization in C++.

```
1  #include <iostream>
```

```
2
3   class Rectangle {
4   private:
5       int width;
6       int height;
7
8   public:
9       // Constructor with member initialization list
10      Rectangle(int w, int h) : width(w), height(h) {
11          // Additional constructor code, if needed
12      }
13
14      // Default member initializer
15      int area = width * height;
16
17      void printArea() {
18          std::cout << "Area: " << area << std::endl;
19      }
20  };
21
22  int main() {
23      Rectangle rect(5, 3);
24      rect.printArea();
25
26      return 0;
27  }
28
```

In this example, we have a Rectangle class with private data members width and height. There are two ways to initialize these data members.

Firstly, in the constructor declaration, we use a member initialization list to initialize the width and height data members directly. The constructor takes two parameters w and h, and the member initialization list assigns these values to the corresponding data members.

Secondly, we can use default member initializer directly in the class declaration. In this case, we initialize the area data member using a default member initializer, which calculates the area as the product of width and height.

In the main() function, we create an object of the Rectangle class named rect with width 5 and height 3. The constructor initializes the width and height data members using the member initialization list. The printArea() function is called, which displays the calculated area of the rectangle.

This example demonstrates how data member initialization can be done using member initialization list in the constructor or default member initializer in the class declaration. It ensures that the data members have valid initial values when objects are created, simplifies object construction, and promotes code readability. Data member initialization is a useful feature in C++ that helps ensure the consistency and integrity of objects' initial states.

Constructors

Constructors in object-oriented programming (OOP) are special member functions that are responsible for initializing objects of a class. They are called automatically when an object is created and allow you to set the initial state of the object. Constructors have the same name as the class and can have parameters to receive values required for initialization. They can perform various tasks, such as allocating memory, initializing data members, setting default values, and executing other necessary initialization logic. Constructors play a crucial role in object creation and ensure that objects start in a valid and consistent state. They promote encapsulation, as they provide a controlled way to initialize objects and enforce any necessary validation or business rules during the creation process. Constructors contribute to code readability, reusability, and maintainability by encapsulating the object initialization logic within the class itself.

Constructors Example

Here is an example of constructors in C++:

```
1   #include <iostream>
2
3   class Rectangle {
4   private:
5       int width;
6       int height;
7
8   public:
9       // Default constructor
10      Rectangle() {
11          width = 0;
12          height = 0;
13      }
14
15      // Parameterized constructor
16      Rectangle(int w, int h) {
```

```
            width = w;
            height = h;
        }

        void printDimensions() {
            std::cout << "Width: " << width << ", Height: " << height << std::endl;
        }
    };

    int main() {
        // Creating objects using constructors
        Rectangle rect1; // Default constructor called
        Rectangle rect2(5, 3); // Parameterized constructor called

        // Printing dimensions
        rect1.printDimensions(); // Output: Width: 0, Height: 0
        rect2.printDimensions(); // Output: Width: 5, Height: 3

        return 0;
    }

```

In this example, we have a Rectangle class with private data members width and height. The class provides two constructors: a default constructor and a parameterized constructor.

The default constructor initializes the width and height to 0. It is called automatically when an object is created without any arguments, as in the case of rect1.

The parameterized constructor takes two arguments w and h and initializes the width and height using the provided values. It is called when an object is created with specific values, as in the case of rect2.

In the main() function, we create two objects of the Rectangle class, rect1 and rect2, using the constructors. We then call the printDimensions() function to display the dimensions of the rectangles.

This example demonstrates how constructors are used to initialize objects of a class. The default constructor allows objects to be created with default values, while the parameterized constructor allows objects to be created with custom values. Constructors enable proper initialization of objects, ensuring they start in a valid state. They provide flexibility and encapsulation in object creation, enhancing code readability and maintainability.

Section 2.7 - Classes and Vectors / Classes

Vectors

The 'std::vector' class in C++ is a dynamic array container that provides a flexible and convenient way to store and manipulate a sequence of elements. It allows for dynamic resizing, efficient element access, insertion, and deletion at both ends, and provides various member functions to perform common operations on the elements. Vectors are templated, which means they can store elements of any type, providing great flexibility. They offer automatic memory management, handling memory allocation and deallocation internally. Vectors are widely used in C++ programming due to their versatility, efficiency, and ease of use, making them a fundamental data structure for managing collections of elements.

Vectors Example

Here is an example of the 'vectors' class in C++:

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

int main() {
    // Create a vector of integers
    std::vector<int> numbers;

    // Add elements to the vector
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);

    // Access elements using indexing
    std::cout << "First element: " << numbers[0] << std::endl;
    std::cout << "Second element: " << numbers[1] << std::endl;
    std::cout << "Third element: " << numbers[2] << std::endl;
```

```
17
18      // Iterate over the vector using a loop
19      std::cout << "All elements: ";
20      for (int i = 0; i < numbers.size(); i++) {
21          std::cout << numbers[i] << " ";
22      }
23      std::cout << std::endl;
24
25      // Remove the last element
26      numbers.pop_back();
27
28      // Check the size of the vector
29      std::cout << "Size of vector: " << numbers.size() << std::endl;
30
31      return 0;
32  }
33
```

In this example, we include the necessary header files for using std::vector. We create a vector named numbers that stores integers.

We use the push_back() function to add elements to the vector. In this case, we add the integers 10, 20, and 30 to the vector. We access elements of the vector using indexing, such as numbers[0] to access the first element. We iterate over the vector using a loop and print all the elements. We use the pop_back() function to remove the last element from the vector. Finally, we check the size of the vector using the size() function.

This example demonstrates the basic usage of std::vector in C++. It shows how to create a vector, add elements, access elements using indexing, iterate over the vector, remove elements, and check the size. The std::vector class provides a convenient and flexible way to work with dynamic arrays, making it a powerful data structure in C++ for managing collections of elements.

Section 2.8 - Separate Files for Classes

Two Files Per Class

Separate files for classes in C++ programs provide a modular approach to organizing code. Each class is defined in its own header file, containing the class declaration, and the member function implementations are placed in a corresponding source file. This practice enhances code organization, readability, and reusability. It simplifies navigation, allowing developers to quickly locate and modify code related to specific classes. Separating classes into individual files also promotes code reuse by facilitating their inclusion in other projects. Additionally, it aids in managing dependencies, prevents name conflicts, and simplifies maintenance and debugging. Overall, utilizing separate files for classes in C++ programs improves code structure and facilitates the development and management of complex projects.

Section 2.9 - Choosing Classes to Create

Decomposing Into Classes

When creating classes in Object-Oriented Programming (OOP), it is important to follow certain guidelines to ensure a well-designed and effective class structure. Start by identifying the attributes and behaviors that define the class's purpose and responsibilities. Encapsulate the data by declaring private data members and provide public access through member functions. Design intuitive and descriptive names for the class and its members. Establish clear and meaningful relationships between classes, using inheritance and composition when appropriate. Implement appropriate constructors, destructors, and assignment operators to manage the lifecycle of objects. Strive for cohesive and focused classes with single responsibilities. Apply principles like encapsulation, abstraction, inheritance, and polymorphism to achieve modularity, code reusability, and maintainability. Document the class

with clear comments and adhere to coding style conventions for consistency. Regularly review and refine the class design as needed to ensure a well-structured and efficient implementation.

Section 2.10 - Unit Testing (Classes)

Testbenches

In Object-Oriented Programming (OOP), a test bench refers to a dedicated component or code module designed to test and validate the functionality of other classes or modules in a system. It serves as an environment for conducting systematic and comprehensive testing of software components. A test bench provides a controlled setting to simulate different scenarios and input conditions, allowing developers to verify the correctness and robustness of their code. It typically includes test cases, input data, and expected output values, along with mechanisms to execute the tests and compare the actual results against the expected ones. By using test benches, developers can identify and rectify issues early in the development process, ensuring the quality and reliability of the software. Test benches play a crucial role in achieving effective testing and debugging practices, enabling thorough assessment and validation of object-oriented systems.

Regression Testing

In Object-Oriented Programming (OOP), regression testing refers to the process of retesting previously tested code to ensure that any modifications or enhancements to the system do not introduce new defects or regressions. It involves rerunning existing test cases on the modified code to verify that the changes made to the system have not adversely affected its existing functionality. Regression testing is crucial for maintaining the stability and reliability of software systems, especially in complex object-oriented projects where changes in one module can have unintended consequences on other interconnected modules. By performing regression testing, developers can identify and fix any regressions or unintended side effects caused by code modifications, ensuring that the system continues to function as expected and previous functionalities are not compromised. Regression testing is an integral part of the software development lifecycle, providing confidence in the system's integrity and minimizing the risk of introducing new defects during the development and maintenance phases.

Erroneous Unit Tests

Erroneous unit tests refer to test cases or test code that are flawed or incorrect, resulting in inaccurate or misleading test results. These tests may have various issues, such as incorrect assumptions, flawed logic, inadequate coverage, or improper assertions. Erroneous unit tests can lead to false positives or false negatives, where passing tests falsely indicate correct functionality or failing tests erroneously indicate defects. Such tests can be problematic as they can give a false sense of security or create confusion during the development process. It is important to identify and rectify erroneous unit tests promptly to ensure the reliability and effectiveness of the testing process. Conducting regular code reviews, employing static analysis tools, and encouraging collaboration and knowledge sharing within the development team can help in identifying and addressing erroneous unit tests, resulting in more accurate and reliable testing outcomes.

Unit Test Example

Here is an example of unit testing classes in C++:

```cpp
// File: MyClass.h
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
private:
    int value;

public:
    MyClass(int val);

    int getValue() const;
    void setValue(int val);
};

#endif

// File: MyClass.cpp
```

```cpp
#include "MyClass.h"

MyClass::MyClass(int val) : value(val) {}

int MyClass::getValue() const {
    return value;
}

void MyClass::setValue(int val) {
    value = val;
}

// File: MyClassTest.cpp
#include <gtest/gtest.h>
#include "MyClass.h"

TEST(MyClassTest, ConstructorSetsInitialValue) {
    MyClass obj(42);
    EXPECT_EQ(obj.getValue(), 42);
}

TEST(MyClassTest, SettingNewValueUpdatesValue) {
    MyClass obj(0);
    obj.setValue(100);
    EXPECT_EQ(obj.getValue(), 100);
}

int main(int argc, char** argv) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}

```

In this example, we have a class called MyClass with a private member variable value and public member functions getValue() and setValue(). We write unit tests for this class using the Google Test framework.

In the MyClassTest.cpp file, we define two test cases using the TEST macro provided by Google Test. Each test case focuses on testing a specific aspect of the MyClass class. For example, one test case checks if the constructor sets the initial value correctly, and another test case verifies that setting a new value updates the value correctly.

The main function initializes the Google Test framework using testing::InitGoogleTest and runs all the defined tests using RUN_ALL_TESTS(). To compile and run the tests, you would need to include the Google Test framework and compile the test files along with it.

This example demonstrates how you can use unit testing to verify the behavior and correctness of your class implementation. Each test case focuses on a specific aspect of the class, ensuring that it behaves as expected in different scenarios. By running these tests, you can identify and address any issues or regressions in your class implementation, leading to more reliable and robust code.

Section 2.11 - Constructor Overloading

Basics

Constructor overloading in Object Oriented Programming (OOP) refers to the ability to define multiple constructors for a class, each with different set of parameters. This allows objects to be created with different initial states or configurations, providing flexibility and customization during object instantiation. By overloading constructors, developers can conveniently initialize objects with different combinations of values or provide default values for certain parameters. Constructor overloading enables the creation of objects that meet specific requirements or use cases, making the class more versatile and adaptable to different scenarios. It promotes code reuse and enhances the usability of the class by accommodating various ways of object initialization.

Constructor Overloading Example

Below is an example of constructor overloading in C++:

```cpp
#include <iostream>

class MyClass {
private:
    int value;
```

```
6
7   public:
8       // Default constructor
9       MyClass() {
10          value = 0;
11      }
12
13      // Constructor with one parameter
14      MyClass(int val) {
15          value = val;
16      }
17
18      // Constructor with two parameters
19      MyClass(int val1, int val2) {
20          value = val1 + val2;
21      }
22
23      int getValue() const {
24          return value;
25      }
26  };
27
28  int main() {
29      MyClass obj1;               // Calls the default constructor
30      MyClass obj2(42);           // Calls the constructor with one parameter
31      MyClass obj3(10, 20);       // Calls the constructor with two parameters
32
33      std::cout << obj1.getValue() << std::endl;    // Output: 0
34      std::cout << obj2.getValue() << std::endl;    // Output: 42
35      std::cout << obj3.getValue() << std::endl;    // Output: 30
36
37      return 0;
38  }
39
```

In the above example, the MyClass class demonstrates constructor overloading. It has three constructors: a default constructor, a constructor with one parameter, and a constructor with two parameters. Each constructor initializes the value member variable based on the provided arguments or default values.

In the main function, we create three objects of the MyClass class using different constructor calls. The first object obj1 is created using the default constructor, which sets the value to 0. The second object obj2 is created by invoking the constructor with one parameter, setting the value to 42. The third object obj3 is created using the constructor with two parameters, where the value is the sum of the two provided values (10 and 20).

By overloading the constructors, we can instantiate objects with different initial states or configurations depending on the parameters provided. This enhances the flexibility and usability of the class, allowing developers to create objects with specific values or default values conveniently. Constructor overloading promotes code reuse and simplifies the process of object creation, making the class more versatile and adaptable to different use cases.

Section 2.12 - Constructor Initializer List

Constructor initializer lists in Object-Oriented Programming (OOP) provide a way to initialize class member variables directly in the constructor declaration, rather than assigning values to them within the body of the constructor. By using the initializer list syntax, constructors can efficiently initialize member variables, especially for cases involving const variables or reference variables that need to be initialized upon object creation. Constructor initializer lists offer several benefits, including improved performance, the ability to initialize const and reference variables, and the initialization of base class subobjects. They enhance code readability and maintainability by clearly expressing the initialization process and ensuring that member variables are properly initialized before the constructor body executes. Overall, constructor initializer lists are a powerful feature in C++ that enable efficient and proper initialization of class member variables during object construction.

Constructor Initializer List Example

Below is an example of constructor initializer list in C++:

```
1   #include <iostream>
2
3   class MyClass {
4   private:
5       int value;
6       const int constantValue;
```

```
7       int& refValue;
8
9   public:
10      MyClass(int val, int& ref) : value(val), constantValue(42), refValue(ref) {
11          // Constructor body
12      }
13
14      int getValue() const {
15          return value;
16      }
17
18      int getConstantValue() const {
19          return constantValue;
20      }
21
22      int& getRefValue() const {
23          return refValue;
24      }
25  };
26
27  int main() {
28      int ref = 100;
29      MyClass obj(42, ref);
30
31      std::cout << obj.getValue() << std::endl;          // Output: 42
32      std::cout << obj.getConstantValue() << std::endl;  // Output: 42
33      std::cout << obj.getRefValue() << std::endl;       // Output: 100
34
35      return 0;
36  }
37
```

In the above example, the MyClass class demonstrates the use of constructor initializer lists. It has three member variables: value, constantValue, and refValue, representing an integer, a constant integer, and a reference to an integer, respectively.

In the MyClassTest.cpp file, we define two test cases using the TEST macro provided by Google Test. Each test case focuses on testing a specific aspect of the MyClass class. For example, one test case checks if the constructor sets the initial value correctly, and another test case verifies that setting a new value updates the value correctly.

By using the constructor initializer list, we can efficiently and directly initialize these member variables, including the initialization of const and reference variables, which cannot be assigned values inside the constructor body.

The main function creates an object of the MyClass class, passing in the values 42 and ref as arguments. We can then access the member variables using the appropriate getter functions to verify their values.

Constructor initializer lists enhance code readability by explicitly and efficiently initializing member variables during object construction. They ensure proper initialization of const and reference variables, making the code more robust and maintainable. By using initializer lists, we can initialize member variables directly, avoiding unnecessary assignment statements within the constructor body.

Section 2.13 - The ‘this’ Implicit Parameter

Implicit Parameter

In C++, the ‘this’ implicit parameter is a pointer that is automatically passed to member functions of a class. It refers to the object on which the member function is being called. The ‘this’ pointer allows access to the member variables and member functions of the object within its own scope, distinguishing them from local variables or function parameters with the same name. It is particularly useful in scenarios where there is a need to differentiate between the object’s member variables and function parameters that have the same names. The ‘this’ pointer enables efficient and unambiguous access to the object’s data and behavior, promoting encapsulation and facilitating object-oriented programming principles.

Using ‘this’ In Class Member Functions and Constructors

In C++, the ‘this’ pointer is used in class member functions and constructors to refer to the object on which the function is being invoked. Within member functions, ‘this’ allows direct access to the member variables and member functions of the current object, differentiating them from local variables or function parameters. It is

particularly useful when there is a need to disambiguate between class members and local variables with the same name. 'this' can also be used in constructors to initialize member variables, especially in cases where the parameter names clash with the member variable names. By using 'this' in member functions and constructors, developers can ensure accurate and unambiguous access to the object's data and behavior, promoting clarity, readability, and maintainability of the code.

'this' Implicit Parameter Example

Here is an example of the use of 'this' implicit parameter in C++:

```cpp
#include <iostream>

class MyClass {
private:
    int value;

public:
    MyClass(int value) {
        this->value = value;
    }

    void printValue() {
        std::cout << "Value: " << this->value << std::endl;
    }
};

int main() {
    MyClass obj(42);
    obj.printValue();  // Output: Value: 42

    return 0;
}

```

In the above example, we have a class called MyClass with a private member variable value and a constructor that takes an integer parameter. Inside the constructor, we use the 'this' pointer to differentiate between the parameter value and the member variable value. By using this->value, we explicitly refer to the member variable and assign the value of the parameter to it.

The printValue() member function of MyClass also uses the 'this' pointer. Within the function, we use this->value to access the member variable and print its value. In the main() function, we create an object of MyClass called obj and pass the value 42 to the constructor. We then call the printValue() member function on the obj object, which outputs the value of the value member variable.

By using the 'this' pointer, we can differentiate between local variables and member variables within the class scope, ensuring the correct variable is accessed or modified. It promotes clarity and avoids naming conflicts between function parameters and member variables.

Section 2.14 - Operator Overloading

Overview

Operator overloading in Object-Oriented Programming (OOP) allows the customization of the behavior of predefined operators for user-defined classes. It enables objects of a class to exhibit intuitive and meaningful behavior when used with operators such as +, -, *, /, ==, and so on. By overloading operators, developers can define how objects of a class interact with operators, making code more expressive and natural. Operator overloading enables the use of familiar syntax and semantics for user-defined types, enhancing code readability and maintainability. It allows objects to participate in operations that are consistent with their intended purpose, leading to more concise and intuitive code.

Overloading Same Operator

Overloading the same operator more than once in a single class in C++ allows different behaviors to be defined for the same operator depending on the types of the operands. This feature is known as operator overloading with different argument types. By providing multiple implementations of an operator, each with distinct parameter types, the class can handle different scenarios and provide appropriate behavior for each case. This enables flexibility in how the class interacts with the operator, accommodating various operand combinations and ensuring consistent and

meaningful operations. Overloading the same operator multiple times in a class allows for versatile and specialized behavior, enhancing the usability and adaptability of the class within different contexts.

Operator Overloading Example

Below is an example of operator overloading in C++:

```cpp
#include <iostream>

class Vector2D {
private:
    double x, y;

public:
    Vector2D(double x = 0.0, double y = 0.0) : x(x), y(y) {}

    Vector2D operator+(const Vector2D& other) const {
        return Vector2D(x + other.x, y + other.y);
    }

    Vector2D operator-(const Vector2D& other) const {
        return Vector2D(x - other.x, y - other.y);
    }

    Vector2D operator*(double scalar) const {
        return Vector2D(x * scalar, y * scalar);
    }
};

int main() {
    Vector2D v1(2.0, 3.0);
    Vector2D v2(1.0, 2.0);

    Vector2D sum = v1 + v2;            // Operator+ overload
    Vector2D difference = v1 - v2;     // Operator- overload
    Vector2D scaled = v1 * 2.5;        // Operator* overload

    std::cout << "Sum: (" << sum.x << ", " << sum.y << ")" << std::endl;
    std::cout << "Difference: (" << difference.x << ", " << difference.y << ")"
    << std::endl;
    std::cout << "Scaled: (" << scaled.x << ", " << scaled.y << ")"
    << std::endl;

    return 0;
}

```

In the above example, we have a class called Vector2D representing a 2D vector. The class overloads the +, -, and * operators to perform vector addition, subtraction, and scalar multiplication, respectively.

By providing multiple implementations of the same operator, each with different parameter types (Vector2D and double in this case), the class can handle different scenarios. The operator+ overload performs element-wise addition of the coordinates, the operator- overload performs element-wise subtraction, and the operator* overload performs scalar multiplication.

In the main() function, we create two Vector2D objects, v1 and v2. We then use the overloaded operators to perform vector addition, subtraction, and scalar multiplication. The results are stored in sum, difference, and scaled variables, respectively.

The program outputs the results, demonstrating how the overloaded operators provide intuitive and meaningful behavior for the Vector2D class. Overloading the same operator multiple times in the class allows for flexible and specialized operations, enhancing the usability and expressiveness of the class.

Section 2.15 - Overloading Comparison Operators

Overloading comparison operators in Object-Oriented Programming (OOP) allows custom behavior to be defined for comparing objects of user-defined classes. By overloading operators such as ==, !=, <, >, <=, and >=, developers can specify how objects should be compared based on their internal data or specific criteria. This enables objects of a class to be compared in a way that is meaningful and appropriate for the class's concept and purpose. Overloading comparison operators allows for more natural and intuitive code, as objects can be compared using familiar syntax and semantics. It enhances the readability and clarity of code by providing consistent and logical comparisons for user-defined types, making it easier to reason about the behavior of objects in comparison operations.

Comparison Operator Overloading Example

Below is an example of comparison operator overloading in C++:

```cpp
#include <iostream>

class Fraction {
private:
    int numerator;
    int denominator;

public:
    Fraction(int numerator = 0, int denominator = 1)
    : numerator(numerator), denominator(denominator) {}

    bool operator==(const Fraction& other) const {
        return (numerator == other.numerator)
        && (denominator == other.denominator);
    }

    bool operator!=(const Fraction& other) const {
        return !(*this == other);
    }

    bool operator<(const Fraction& other) const {
        return (numerator * other.denominator)
        < (other.numerator * denominator);
    }

    bool operator>(const Fraction& other) const {
        return (numerator * other.denominator)
        > (other.numerator * denominator);
    }
};

int main() {
    Fraction f1(3, 4);
    Fraction f2(2, 3);
    Fraction f3(3, 4);

    if (f1 == f2) {
        std::cout << "f1 and f2 are equal." << std::endl;
    } else {
        std::cout << "f1 and f2 are not equal." << std::endl;
    }

    if (f1 != f3) {
        std::cout << "f1 and f3 are not equal." << std::endl;
    } else {
        std::cout << "f1 and f3 are equal." << std::endl;
    }

    if (f2 < f1) {
        std::cout << "f2 is less than f1." << std::endl;
    } else {
        std::cout << "f2 is not less than f1." << std::endl;
    }

    if (f1 > f2) {
        std::cout << "f1 is greater than f2." << std::endl;
    } else {
        std::cout << "f1 is not greater than f2." << std::endl;
    }

    return 0;
}

```

In the above example, we have a Fraction class representing a fraction with a numerator and denominator. We overload the comparison operators ==, !=, <, and > to compare fractions.

The operator== compares two fractions for equality, checking if both the numerator and denominator are the same. The operator!= is implemented in terms of operator==, negating the result. The operator< compares fractions based on their relative values, using cross multiplication to compare the numerators and denominators. Similarly, the operator> is implemented based on operator<, but with the operands swapped.

In the main() function, we create three Fraction objects, f1, f2, and f3, and perform comparison operations using the overloaded operators. We check for equality, inequality, less than, and greater than relationships between fractions and print the corresponding messages.

The program outputs the results of the comparisons, demonstrating the custom behavior defined by overloading the comparison operators. By overloading these operators, we can compare fractions using intuitive syntax and obtain meaningful results based on their numerical values.

Section 2.16 - Vector ADT

The Vector Abstract Data Type (ADT) is a versatile and efficient dynamic array-like structure that allows for the flexible storage and manipulation of elements. It offers constant-time access by index, efficient appending and removal of elements, and automatic resizing when needed. Vectors are widely used in programming for their ability to adapt to changing collection sizes, making them suitable for a variety of applications. They provide a contiguous block of memory, allowing for efficient traversal and sequential access. With their ability to store elements of any type, vectors serve as a fundamental data structure in algorithms, data structures, and applications that require dynamic and efficient element storage. Understanding the capabilities and operations of the Vector ADT is crucial for effectively managing and manipulating collections of elements in programming tasks.

Vector ADT Example

Here is an example of a vector ADT in C++:

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

int main() {
    // Creating a vector to store integers
    std::vector<int> numbers;

    // Adding elements to the vector
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);

    // Accessing elements by index
    std::cout << "First element: " << numbers[0] << std::endl;
    std::cout << "Second element: " << numbers[1] << std::endl;
    std::cout << "Third element: " << numbers[2] << std::endl;

    // Iterating over the vector
    std::cout << "Elements in the vector: ";
    for (int i = 0; i < numbers.size(); i++) {
        std::cout << numbers[i] << " ";
    }
    std::cout << std::endl;

    // Removing an element from the vector
    numbers.pop_back();

    // Querying the size and capacity of the vector
    std::cout << "Size of the vector: " << numbers.size() << std::endl;
    std::cout << "Capacity of the vector: " << numbers.capacity() << std::endl;

    return 0;
}

```

In this example, we include the <vector> header to use the Vector ADT provided by the C++ Standard Library. We create a vector called numbers to store integers. We use the push_back() function to add elements to the vector, and the [] operator to access elements by index. We iterate over the vector using a loop and print the elements. Then, we remove an element using the pop_back() function. Finally, we query the size of the vector using the size() function and the capacity using the capacity() function.

When you run this program, it will output the elements of the vector, the size, and the capacity. This example demonstrates the basic usage of the Vector ADT in C++ for dynamic storage and manipulation of elements.

Section 2.17 - Namespaces

Namespaces in C++ provide a way to group related code elements and prevent naming conflicts. They act as a container for identifiers such as variables, functions, and classes, allowing them to be organized and accessed in a structured manner. By enclosing code within a namespace, we can avoid naming collisions between entities with the same name but defined in different contexts. Namespaces enhance code modularity, readability, and maintainability by providing a hierarchical structure to the codebase. They enable developers to create separate logical units and manage the scope of identifiers more effectively. With namespaces, it becomes easier to differentiate and reference code elements, making the codebase more manageable and reducing the risk of naming conflicts when integrating different libraries or modules.

Namespaces Example

Here is an example of namespaces in C++:

```cpp
#include <iostream>

// First namespace
namespace First {
    void greet() {
        std::cout << "Hello from First namespace!" << std::endl;
    }
}

// Second namespace
namespace Second {
    void greet() {
        std::cout << "Hello from Second namespace!" << std::endl;
    }
}

int main() {
    First::greet();   // Calling greet() from the First namespace
    Second::greet();  // Calling greet() from the Second namespace

    return 0;
}

```

In this example, we define two namespaces: First and Second. Each namespace has its own greet() function that outputs a greeting message. In the main() function, we explicitly specify the namespace when calling the greet() function to differentiate between the two implementations.

This example demonstrates how namespaces in C++ allow us to organize code elements into separate logical units. By enclosing code within namespaces, we can prevent naming conflicts and explicitly specify which version of a function or variable to use. Namespaces help improve code readability and maintainability, especially in larger projects where different libraries or modules may have overlapping identifiers.

Section 2.18 - Static Data Members & Functions

Static data members and functions in C++ are associated with the class itself rather than specific instances of the class. A static data member is shared among all objects of the class and has a single instance regardless of the number of objects created. Similarly, a static member function is not bound to any specific object and can be called directly using the class name. Static members are useful for storing and accessing shared data or performing operations that are independent of individual objects. They can be accessed without creating an instance of the class and are commonly used for maintaining counts, global variables, utility functions, or class-wide properties. Static members provide a way to encapsulate data or functionality that is not tied to a specific object but belongs to the class as a whole.

Static Data Members & Functions Example

Below is an example of static data members & functions in C++:

```cpp
#include <iostream>

class MyClass {
public:
    static int count;  // Static data member

    static void incrementCount() {  // Static member function
        count++;
    }

    void displayCount() {
        std::cout << "Count: " << count << std::endl;
    }
};

int MyClass::count = 0;  // Initializing static data member

int main() {
    MyClass::incrementCount();  // Calling static member function
    MyClass obj1;
    obj1.displayCount();  // Output: Count: 1

    MyClass::incrementCount();
```

```
24        MyClass obj2;
25        obj2.displayCount();  // Output: Count: 2
26
27        MyClass::count = 10;  // Modifying static data member directly
28
29        MyClass obj3;
30        obj3.displayCount();  // Output: Count: 10
31
32        return 0;
33    }
34
```

In this example, we have a class called MyClass with a static data member count and a static member function incrementCount(). The count variable is shared among all objects of the class and is initialized to 0. The incrementCount() function increments the count by one. In the main() function, we call the static member function incrementCount() using the class name MyClass::incrementCount(). We also create multiple objects of MyClass and call the member function displayCount() to display the current value of count. We can directly access and modify the static data member count using the class name as shown. The output demonstrates how the static data member is shared among all objects and how the static member function can be used to manipulate it.

The second chapter of this week is **Chapter 3: Introduction to Algorithms**.

Section 3.1 - Introduction to Algorithms

Algorithms

In object-oriented programming (OOP), an algorithm refers to a set of step-by-step instructions or procedures designed to solve a specific problem or perform a particular task. It is a logical sequence of operations that can be implemented in code to achieve a desired outcome. In OOP, algorithms are often encapsulated within methods or functions of classes, enabling reusability and modularity. Algorithms in OOP can involve various operations such as data manipulation, conditional statements, loops, and function calls. They play a crucial role in implementing the logic and functionality of programs by providing a systematic approach to solving problems and achieving specific objectives. Well-designed algorithms are efficient, correct, and maintainable, contributing to the overall effectiveness and quality of the software.

Algorithm Efficiency

Algorithm efficiency in object-oriented programming (OOP) refers to the measure of how well an algorithm utilizes computational resources such as time and memory. It involves analyzing the performance characteristics of an algorithm and understanding its scalability as the input size increases. Efficiency is crucial in OOP as it directly impacts the program's overall performance and resource utilization. By designing and implementing efficient algorithms, developers can optimize the execution time and memory usage of their programs, leading to faster and more responsive software. Techniques like algorithmic complexity analysis, Big O notation, and data structure selection are employed to evaluate and improve algorithm efficiency. Striving for efficient algorithms is essential for developing high-performance applications that can handle large-scale data and complex computations effectively.

Big $\mathcal{O}$ Notation

Below are the different Big $\mathcal{O}$ Notations for algorithms with a simple explanation of each:

Big $\mathcal{O}$ Notation	Explanation
$\mathcal{O}(1)$	Constant time complexity - The algorithm's execution time is constant regardless of the input size.
$\mathcal{O}(\log(n))$	Logarithmic time complexity - The algorithm's execution time increases logarithmically with the input size.
$\mathcal{O}(n)$	Linear time complexity - The algorithm's execution time increases linearly with the input size.
$\mathcal{O}(n \log(n))$	Linearithmic time complexity - The algorithm's execution time grows in proportion to the product of the input size and its logarithm.
$\mathcal{O}(n^2)$	Quadratic time complexity - The algorithm's execution time increases quadratically with the input size.
$\mathcal{O}(2^n)$	Exponential time complexity - The algorithm's execution time grows exponentially with the input size.
$\mathcal{O}(n!)$	Factorial time complexity - The algorithm's execution time increases factorially with the input size.

These notations provide a way to express the scalability and efficiency of algorithms, allowing developers to compare and analyze different algorithms based on their time complexity and make informed decisions when designing and optimizing their programs.

To further demonstrate what an algorithm is, we take a look at a couple of examples.

Algorithms Example

Below are some examples of algorithms in C++:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> numbers = {4, 2, 7, 5, 1, 3, 6};

    // O(1) - Accessing an element in a vector using index
    int element = numbers[2];  // Accessing the third element

    // O(log n) - Binary search algorithm
    std::sort(numbers.begin(), numbers.end());
    bool found = std::binary_search(numbers.begin(), numbers.end(), 5);

    // O(n) - Linear search algorithm
    bool exists = std::find(numbers.begin(), numbers.end(), 8) != numbers.end();

    // O(n log n) - Sorting algorithm (e.g., Quick Sort)
    std::sort(numbers.begin(), numbers.end());

    // O(n^2) - Bubble sort algorithm
    for (int i = 0; i < numbers.size() - 1; i++) {
        for (int j = 0; j < numbers.size() - i - 1; j++) {
            if (numbers[j] > numbers[j + 1]) {
                std::swap(numbers[j], numbers[j + 1]);
            }
        }
    }

    // O(2^n) - Recursive Fibonacci sequence calculation
    int fibonacci(int n) {
        if (n <= 1)
            return n;
        return fibonacci(n - 1) + fibonacci(n - 2);
    }

    // O(n!) - Permutation generation using recursion
    void generatePermutations(std::vector<int>& arr, int start, int end) {
        if (start == end) {
            for (int num : arr) {
                std::cout << num << " ";
            }
            std::cout << std::endl;
        } else {
            for (int i = start; i <= end; i++) {
                std::swap(arr[start], arr[i]);
                generatePermutations(arr, start + 1, end);
                std::swap(arr[start], arr[i]);
            }
        }
    }

    // Example usage of the functions
    std::cout << "Element: " << element << std::endl;
    std::cout << "Binary search found: " << found << std::endl;
```

```
56        std::cout << "Linear search exists: " << exists << std::endl;
57
58        std::cout << "Sorted numbers: ";
59        for (int num : numbers) {
60            std::cout << num << " ";
61        }
62        std::cout << std::endl;
63
64        int fibResult = fibonacci(5);
65        std::cout << "Fibonacci(5): " << fibResult << std::endl;
66
67        std::vector<int> permutationArr = {1, 2, 3};
68        generatePermutations(permutationArr, 0, permutationArr.size() - 1);
69
70        return 0;
71    }
72
```

This code demonstrates the use of different algorithms corresponding to various Big $\mathcal{O}$ notations. It includes examples such as accessing an element in a vector with $\mathcal{O}(1)$, binary search with $\mathcal{O}(\log(n))$, linear search with $\mathcal{O}(n)$, sorting algorithms with $\mathcal{O}(n\log(n))$ and $\mathcal{O}(n^2)$, recursive Fibonacci sequence calculation with $\mathcal{O}(2^n)$, and permutation generation using recursion with $\mathcal{O}(n!)$. The output of the code showcases the results of each algorithm. This example provides a practical illustration of how different algorithms perform in terms of time complexity and highlights their corresponding efficiency characteristics.

Section 3.2 - Relation Between Data Structures and Algorithms

---

Algorithms for Data Structures

Algorithms for data structures refer to the set of procedures or methods designed to operate on specific data structures efficiently. These algorithms encompass a wide range of operations, including insertion, deletion, searching, sorting, and traversal, among others. The goal is to devise algorithms that leverage the underlying properties and organization of the data structure to optimize time and space complexity. For example, data structures like arrays, linked lists, stacks, queues, trees, and graphs each have their own set of algorithms tailored to their unique characteristics and usage scenarios. Efficient algorithms for data structures are essential for achieving optimal performance and scalability in various applications, enabling efficient data manipulation and retrieval operations. By employing appropriate algorithms for specific data structures, developers can harness the full potential of these structures and unlock efficient solutions for a wide range of computational problems.

Algorithms Using Data Structures

Algorithms using data structures refer to the utilization of specific data structures in combination with well-designed procedures to solve computational problems efficiently. These algorithms leverage the properties and functionality of data structures to store, organize, and manipulate data in a way that optimizes performance and resource utilization. By selecting the appropriate data structure for a given problem and implementing efficient algorithms, it is possible to achieve faster execution times, reduced memory consumption, and improved overall efficiency. Algorithms using data structures encompass a broad range of applications, including searching, sorting, graph traversal, pathfinding, data compression, and more. The synergy between algorithms and data structures is fundamental in computer science, enabling the development of powerful and efficient solutions to complex problems across various domains.

Data Structures Algorithm Example

Here is an example of an algorithm that is using a data structure in C++:

```
1   #include <iostream>
2   #include <vector>
3   #include <algorithm>
4
5   int main() {
6       std::vector<int> numbers = {5, 2, 7, 1, 3};
7
8       // Sorting the numbers using the std::sort algorithm
9       std::sort(numbers.begin(), numbers.end());
10
```

```
    // Searching for a specific number using the std::binary_search algorithm
    int target = 7;
    bool found = std::binary_search(numbers.begin(), numbers.end(), target);

    // Displaying the result
    if (found) {
        std::cout << "The number " << target
        << " is found in the vector." << std::endl;
    } else {
        std::cout << "The number " << target
        << " is not found in the vector." << std::endl;
    }

    return 0;
}

```

In this example, a std::vector is used as the data structure to store a collection of numbers. The std::sort algorithm is employed to sort the numbers in ascending order. Then, the std::binary_search algorithm is utilized to search for a specific number (target) within the sorted vector. The result is displayed based on whether the number is found or not. This example showcases the combination of algorithms (std::sort and std::binary_search) with the data structure (std::vector) to efficiently manipulate and search data, providing a concise and practical illustration of algorithms using data structures in C++.

Section 3.3 - Algorithm Efficiency

Algorithm Efficiency

Algorithm efficiency refers to the measure of how well an algorithm performs in terms of time and space usage. It is crucial to assess and analyze the efficiency of algorithms as it directly impacts the overall performance and scalability of a program. Efficiency is commonly evaluated by considering the time complexity, which measures how the algorithm's execution time grows with the input size, and the space complexity, which determines the amount of memory required by the algorithm. The goal is to design and select algorithms that exhibit favorable efficiency characteristics, such as lower time and space complexities, to ensure optimal performance and resource utilization. By employing efficient algorithms, developers can significantly improve program efficiency, reduce computational costs, and enable the handling of larger datasets and more complex problem instances. Evaluating and optimizing algorithm efficiency is a fundamental aspect of algorithm design and analysis, enabling the development of faster and more scalable solutions in various domains.

Runtime Complexity, Best Case, & Worst Case

Runtime complexity refers to the measure of how the performance of an algorithm scales with the size of the input. It provides insights into the efficiency of an algorithm in terms of time and space usage. The best case runtime complexity represents the lowest possible amount of time an algorithm can take to complete, usually occurring when the input is in the most favorable configuration. On the other hand, the worst case runtime complexity represents the maximum amount of time an algorithm can take to complete, typically occurring when the input is in the least favorable configuration. Analyzing the best and worst case scenarios helps in understanding the upper and lower bounds of an algorithm's performance. By considering both the best and worst case runtime complexities, developers can make informed decisions about the algorithm's efficiency and choose the most suitable algorithm for a given problem, balancing trade-offs between time and space requirements.

Space Complexity

Space complexity refers to the measure of the amount of memory or storage space required by an algorithm to solve a problem. It assesses how the space usage of an algorithm grows with the size of the input. The space complexity of an algorithm is influenced by factors such as the data structures used, the number of variables and their sizes, and any auxiliary space required during the execution. It is commonly expressed in terms of the maximum space used by the algorithm relative to the input size. Analyzing the space complexity helps in understanding the memory requirements of an algorithm and enables the estimation of how much space will be consumed during its execution. By considering the space complexity, developers can optimize memory utilization,

minimize unnecessary storage allocation, and ensure the algorithm can handle larger inputs without exhausting available memory resources.

Algorithm Efficiency Example

Below is an example of algorithm efficiency in C++:

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

// Function to find the maximum element in a vector
int findMax(const std::vector<int>& nums) {
    int max = nums[0];
    for (int i = 1; i < nums.size(); ++i) {
        if (nums[i] > max) {
            max = nums[i];
        }
    }
    return max;
}

int main() {
    std::vector<int> numbers = {5, 2, 8, 3, 1};

    // Find the maximum element in the vector
    int maxNum = findMax(numbers);
    std::cout << "Maximum number: " << maxNum << std::endl;

    return 0;
}

```

In this example, the findMax function takes a vector of integers as input and returns the maximum element in the vector. It uses a simple linear search algorithm to iterate through the vector and update the maximum element as it encounters larger values. The runtime complexity of this algorithm is $\mathcal{O}(n)$, where n is the size of the input vector. In the best case, when the maximum element is located at the beginning of the vector, the algorithm will terminate early, resulting in a lower execution time. In the worst case, when the maximum element is located at the end of the vector or when all elements are the same, the algorithm will perform the maximum number of comparisons, leading to a higher execution time. As for space complexity, this algorithm requires a constant amount of additional space to store the maximum element and loop variables, resulting in $\mathcal{O}(1)$ space complexity.

By analyzing the runtime complexity, best case, worst case, and space complexity of this example, we can understand the performance characteristics of the algorithm and make informed decisions about its efficiency and suitability for different input scenarios.

Pointers & Lists

Pointers & Lists

3.0.1 Activities

The following are the activities that are planned for Week 3 of this course.

- Reading Quiz(s) from last week are due on Monday.
- Assignment-1 (Vector10) is due Tuesday.
- Read the zyBook chapter(s) assigned and complete the reading quiz(s) by next Monday.
- Read the C++ refresher or access other resources to improve your skills.
- Watch videos on Pointers and Linked Lists.
- Implement the examples In this week videos on your Jupytherhub machine.
- Watch the video about Assignment-2 (Linked List).
- Access the GitHub Classroom to get your Assignment-2 repository (assignment due next Tuesday).

3.0.2 Lectures

Here are the lectures that can be found for this week:

- Pointers in C/C++
- Pass Objects by Value / Reference
- Stack vs. Heap
- Smart Pointers in C++
- shared_ptr Examples
- Linked List

3.0.3 Programming Assignment

The programming assignment for Week 3 is:

- Programming Assignment 2 - Linked List
- Linked List Interview Notes

3.0.4 Chapter Summary

The first chapter of this week is **Chapter 4: Pointers**.

Section 4.1 - Why Pointers?

Pointers are variables that store memory addresses as their values. They play a crucial role in programming languages, allowing direct manipulation and access to memory locations. Pointers enable efficient data manipulation by providing a way to refer to and modify data indirectly, rather than making unnecessary copies. They are especially useful in data structures and algorithms, as they facilitate dynamic memory allocation, efficient traversal of linked data structures, and enable the passing of values by reference. However, working with pointers requires careful management to avoid memory leaks and undefined behavior, making them a fundamental concept to understand in programming

Pointers Example

Below is an example of pointers in C++:

```cpp
#include <iostream>

int main() {
    int value = 42;
    int* pointer = &value;

    std::cout << "Value: " << value << std::endl;
    std::cout << "Memory address of value: " << &value << std::endl;
    std::cout << "Pointer value: " << pointer << std::endl;
    std::cout << "Dereferenced pointer value: " << *pointer << std::endl;

    *pointer = 99;

    std::cout << "Updated value: " << value << std::endl;

    return 0;
}

```

In this code snippet, we declare a variable value and initialize it with the value 42. We then declare a pointer variable pointer of type int* (pointer to an integer) and assign it the memory address of the value variable using the & (address-of) operator. By dereferencing the pointer with *pointer, we can access the value stored at that memory address, which is the value of value. We can modify the value of value indirectly by assigning a new value to *pointer. In this case, we update it to 99. Finally, we print the original and updated values to demonstrate how modifying the value through the pointer affects the original variable.

Section 4.2 - Pointer Basics

Pointer Variables

In object-oriented programming (OOP), pointer variables serve as essential tools for managing objects and dynamic memory allocation. Pointers allow for the creation and manipulation of objects indirectly by holding the memory address of the object instead of its actual value. This enables efficient memory usage and facilitates complex data structures and polymorphism. By using pointer variables, objects can be accessed and modified across different scopes and functions, providing flexibility and modularity in OOP. Additionally, pointers play a crucial role in managing resources and memory deallocation through techniques such as garbage collection or smart pointers. Understanding pointer variables in OOP is vital for effective memory management and advanced object manipulation.

Dereferencing a Pointer

Dereferencing pointers is the process of accessing the value stored at the memory address pointed to by a pointer variable. By using the dereference operator (*) in programming languages like C++ or C, we can retrieve and manipulate the actual value associated with the pointer. Dereferencing allows us to read or modify the data pointed

to by the pointer, enabling direct access and manipulation of objects, arrays, or structures in memory. Care must be taken when dereferencing pointers to ensure that they are pointing to valid memory locations, as accessing invalid or uninitialized memory can lead to unpredictable behavior or runtime errors. Understanding how to properly dereference pointers is crucial for efficient and correct utilization of pointer variables in programming.

Null Pointers

Null pointers are special pointers that do not point to a valid memory location. They are used to indicate that a pointer variable does not currently refer to any object or memory address. In programming languages like C++ or C, a null pointer is typically represented by the value 0 or nullptr. Null pointers are useful in several scenarios, such as initializing pointers before they are assigned valid addresses, checking for the absence of a valid object reference, or signaling the end of data structures like linked lists. However, accessing or dereferencing a null pointer can lead to runtime errors like segmentation faults, so it is important to check for nullness before using a pointer to avoid such issues. Understanding null pointers is crucial for handling pointer variables and ensuring proper memory safety in programming.

Pointer Basics Example

Below is an example of pointer basics in C++:

```cpp
#include <iostream>

int main() {
    int* ptr = nullptr;  // Initializing pointer to null

    if (ptr == nullptr) {
        std::cout << "Pointer is null!" << std::endl;
    } else {
        std::cout << "Pointer is not null!" << std::endl;
    }

    int value = 42;
    ptr = &value;  // Assigning valid memory address to the pointer

    if (ptr != nullptr) {
        std::cout << "Pointer is not null!" << std::endl;
        std::cout << "Dereferenced value: " << *ptr << std::endl;
    }

    return 0;
}

```

In this code, we start by initializing a pointer variable 'ptr' to 'nullptr', indicating that it does not currently point to a valid memory address. We then check if the pointer is null using the '==' comparison operator and print a corresponding message. Next, we declare an integer variable 'value' and assign it the value 42. We assign the memory address of 'value' to the pointer 'ptr' using the address-of operator '&'. After verifying that the pointer is not null, we dereference it using the '*' operator to access the value stored at the memory address pointed to by 'ptr'. Finally, we print the dereferenced value

In this example, we demonstrate the use of null pointers to indicate the absence of a valid memory address. We initialize the pointer to null, check its nullness, and then assign it a valid address. By dereferencing the pointer, we retrieve the value stored in the memory location pointed to by the pointer. The example highlights the importance of checking for nullness before accessing or dereferencing pointers to avoid runtime errors. Understanding and properly handling null pointers is crucial for ensuring memory safety and preventing unexpected behavior in C++ programs.

Section 4.3 - Operators, new, delete, and The Member Access

The 'new' Operator

In C++, the 'new' operator is used to dynamically allocate memory for objects or data structures at runtime. It returns a pointer to the allocated memory, allowing us to initialize and work with objects that reside in the heap rather than the stack. The 'new' operator is followed by the type of the object being allocated, and it automatically handles memory allocation and initialization. This allows for dynamic memory management and flexibility in creating objects whose size or lifetime may not be known at compile-time. It is important to pair the 'new' operator

with the corresponding 'delete' or 'delete[]' operator to deallocate the memory and avoid memory leaks. Proper understanding and usage of the 'new' operator are essential for managing dynamic memory allocation and object creation in C++ programs.

Member Access Operator

In programming languages like C++ and C#, the member access operator (->) is used to access members (variables or functions) of an object through a pointer to that object. It provides a convenient way to interact with objects when working with pointers, allowing access to the members of the object without dereferencing the pointer explicitly. By using the member access operator, we can access and modify the object's members, invoke member functions, or retrieve values stored in member variables. This operator simplifies the syntax and readability when working with objects through pointers, enabling seamless interaction with the underlying object's members and behavior. Understanding and correctly utilizing the member access operator is crucial when working with objects through pointers in object-oriented programming languages.

The 'delete' Operator

In C++ and similar languages, the 'delete' operator is used to deallocate memory that was previously allocated dynamically using the 'new' operator. It is used to free the memory occupied by objects or arrays created with 'new', ensuring efficient memory management. When 'delete' is applied to a single object, the memory occupied by that object is released. When 'delete[]' is used, it is used to deallocate memory allocated for arrays. By properly deallocating memory with the 'delete' operator, we prevent memory leaks and improve the overall performance of our programs. It is important to note that the 'delete' operator should only be used for memory that was dynamically allocated with 'new'. Using 'delete' on a non-dynamically allocated or previously freed memory can lead to undefined behavior and program crashes. Understanding how to correctly apply the 'delete' operator is crucial for effective memory management in C++ programs.

Pointer Operators Example

Below is an example of pointer operators in C++:

```cpp
#include <iostream>

int main() {
    int* ptr = new int;  // Dynamically allocate memory for an integer

    *ptr = 42;  // Assign a value to the dynamically allocated memory

    std::cout << "Dynamically allocated value: " << *ptr << std::endl;

    delete ptr;  // Deallocate the dynamically allocated memory

    return 0;
}

```

In this code, we use the 'new' operator to dynamically allocate memory for an integer. The 'new' operator allocates memory from the heap and returns a pointer to the allocated memory. We assign the address of this memory to the pointer variable 'ptr'. We then assign the value 42 to the memory location pointed to by 'ptr' using the dereference operator '*ptr'. Finally, we print the value stored in the dynamically allocated memory using 'std::cout'

To properly manage memory and prevent memory leaks, we use the 'delete' operator to deallocate the dynamically allocated memory when we no longer need it. In this example, we deallocate the memory pointed to by 'ptr' using 'delete ptr'. This frees up the memory for reuse by the system. It is essential to pair every 'new' operation with a corresponding 'delete' operation to avoid memory leaks and ensure efficient memory management

Section 4.4 - String Functions With Pointers

C String Library Functions

The C string library functions provide a set of built-in functions for working with null-terminated strings in the C programming language. These functions allow for efficient manipulation, searching, comparison, and copying of strings. Some commonly used C string library functions include 'strlen()' to calculate the length of a string, 'strcpy()' and 'strncpy()' to copy strings, 'strcmp()' and 'strncmp()' to compare strings, 'strcat()' and 'strncat()' to concatenate strings, and 'strstr()' to search for a substring within a string. These functions provide powerful tools for handling strings in C, making string manipulation and processing tasks more convenient and efficient. Understanding and utilizing the C string library functions are essential for effective string handling in C programs.

C String Search Functions

C string search functions are part of the C string library and provide efficient mechanisms for searching substrings within null-terminated strings. These functions offer ways to locate occurrences of specific characters or entire substring patterns within a larger string. Commonly used C string search functions include 'strchr()' to find the first occurrence of a character, 'strrchr()' to find the last occurrence of a character, 'strstr()' to locate the first occurrence of a substring, and 'strpbrk()' to search for any character from a set within a string. These functions simplify the process of searching and locating specific patterns within strings, enabling efficient text processing and manipulation in C programming. Understanding and utilizing C string search functions are essential for tasks such as parsing, pattern matching, and text analysis in C programs.

C String Functions Example

Below is an example of C string functions that involve pointers:

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

int main() {
    char str[] = "Hello, World!";
    char *ptr;

    ptr = strchr(str, 'W');
    if (ptr != NULL) {
        printf("Found: %s\n", ptr);
    } else {
        printf("Not found!\n");
    }

    ptr = strstr(str, "World");
    if (ptr != NULL) {
        printf("Found: %s\n", ptr);
    } else {
        printf("Not found!\n");
    }

    return 0;
}

```

In this code, we declare a null-terminated string 'str' containing the text "Hello, World!". We then use the 'strchr()' function to search for the first occurrence of the character 'W' within the string 'str'. If the character is found, 'strchr()' returns a pointer to the first occurrence of 'W', which we assign to the pointer variable 'ptr'. We then check if 'ptr' is not 'NULL' and print the result accordingly. Next, we use the 'strstr()' function to search for the first occurrence of the substring "World" within 'str'. If the substring is found, 'strstr()' returns a pointer to the start of the substring, which we assign to 'ptr'. Again, we check if 'ptr' is not 'NULL' and print the result

This example demonstrates how C string search functions can be used to locate specific characters or substrings within a larger string. By utilizing functions like 'strchr()' and 'strstr()', we can easily search for patterns and retrieve pointers to the locations of the found substrings within the original string. These C string search functions provide powerful tools for text processing, pattern matching, and string manipulation in C programming, making it easier to perform various tasks such as searching, parsing, and extracting information from strings

Section 4.5 - A First Linked List

Linked lists are a fundamental data structure used in computer science and programming to store and manage

collections of data. A linked list is composed of nodes, where each node contains a value and a reference to the next node in the list. Unlike arrays, linked lists do not require contiguous memory allocation, enabling dynamic memory allocation and efficient insertion and deletion operations. Linked lists offer flexibility in size and structure, allowing for efficient insertion and removal of elements at any position. However, accessing elements in a linked list requires traversing through the list sequentially, making it less efficient for random access compared to arrays. Overall, linked lists are valuable for scenarios that involve frequent insertions or removals, dynamic size requirements, or situations where efficient memory utilization is essential.

Linked List Example

Below is an example of linked lists in C++:

```cpp
#include <iostream>

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

class LinkedList {
private:
    Node* head;

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

    void insert(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        newNode->next = head;
        head = newNode;
    }

    void display() {
        Node* current = head;
        while (current != nullptr) {
            std::cout << current->data << " ";
            current = current->next;
        }
        std::cout << std::endl;
    }
};

int main() {
    LinkedList list;

    list.insert(5);
    list.insert(10);
    list.insert(15);
    list.insert(20);

    list.display();

    return 0;
}

```

In this code, we define a 'Node' struct that represents a single node in the linked list. Each node contains a data field to hold the value and a 'next' pointer to refer to the next node in the list. We then define a 'LinkedList' class that has a 'head' pointer as a private member, which points to the first node in the list

The 'LinkedList' class provides two member functions. The 'insert()' function inserts a new node at the beginning of the list. It creates a new node, assigns the given value to its 'data' field, and updates the 'next' pointer to point to the current head. The 'head' pointer is then updated to point to the newly inserted node. The 'display()' function traverses the linked list and prints the values of each node. It starts from the head and continues moving to the next node until reaching the end (i.e., when the 'next' pointer is 'nullptr'). In the 'main()' function, we create an instance of the 'LinkedList' class, insert several values into the list, and then call the 'display()' function to print the values

This example showcases a basic implementation of a linked list in C++. Linked lists are dynamic data structures that provide efficient insertion and deletion operations, making them suitable for scenarios where frequent modifications to the list are required. By utilizing pointers to link nodes, linked lists offer flexibility and efficient memory utilization compared to other data structures like arrays. Understanding and utilizing linked lists are essential for managing and manipulating collections of data in various programming scenarios

Section 4.6 - Memory Regions, Heap / Stack

In computer programming, memory regions are areas of memory that serve different purposes in managing data. The static memory region, also known as the global memory, stores static and global variables that are allocated at compile-time and have a fixed lifetime throughout the program execution. The stack memory region is used for storing local variables, function call frames, and other runtime data. It operates in a Last-In-First-Out (LIFO) manner, where memory is allocated and deallocated as functions are called and return. The heap memory region is a dynamically allocated memory area that is used for managing dynamic memory at runtime. It allows for dynamic allocation and deallocation of memory using mechanisms such as 'new' and 'delete' or 'malloc()' and 'free()'. The heap is more flexible than the stack and can be used to allocate memory for objects whose size or lifetime is not known at compile-time. Understanding the distinctions between static memory, stack memory, and heap memory is crucial for efficient memory management and proper allocation of resources in programming.

Memory Regions Example

Below is an example of memory regions in C++:

```cpp
#include <iostream>

// Static memory region - global variable
int globalVariable = 10;

void stackFunction() {
    // Stack memory region - local variable
    int stackVariable = 20;
    std::cout << "Stack variable: " << stackVariable << std::endl;
}

int main() {
    // Static memory region - global variable
    std::cout << "Global variable: " << globalVariable << std::endl;

    stackFunction();

    // Heap memory region - dynamically allocated memory
    int* heapVariable = new int(30);
    std::cout << "Heap variable: " << *heapVariable << std::endl;
    delete heapVariable;

    return 0;
}

```

In this code, we demonstrate the different memory regions: static memory, stack memory, and heap memory. The 'globalVariable' is allocated in the static memory region and has a global scope, accessible throughout the program. We print its value in the 'main()' function

The 'stackFunction()' represents a function, and any local variables declared inside it, such as 'stackVariable', are allocated in the stack memory region. These variables have a limited lifetime within the scope of the function. We print the value of 'stackVariable' inside the function. Next, we allocate memory dynamically on the heap memory region using the 'new' operator. We assign a value of 30 to 'heapVariable' and print its value. It is important to note that memory allocated on the heap must be deallocated using the 'delete' operator to prevent memory leaks. We deallocate the memory at the end of 'main()' using 'delete'

In summary, this example illustrates the distinctions between static memory, stack memory, and heap memory in C++. Static memory is used for global variables with a fixed lifetime, while stack memory is used for local variables within functions. Stack memory allocation and deallocation are handled automatically as functions are called and return. Heap memory allows for dynamic memory allocation and deallocation using 'new' and 'delete', providing flexibility for objects with unknown size or lifetime. Understanding these memory regions is crucial for efficient memory management and proper utilization of resources in C++ programs

Section 4.7 - Memory Leaks

Memory leaks occur when dynamically allocated memory is not properly deallocated or released after it is no longer needed. In programming, memory leaks can happen when the programmer forgets to free memory using the appropriate deallocation mechanisms, such as 'delete' in C++ or 'free()' in C. As a result, the allocated memory remains inaccessible, leading to a gradual accumulation of unused memory over time. Memory leaks can cause programs to consume excessive memory, leading to degraded performance, increased resource usage, and potentially

causing the program to crash or terminate unexpectedly. Detecting and fixing memory leaks is crucial for efficient memory management, and it involves identifying and releasing dynamically allocated memory when it is no longer required, thus preventing unnecessary memory consumption and maintaining the stability and performance of the program.

Memory Leak Example

Below is an example of a memory leak in C++:

```cpp
#include <iostream>

void memoryLeak() {
    int* ptr = new int(42);  // Dynamically allocate memory

    // The following line is missing the deallocation step
    // delete ptr;  // Uncommenting this line will fix the memory leak
}

int main() {
    memoryLeak();

    // More code...

    return 0;
}

```

In this code, we have a function 'memoryLeak()' that demonstrates a memory leak scenario. Inside this function, we dynamically allocate memory using the 'new' operator to create an integer with a value of 42 and assign it to the pointer variable 'ptr'. However, the crucial step of deallocating the dynamically allocated memory is missing. The 'delete' operator, which would free the memory, is commented out in the code

When 'memoryLeak()' is called, memory is allocated for the integer but never released, resulting in a memory leak. The memory leak occurs because the program loses track of the allocated memory, making it inaccessible for future use. In this example, the memory leak is intentional to demonstrate the concept, but in real-world scenarios, memory leaks are typically unintentional

Memory leaks can cause the program to consume more and more memory over time, potentially leading to performance issues, resource exhaustion, or program crashes. Detecting and fixing memory leaks involves being diligent in deallocating dynamically allocated memory using the appropriate deallocation mechanisms ('delete' in C++). By ensuring proper memory management, programs can avoid unnecessary memory consumption and maintain stability and efficiency

Section 4.8 - Destructor

Destructors are special member functions in object-oriented programming languages like C++ that are automatically called when an object is destroyed or goes out of scope. They have the same name as the class, preceded by a tilde (). Destructors are primarily used to clean up resources allocated by the object, such as releasing dynamically allocated memory or closing files or connections. They are particularly useful for ensuring proper resource management and preventing memory leaks or resource leaks. When an object is no longer needed, either because it goes out of scope or is explicitly deleted, the destructor is automatically invoked. The destructor allows the object to perform any necessary cleanup operations, freeing up resources and maintaining the integrity of the program. Understanding and implementing destructors appropriately is crucial for effective resource management and maintaining the overall robustness and efficiency of object-oriented programs.

Destructors Example

Below is an example of destructors in C++:

```cpp
#include <iostream>

class Resource {
private:
    int* data;

public:
    Resource() {
        data = new int[10];
        std::cout << "Resource acquired." << std::endl;
    }
```

```
12
13      ~Resource() {
14          delete[] data;
15          std::cout << "Resource released." << std::endl;
16      }
17
18      // Other member functions...
19  };
20
21  int main() {
22      Resource myResource;
23      // ...do some operations with myResource
24
25      return 0;
26  }
27
```

In this code, we have a class 'Resource' that manages a dynamically allocated array 'data'. The constructor of the class, 'Resource()', is responsible for acquiring the resource by allocating memory using the 'new' operator. In this case, we allocate an array of integers with a size of 10. Within the constructor, we display a message to indicate that the resource has been acquired

The crucial aspect is the destructor, ' Resource()'. It is automatically invoked when the object of the 'Resource' class goes out of scope, which happens when the 'main()' function ends. The destructor takes care of releasing the allocated memory using the 'delete[]' operator to free the array. We also display a message in the destructor to indicate that the resource has been released. In the 'main()' function, we create an object 'myResource' of the 'Resource' class. When 'main()' finishes executing, the 'myResource' object goes out of scope, causing the destructor to be automatically called. As a result, the allocated memory is properly deallocated, ensuring efficient resource management

The example illustrates the usage of a destructor in C++. Destructors are essential for cleaning up resources and performing necessary cleanup operations when an object is no longer needed. By implementing a destructor, developers can ensure proper resource management, prevent memory leaks, and maintain the overall robustness and efficiency of their programs

Section 4.9 - Copy Constructors

Copy constructors are special member functions in object-oriented programming languages like C++ that are used to create a new object as a copy of an existing object of the same class. The copy constructor is invoked when a new object is initialized from an existing object, either by direct initialization or by passing an object as a function argument by value. It performs a member-wise copy of the data from the source object to the newly created object. Copy constructors are particularly useful when working with dynamically allocated memory or complex objects that require deep copying. By defining a custom copy constructor, developers can ensure that the new object has its own copy of the data, preventing unintended side effects due to shallow copying. Understanding and implementing copy constructors properly is crucial for correct object initialization and avoiding unexpected object state modifications when working with objects in C++.

Copy Constructor Example

Below is an example of copy constructors in C++:

```
1       #include <iostream>
2
3   class Car {
4   private:
5       std::string brand;
6
7   public:
8       Car(const std::string& carBrand) : brand(carBrand) {}
9
10      // Copy constructor
11      Car(const Car& other) : brand(other.brand) {
12          std::cout << "Copy constructor called." << std::endl;
13      }
14
15      void displayBrand() {
16          std::cout << "Brand: " << brand << std::endl;
17      }
18  };
19
20  int main() {
21      Car car1("Toyota");
```

```
22      Car car2 = car1;  // Copy constructor is invoked here
23
24      car1.displayBrand();
25      car2.displayBrand();
26
27      return 0;
28  }
29
```

In this code, we have a 'Car' class that represents a car object with a 'brand' attribute. The constructor 'Car(const std::string& carBrand)' initializes the 'brand' attribute with the provided 'carBrand' value

The important aspect is the copy constructor 'Car(const Car& other)', which is invoked when a new 'Car' object is created as a copy of an existing 'Car' object. In the 'Car' class, the copy constructor performs a member-wise copy of the 'brand' attribute from the source object to the newly created object. In this example, we also include a message to indicate when the copy constructor is called. In the 'main()' function, we create an object 'car1' with the brand "Toyota". Then, we initialize 'car2' using the copy constructor by assigning 'car1' to 'car2'. This invokes the copy constructor, creating a new 'Car' object 'car2' as a copy of 'car1'. Afterward, we call the 'displayBrand()' function on both 'car1' and 'car2' to verify that they hold the same brand value.

The example demonstrates the usage of copy constructors in C++. Copy constructors are useful for creating new objects that are copies of existing objects, ensuring the proper initialization of member variables. By defining a copy constructor, developers can perform a deep copy of data, preventing unintended side effects and maintaining the integrity of the copied object. Understanding and implementing copy constructors correctly are crucial for handling object copies and ensuring the expected behavior of objects in C++ programs.

Section 4.10 - Copy Assignment Operator

Default Assignment Operator Behavior

In C++, the default assignment operator ('operator=') is a member function automatically generated by the compiler if no custom assignment operator is provided in the class. It performs a member-wise assignment, copying each member variable from the source object to the target object. The default assignment operator assigns the values of the member variables of the source object to the corresponding member variables of the target object. However, it does a shallow copy, which means that if the class contains dynamically allocated memory or resources, a shallow copy would simply copy the memory addresses rather than creating independent copies. This can lead to issues when multiple objects share the same resources. Therefore, if a class contains dynamically allocated memory or resources, it is recommended to define a custom assignment operator to perform a deep copy, ensuring that each object has its own separate copy of the resources.

Overloading the Assignment Operator

In C++, the assignment operator ('operator=') can be overloaded to provide a custom implementation for assigning one object to another of the same class. Overloading the assignment operator allows for more control over how the assignment operation is performed, especially when dealing with complex objects or dynamically allocated memory. By providing a custom assignment operator, developers can define their own rules for copying or transferring data between objects. This can involve deep copying of dynamically allocated memory, handling of resources, or any other necessary operations to ensure proper assignment semantics. Overloading the assignment operator enables greater flexibility and customization when it comes to assigning objects, allowing for more precise control over the behavior of the assignment operation in C++ programs.

Overloading Assignment Operator Example

Below is an example of overloading the assignment operator in C++:

```
1   #include <iostream>
2
3   class Car {
4   private:
5       std::string brand;
6
```

```
public:
    Car(const std::string& carBrand) : brand(carBrand) {}

    // Overloaded assignment operator
    Car& operator=(const Car& other) {
        if (this != &other) {
            brand = other.brand;
            std::cout << "Assignment operator called." << std::endl;
        }
        return *this;
    }

    void displayBrand() {
        std::cout << "Brand: " << brand << std::endl;
    }
};

int main() {
    Car car1("Toyota");
    Car car2("Honda");

    car1.displayBrand();  // Output: Brand: Toyota
    car2.displayBrand();  // Output: Brand: Honda

    car2 = car1;  // Overloaded assignment operator is invoked here

    car1.displayBrand();  // Output: Brand: Toyota
    car2.displayBrand();  // Output: Brand: Toyota

    return 0;
}

```

In this code, we have a 'Car' class with a 'brand' attribute and a constructor 'Car(const std::string& carBrand)' to initialize the 'brand' attribute

The key aspect is the overloaded assignment operator 'Car& operator=(const Car& other)'. This assignment operator is defined within the 'Car' class and provides a custom implementation for assigning one 'Car' object to another. It first checks if the source object ('other') is not the same as the target object ('this') to avoid self-assignment. Then, it assigns the 'brand' value of the source object to the target object and outputs a message indicating that the assignment operator is called. The assignment operator returns a reference to the modified object. In the 'main()' function, we create two 'Car' objects, 'car1' and 'car2', with different brand names. We call the 'displayBrand()' function to verify their respective brand names. Next, we assign 'car1' to 'car2' using the overloaded assignment operator. This invokes the assignment operator, which copies the 'brand' value of 'car1' to 'car2'. After the assignment, we call the 'displayBrand()' function again to confirm that 'car2' now has the same brand as 'car1'.

The example demonstrates the overloading of the assignment operator in C++. By providing a custom implementation for the assignment operator, developers can define their own rules for copying or transferring data between objects of the same class. This allows for greater control over the behavior of the assignment operation, ensuring that objects are assigned correctly and any necessary operations, such as deep copying, are performed. Overloading the assignment operator provides flexibility and customization in handling object assignments in C++ programs.

Section 4.11 - Rule of Three

The Rule of Three in object-oriented programming (OOP) states that if a class requires the explicit definition of a destructor, copy constructor, or copy assignment operator, then it most likely requires all three. This rule arises from the need to properly manage resources, particularly when a class contains dynamically allocated memory or other non-copyable resources. By implementing all three functions, developers ensure that objects are correctly initialized, copied, and deallocated. Failure to adhere to the Rule of Three can lead to issues such as memory leaks, resource leaks, or unexpected object state modifications. By following this rule, developers can ensure proper resource management and maintain the integrity and behavior of objects in OOP programs.

Rule of Three Example

Below is an example of the rule of three in C++:

```
#include <iostream>
#include <cstring>

```

```cpp
class String {
private:
    char* data;

public:
    String(const char* str) {
        size_t length = strlen(str);
        data = new char[length + 1];
        strcpy(data, str);
    }

    ~String() {
        delete[] data;
    }

    String(const String& other) {
        size_t length = strlen(other.data);
        data = new char[length + 1];
        strcpy(data, other.data);
    }

    String& operator=(const String& other) {
        if (this != &other) {
            delete[] data;
            size_t length = strlen(other.data);
            data = new char[length + 1];
            strcpy(data, other.data);
        }
        return *this;
    }

    void print() {
        std::cout << data << std::endl;
    }
};

int main() {
    String s1("Hello");
    String s2 = s1;  // Copy constructor is invoked

    s1.print();  // Output: Hello
    s2.print();  // Output: Hello

    String s3("World");
    s2 = s3;  // Copy assignment operator is invoked

    s2.print();  // Output: World

    return 0;
}
```

In this code, we have a ‘String’ class that represents a string object. The class manages a dynamically allocated character array ‘data’ to hold the string

The example adheres to the Rule of Three by defining the destructor, copy constructor, and copy assignment operator. The destructor ‘S̃tring()’ deallocates the dynamically allocated memory held by ‘data’. The copy constructor ‘String(const String& other)’ creates a new ‘String’ object by making a deep copy of the ‘data’ from the source object. The copy assignment operator ‘String& operator=(const String& other)’ assigns the ‘data’ from the source object to the target object, handling the deallocation and reallocation of memory.

In the ‘main()’ function, we create ‘String’ objects ‘s1’, ‘s2’, and ‘s3’. We initialize ‘s1’ with the string "Hello" and then create ‘s2’ as a copy of ‘s1’ using the copy constructor. We print the contents of both ‘s1’ and ‘s2’ to verify that they have the same string value. Next, we create ‘s3’ with the string "World" and assign it to ‘s2’ using the copy assignment operator. This invokes the copy assignment operator, which deallocates the ‘data’ of ‘s2’ and copies the ‘data’ from ‘s3’. We print the contents of ‘s2’ to confirm that it now holds the string "World".

The example demonstrates the Rule of Three in action, where the destructor, copy constructor, and copy assignment operator are implemented to ensure proper resource management and object behavior. By following this rule, developers can prevent memory leaks, resource leaks, or unexpected object state modifications caused by incorrect copying or destruction of objects. Adhering to the Rule of Three is crucial when dealing with classes that manage resources or have non-copyable members, ensuring the correct behavior and integrity of objects in C++ programs.

The second chapter of this week is **Chapter 5: Lists**.

Section 5.1 - List Abstract Data Type (ADT)

The list abstract data type (ADT) represents a collection of elements where each element is linked to the next element in the sequence. It allows for efficient insertion, deletion, and retrieval operations. The list ADT supports dynamic resizing, allowing for the storage of an arbitrary number of elements. Lists can be implemented using various data structures, such as linked lists or arrays. They provide flexibility in terms of element manipulation and are commonly used in scenarios where the order and accessibility of elements are important. The list ADT provides a versatile and efficient way to manage collections of data in computer programs, facilitating operations that involve sequential access and modification of elements.

List (ADT) Example

Below is an example of a list (ADT) in C++:

```cpp
#include <iostream>

class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

class LinkedList {
private:
    Node* head;
    Node* tail;

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

    void insert(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
            tail = newNode;
        } else {
            tail->next = newNode;
            tail = newNode;
        }
    }

    void display() {
        Node* current = head;
        while (current != nullptr) {
            std::cout << current->data << " ";
            current = current->next;
        }
        std::cout << std::endl;
    }
};

int main() {
    LinkedList myList;

    myList.insert(5);
    myList.insert(10);
    myList.insert(15);

    myList.display();  // Output: 5 10 15

    return 0;
}

```

In this code, we have a Node class that represents a node in a linked list. Each node holds an integer value (data) and a pointer to the next node (next).

The LinkedList class represents the list abstract data type. It consists of a head pointer (head) and a tail pointer (tail). The insert function inserts a new node with the given value at the end of the list. If the list is empty, the new node becomes both the head and the tail. Otherwise, the new node is appended after the current tail, and the tail pointer is updated accordingly. The display function traverses the list, starting from the head, and prints the data of each node. In the main() function, we create an instance of LinkedList named myList. We insert three elements into the list using the insert function, namely 5, 10, and 15. Finally, we call the display function to print the contents of the list, which outputs "5 10 15".

The example demonstrates the list abstract data type implemented using a linked list. It showcases the insertion operation, where elements are added to the end of the list, preserving the order. The linked list provides efficient insertion and sequential access to elements. It allows for the dynamic storage of elements, as nodes are dynamically allocated during insertion. The list ADT offers flexibility and versatility in managing

collections of data, enabling efficient manipulation and traversal of elements.

Section 5.2 - Singly-Linked Lists

Singly-Linked List Data Structure

A singly-linked list is a data structure in which each element, known as a node, contains data and a pointer to the next node in the list. The list is "singly-linked" because it allows traversal in one direction, from the head (the first node) to the tail (the last node). Singly-linked lists are dynamic data structures, allowing for efficient insertion and deletion of elements at the beginning or end of the list. However, accessing or modifying elements in the middle of the list can be less efficient due to the need to traverse from the head. Singly-linked lists are commonly used when the order of elements matters, and frequent insertions and deletions are expected, but random access is not a primary requirement. They provide a flexible and memory-efficient way to manage and manipulate collections of data in various programming scenarios.

Appending to Singly Linked List

Appending to a singly-linked list involves adding a new element to the end of the list. To perform this operation, the tail pointer of the list is utilized. If the list is initially empty, the new element becomes both the head and the tail. Otherwise, the tail pointer is updated to point to the new element, effectively making it the new tail. This operation is efficient in a singly-linked list since inserting at the end does not require traversing the entire list. By appending elements to a singly-linked list, the list can dynamically grow, maintaining the order of elements while enabling efficient insertion of new elements at the tail.

Prepending to Singly Linked List

Prepending to a singly-linked list involves adding a new element at the beginning of the list. This operation requires updating the head pointer of the list to point to the new element, making it the new head. If the list is initially empty, the new element becomes both the head and the tail. Prepending is an efficient operation in a singly-linked list since it does not require traversing the entire list. By prepending elements to a singly-linked list, the list can dynamically grow while maintaining the order of elements, allowing for efficient insertion of new elements at the beginning. This operation is useful in scenarios where the most recent or frequently accessed elements need to be accessed quickly without traversing the entire list.

Singly Linked List Example

Below is an example of a singly linked list in C++:

```cpp
#include <iostream>

class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

class SinglyLinkedList {
private:
    Node* head;
    Node* tail;

public:
    SinglyLinkedList() : head(nullptr), tail(nullptr) {}

    void append(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
            tail = newNode;
        } else {
            tail->next = newNode;
            tail = newNode;
        }
    }
```

```
29
30      void prepend(int value) {
31          Node* newNode = new Node(value);
32          if (head == nullptr) {
33              head = newNode;
34              tail = newNode;
35          } else {
36              newNode->next = head;
37              head = newNode;
38          }
39      }
40
41      void display() {
42          Node* current = head;
43          while (current != nullptr) {
44              std::cout << current->data << " ";
45              current = current->next;
46          }
47          std::cout << std::endl;
48      }
49  };
50
51  int main() {
52      SinglyLinkedList myList;
53
54      myList.append(5);
55      myList.append(10);
56      myList.append(15);
57
58      myList.display();  // Output: 5 10 15
59
60      myList.prepend(2);
61      myList.prepend(1);
62
63      myList.display();  // Output: 1 2 5 10 15
64
65      return 0;
66  }
67
```

The 'SinglyLinkedList' class represents the singly-linked list data structure. It consists of a head pointer ('head') and a tail pointer ('tail'). The 'append' function adds a new node with the given value at the end of the list. If the list is empty, the new node becomes both the head and the tail. Otherwise, the new node is appended after the current tail, and the tail pointer is updated accordingly. The 'prepend' function adds a new node with the given value at the beginning of the list. If the list is empty, the new node becomes both the head and the tail. Otherwise, the new node is inserted before the current head, and the head pointer is updated. The 'display' function traverses the list and prints the data of each node.

In the 'main()' function, we create an instance of 'SinglyLinkedList' named 'myList'. We append three elements to the list using the 'append' function, namely 5, 10, and 15. We then display the contents of the list, which outputs "5 10 15". Next, we prepend two elements to the list using the 'prepend' function, namely 2 and 1. Finally, we display the updated contents of the list, which outputs "1 2 5 10 15".

The example demonstrates the concepts of appending and prepending to a singly-linked list. The 'append' operation adds elements at the end, while the 'prepend' operation adds elements at the beginning. These operations allow for dynamic growth of the list while preserving the order of elements. By using the appropriate pointers, the list is efficiently updated without the need to traverse the entire list. Singly-linked lists provide flexibility in adding elements at either end, allowing for efficient insertion of new elements in various programming scenarios.

Section 5.3 - List Data Structure

The list data structure is a fundamental abstract data type that represents a collection of elements. Lists provide a dynamic and flexible way to store and manage data, allowing for efficient insertion, deletion, and retrieval operations. The key characteristic of lists is that they maintain the order of elements as they are added or removed. This order can be based on the position of the elements (e.g., positional index) or on the values of the elements themselves.

Lists can be implemented using different underlying data structures, such as arrays or linked lists. Each implementation has its advantages and trade-offs. Arrays provide fast random access to elements but can be less efficient for insertions and deletions, especially in the middle of the list, as it requires shifting elements. Linked lists, on the other hand, offer efficient insertions and deletions but require sequential traversal for accessing elements. Linked lists also have the flexibility to grow dynamically by allocating memory for new elements as needed.

Lists support various operations to manipulate the elements, including appending, prepending, inserting, deleting,

and searching. These operations allow for the modification and rearrangement of elements within the list. Lists can also support additional functionality such as sorting, merging, and splitting.

Lists are widely used in computer programming and software development, as they provide an essential building block for many other data structures and algorithms. They are particularly useful in scenarios where the order of elements matters and frequent insertions and deletions are expected. Lists are commonly employed in applications involving data processing, task scheduling, file systems, and more.

Overall, the list data structure offers a flexible and efficient way to organize and manage collections of elements, making it a fundamental tool in computer science and programming. It provides a foundation for implementing more complex data structures and algorithms, while also serving as a versatile and practical solution for various programming tasks.

Section 5.4 - Singly-Linked Lists: Insert

Inserting in a singly-linked list involves adding a new element at a specific position within the list. This operation requires updating the appropriate pointers to maintain the integrity and order of the list. Insertion can be performed at the beginning, in the middle, or at the end of the list. To insert an element at the beginning, the head pointer is updated to point to the new node, while its next pointer is set to the current head. For inserting in the middle, the pointers of the adjacent nodes are modified to link the new node appropriately. Inserting at the end involves updating the tail pointer to point to the new node and setting its next pointer to nullptr. Inserting in a singly-linked list is efficient for operations that involve updating only a few pointers, allowing for dynamic growth and modification of the list while preserving the order of elements.

Singly-Linked List Insert Example

Below is an example of inserting into a singly-linked list in C++:

```cpp
#include <iostream>

class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

class SinglyLinkedList {
private:
    Node* head;
    Node* tail;

public:
    SinglyLinkedList() : head(nullptr), tail(nullptr) {}

    void insert(int value, int position) {
        Node* newNode = new Node(value);

        if (head == nullptr || position == 0) {
            newNode->next = head;
            head = newNode;
            if (tail == nullptr) {
                tail = newNode;
            }
        } else {
            Node* current = head;
            int count = 0;

            while (current->next != nullptr && count < position - 1) {
                current = current->next;
                count++;
            }

            newNode->next = current->next;
            current->next = newNode;

            if (newNode->next == nullptr) {
                tail = newNode;
            }
        }
    }

    void display() {
        Node* current = head;
        while (current != nullptr) {
            std::cout << current->data << " ";
```

```
            current = current->next;
        }
        std::cout << std::endl;
    }
};

int main() {
    SinglyLinkedList myList;

    myList.insert(5, 0);   // Insert 5 at position 0
    myList.insert(10, 1);  // Insert 10 at position 1
    myList.insert(7, 1);   // Insert 7 at position 1

    myList.display();  // Output: 5 7 10

    return 0;
}

```

The 'SinglyLinkedList' class represents the singly-linked list data structure. It consists of a head pointer ('head') and a tail pointer ('tail'). The 'insert' function adds a new node with the given value at the specified position within the list. If the list is empty or the position is 0, the new node becomes the new head, and its next pointer is set to the previous head. If the position is not at the beginning, the function iterates through the list to find the appropriate position. Once found, the new node is inserted by updating the pointers of the previous and next nodes accordingly. The 'display' function traverses the list and prints the data of each node.

In the 'main()' function, we create an instance of 'SinglyLinkedList' named 'myList'. We insert three elements into the list using the 'insert' function, with values 5, 10, and 7, respectively, at positions 0, 1, and 1. Finally, we call the 'display' function to print the contents of the list, which outputs "5 7 10".

The example demonstrates the concept of inserting an element at a specific position in a singly-linked list. The 'insert' operation allows for dynamic modification of the list by rearranging the pointers of the nodes. By updating the appropriate pointers, the new node is seamlessly integrated into the list while maintaining the order of elements. Singly-linked lists provide an efficient way to insert elements at various positions, enabling flexibility and versatility in managing collections of data.

Section 5.5 - Singly-Linked Lists: Remove

Removing elements from a singly-linked list involves deleting a node at a specific position or with a particular value. To remove a node at a given position, the pointers of the preceding and following nodes are adjusted to bypass the node being removed, effectively removing it from the list. If the node to be removed is the head, the head pointer is updated to point to the next node. Removing a node based on a value requires traversing the list to find the node with the desired value. Once found, the pointers of the preceding and following nodes are adjusted to exclude the node with the target value. Removing elements from a singly-linked list is an efficient operation, especially when the position or value to be removed is known, as it involves updating a few pointers rather than shifting elements. This flexibility in removing nodes allows for dynamic modification of the list while maintaining the order of the remaining elements.

Singly-Linked List Remove Example

Below is an example of removing in a singly-linked list in C++:

```
#include <iostream>

class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

class SinglyLinkedList {
private:
    Node* head;
    Node* tail;

public:
    SinglyLinkedList() : head(nullptr), tail(nullptr) {}

    void remove(int position) {
```

```
20          if (head == nullptr)
21              return;
22
23          Node* current = head;
24
25          if (position == 0) {
26              head = head->next;
27              delete current;
28
29              if (head == nullptr)
30                  tail = nullptr;
31          } else {
32              Node* previous = nullptr;
33              int count = 0;
34
35              while (current != nullptr && count < position) {
36                  previous = current;
37                  current = current->next;
38                  count++;
39              }
40
41              if (current != nullptr) {
42                  previous->next = current->next;
43
44                  if (previous->next == nullptr)
45                      tail = previous;
46
47                  delete current;
48              }
49          }
50      }
51
52      void display() {
53          Node* current = head;
54          while (current != nullptr) {
55              std::cout << current->data << " ";
56              current = current->next;
57          }
58          std::cout << std::endl;
59      }
60  };
61
62  int main() {
63      SinglyLinkedList myList;
64
65      myList.remove(0);  // No effect, list is empty
66
67      myList.display();  // Output: (empty)
68
69      myList.insert(5, 0);   // Insert 5 at position 0
70      myList.insert(10, 1);  // Insert 10 at position 1
71      myList.insert(7, 1);   // Insert 7 at position 1
72
73      myList.display();  // Output: 5 7 10
74
75      myList.remove(1);  // Remove element at position 1
76
77      myList.display();  // Output: 5 10
78
79      return 0;
80  }
81
```

The ‘SinglyLinkedList’ class represents the singly-linked list data structure. It consists of a head pointer (‘head’) and a tail pointer (‘tail’). The ‘remove’ function removes a node at the specified position within the list. If the list is empty or the position is out of range, the function does nothing. If the position is 0, the head is updated to the next node, and the original head is deleted. If the position is not 0, the function iterates through the list to find the node at the specified position. Once found, the function updates the pointers of the previous and following nodes to exclude the node being removed and deletes the target node. The ‘display’ function traverses the list and prints the data of each node.

In the ‘main()’ function, we create an instance of ‘SinglyLinkedList’ named ‘myList’. We attempt to remove an element from an empty list, which has no effect. We then insert three elements into the list using the ‘insert’ function, with values 5, 10, and 7, respectively, at positions 0, 1, and 1. We display the contents of the list, which outputs "5 7 10". Next, we ‘remove’ the element at position 1 using the remove function. Finally, we display the updated contents of the list, which outputs "5 10" since the element with value 7 has been successfully removed.

The example demonstrates the concept of removing elements from a singly-linked list. The ‘remove’ operation allows for the dynamic modification of the list by adjusting the pointers of the preceding and following nodes. By updating these pointers, the target node is effectively bypassed and removed from the list while preserving the order of the remaining elements. Removing nodes from a singly-linked list is an efficient operation, as it involves updating a few pointers rather than shifting elements. This flexibility in removing nodes provides practicality and versatility in managing collections of data with varying needs.

Section 5.6 - Linked List Search

Linked list search involves finding a specific value or element within a linked list data structure. The search operation requires traversing through the list, examining each node's data until the desired value is found or the end of the list is reached. During the traversal, the data in each node is compared with the target value, and if a match is found, the search operation can be considered successful. Linked list search is a linear search process that has a time complexity of $\mathcal{O}(n)$ in the worst case, where n is the number of elements in the list. The efficiency of the search operation can be improved by using techniques such as maintaining a sorted linked list or utilizing additional data structures like hash tables or binary search trees.

Linked List Search Example

Below is an example of a linked list search in C++:

```cpp
#include <iostream>

class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

class SinglyLinkedList {
private:
    Node* head;

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

    bool search(int value) {
        Node* current = head;

        while (current != nullptr) {
            if (current->data == value) {
                return true;  // Value found
            }
            current = current->next;
        }

        return false;  // Value not found
    }

    void insert(int value) {
        Node* newNode = new Node(value);

        if (head == nullptr) {
            head = newNode;
        } else {
            Node* current = head;
            while (current->next != nullptr) {
                current = current->next;
            }
            current->next = newNode;
        }
    }
};

int main() {
    SinglyLinkedList myList;

    myList.insert(5);
    myList.insert(10);
    myList.insert(7);

    int searchValue = 10;
    if (myList.search(searchValue)) {
        std::cout << "Value " << searchValue << " found in the linked list."
        << std::endl;
    } else {
        std::cout << "Value " << searchValue << " not found in the linked list."
        << std::endl;
    }

    return 0;
}

```

The 'SinglyLinkedList' class represents the singly-linked list data structure. It consists of a head pointer

('head'). The 'search' function performs a search operation to find a specific value within the list. It starts from the head and traverses the list by comparing the value in each node with the target value. If a match is found, the function returns 'true', indicating that the value exists in the list. If the end of the list is reached without finding a match, the function returns 'false', indicating that the value does not exist in the list. The 'insert' function adds new nodes to the end of the list.

In the 'main()' function, we create an instance of 'SinglyLinkedList' named 'myList'. We insert three elements with values 5, 10, and 7, respectively. We then perform a search for the value 10 using the 'search' function. Since 10 is present in the list, the program outputs "Value 10 found in the linked list." The example demonstrates how to search for a specific value within a linked list using a linear search approach. By traversing the list and comparing each element, the search operation efficiently determines the presence or absence of a target value in the linked list.

Section 5.7 - Doubly-Linked Lists

Overview

A doubly linked list is a type of linked list where each node contains two pointers: one pointing to the previous node and another pointing to the next node. This bidirectional linkage allows traversal in both directions, enabling efficient insertion, deletion, and searching operations. Each node in a doubly linked list stores data and maintains references to the previous and next nodes, except for the first node (head) that only has a next pointer, and the last node (tail) that only has a previous pointer. The ability to traverse in both directions makes doubly linked lists useful for scenarios that require backward traversal or frequent insertions and deletions at both ends of the list. However, the presence of additional pointers increases the memory overhead compared to singly linked lists.

Appending to a Doubly-Linked List

Appending to a doubly linked list involves adding a new node at the end of the list. To append a node, a new node is created with the desired data, and the necessary pointers are adjusted to establish the appropriate connections. If the list is empty, the new node becomes both the head and the tail of the list. Otherwise, the new node's previous pointer is set to the current tail, and the current tail's next pointer is updated to point to the new node. Finally, the tail pointer is updated to point to the newly appended node. By properly updating the pointers, the new node is seamlessly integrated into the existing doubly linked list structure, preserving the order of elements and enabling efficient access to both the head and tail of the list.

Prepending to a Doubly-Linked List

Prepending to a doubly linked list involves adding a new node at the beginning of the list. To prepend a node, a new node is created with the desired data, and the necessary pointers are adjusted to establish the appropriate connections. If the list is empty, the new node becomes both the head and the tail of the list. Otherwise, the new node's next pointer is set to the current head, and the current head's previous pointer is updated to point to the new node. Finally, the head pointer is updated to point to the newly prepended node. By properly updating the pointers, the new node is seamlessly integrated into the existing doubly linked list structure, maintaining the order of elements and allowing efficient access to both the head and tail of the list. Prepending to a doubly linked list is an efficient operation, as it involves adjusting a few pointers without the need to traverse the entire list.

Doubly-Linked List Example

Below is an example of doubly-linked lists in C++:

```cpp
#include <iostream>

class Node {
public:
    int data;
    Node* prev;
    Node* next;

    Node(int value) : data(value), prev(nullptr), next(nullptr) {}
};

```

```cpp
class DoublyLinkedList {
private:
    Node* head;
    Node* tail;

public:
    DoublyLinkedList() : head(nullptr), tail(nullptr) {}

    void append(int value) {
        Node* newNode = new Node(value);

        if (head == nullptr) {
            head = newNode;
            tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }

    void prepend(int value) {
        Node* newNode = new Node(value);

        if (head == nullptr) {
            head = newNode;
            tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
    }

    void display() {
        Node* current = head;
        while (current != nullptr) {
            std::cout << current->data << " ";
            current = current->next;
        }
        std::cout << std::endl;
    }
};

int main() {
    DoublyLinkedList myList;

    myList.append(5);
    myList.append(10);
    myList.append(7);

    myList.display();  // Output: 5 10 7

    myList.prepend(3);
    myList.prepend(1);

    myList.display();  // Output: 1 3 5 10 7

    return 0;
}
```

The ‘DoublyLinkedList’ class represents the doubly linked list data structure. It consists of a head pointer (‘head’) and a tail pointer (‘tail’). The ‘append’ function appends a new node with the specified value at the end of the list. It creates a new node, adjusts the pointers of the current tail node and the new node, and updates the tail pointer accordingly. The ‘prepend’ function prepends a new node with the specified value at the beginning of the list. It creates a new node, adjusts the pointers of the current head node and the new node, and updates the head pointer accordingly. The ‘display’ function traverses the list and prints the data of each node.

In the ‘main()’ function, we create an instance of ‘DoublyLinkedList’ named ‘myList’. We append three elements with values 5, 10, and 7, respectively, and display the contents of the list. The output is "5 10 7". Next, we prepend two elements with values 3 and 1, respectively, and display the updated contents of the list. The output is "1 3 5 10 7", showing that the new nodes are correctly appended and prepended to the doubly linked list.

The example demonstrates how to append and prepend nodes to a doubly linked list. By properly adjusting the pointers of the nodes and updating the head and tail pointers, the new nodes are seamlessly integrated into the existing list structure. Doubly linked lists provide efficient operations for adding elements at both ends, allowing for flexibility in managing and manipulating the list’s contents.

Section 5.8 - Doubly-Linked Lists: Insert

Inserting in a doubly linked list involves adding a new node at a specific position within the list. The insertion operation requires adjusting the pointers of the neighboring nodes to maintain the correct order. The process begins by creating a new node with the desired data. Then, the next and previous pointers of the new node are modified to establish the connections with the neighboring nodes. By updating the pointers of the preceding and succeeding nodes accordingly, the new node is seamlessly integrated into the list structure. Inserting in a doubly linked list provides flexibility in modifying the list by allowing the addition of elements at arbitrary positions, making it suitable for scenarios that require dynamic data manipulation and efficient data insertion.

Doubly-Linked List Insert Example

Below is an example of inserting into doubly-linked lists in C++:

```cpp
#include <iostream>

class Node {
public:
    int data;
    Node* prev;
    Node* next;

    Node(int value) : data(value), prev(nullptr), next(nullptr) {}
};

class DoublyLinkedList {
private:
    Node* head;
    Node* tail;

public:
    DoublyLinkedList() : head(nullptr), tail(nullptr) {}

    void insert(int value, int position) {
        Node* newNode = new Node(value);

        if (head == nullptr) {
            head = newNode;
            tail = newNode;
        } else if (position == 0) {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        } else {
            Node* current = head;
            int count = 0;

            while (current != nullptr && count < position) {
                current = current->next;
                count++;
            }

            if (current == nullptr) {
                tail->next = newNode;
                newNode->prev = tail;
                tail = newNode;
            } else {
                newNode->prev = current->prev;
                newNode->next = current;
                current->prev->next = newNode;
                current->prev = newNode;
            }
        }
    }

    void display() {
        Node* current = head;
        while (current != nullptr) {
            std::cout << current->data << " ";
            current = current->next;
        }
        std::cout << std::endl;
    }
};

int main() {
    DoublyLinkedList myList;

    myList.insert(5, 0);   // Inserting at position 0
    myList.insert(10, 1);  // Inserting at position 1
    myList.insert(7, 1);   // Inserting at position 1

    myList.display();  // Output: 5 7 10

    return 0;
}
```

```
73
```

The 'DoublyLinkedList' class represents the doubly linked list data structure. It consists of a head pointer ('head') and a tail pointer ('tail'). The 'insert' function inserts a new node with the specified value at the given position within the list. It handles different cases: if the list is empty, the new node becomes both the head and the tail; if the position is 0, the new node is prepended to the list; otherwise, the function traverses the list to find the desired position. If the position exceeds the length of the list, the new node is appended to the end. Otherwise, the new node is inserted at the specified position by updating the pointers of the neighboring nodes accordingly. The 'display' function is used to traverse the list and print its contents.

In the 'main()' function, we create an instance of 'DoublyLinkedList' named 'myList'. We insert three elements with values 5, 10, and 7, respectively, at different positions within the list. After inserting the nodes, we display the contents of the list, which outputs "5 7 10". The example demonstrates how to insert a node at a specific position in a doubly linked list. By appropriately adjusting the pointers of the neighboring nodes, the new node is seamlessly integrated into the list structure while preserving the order of the elements. Inserting in a doubly linked list provides flexibility in managing the list's contents.

Section 5.9 - Doubly-Linked Lists: Remove

Removing nodes from a doubly linked list involves unlinking a node from the list while maintaining the integrity of the remaining nodes. The removal process requires updating the pointers of the neighboring nodes to bypass the node being removed. To remove a node, the previous node's next pointer is connected to the next node, and the next node's previous pointer is connected to the previous node. Additionally, if the node being removed is the head or tail of the list, the head or tail pointers are updated accordingly. By properly adjusting the pointers and deallocating the memory occupied by the removed node, the doubly linked list structure is maintained, and the list's integrity is preserved. Removing nodes from a doubly linked list allows for efficient deletion of elements at arbitrary positions, enabling dynamic data manipulation and effective memory management.

Doubly-Linked List Remove Example

Below is an example of removing in doubly-linked lists in C++:

```cpp
#include <iostream>

class Node {
public:
    int data;
    Node* prev;
    Node* next;

    Node(int value) : data(value), prev(nullptr), next(nullptr) {}
};

class DoublyLinkedList {
private:
    Node* head;
    Node* tail;

public:
    DoublyLinkedList() : head(nullptr), tail(nullptr) {}

    void append(int value) {
        Node* newNode = new Node(value);

        if (head == nullptr) {
            head = newNode;
            tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }

    void remove(int value) {
        Node* current = head;

        while (current != nullptr) {
            if (current->data == value) {
                if (current == head) {
                    head = current->next;
                    if (head != nullptr) {
```

```
                        head->prev = nullptr;
                    }
                } else if (current == tail) {
                    tail = current->prev;
                    if (tail != nullptr) {
                        tail->next = nullptr;
                    }
                } else {
                    current->prev->next = current->next;
                    current->next->prev = current->prev;
                }

                delete current;
                return;
            }

            current = current->next;
        }
    }

    void display() {
        Node* current = head;
        while (current != nullptr) {
            std::cout << current->data << " ";
            current = current->next;
        }
        std::cout << std::endl;
    }
};

int main() {
    DoublyLinkedList myList;

    myList.append(5);
    myList.append(10);
    myList.append(7);

    myList.display();  // Output: 5 10 7

    myList.remove(10);

    myList.display();  // Output: 5 7

    return 0;
}

```

The 'DoublyLinkedList' class represents the doubly linked list data structure. It consists of a head pointer ('head') and a tail pointer ('tail'). The 'append' function is used to append a new node with the specified value at the end of the list. It handles the case where the list is empty and adjusts the pointers accordingly. The 'remove' function removes a node with the specified value from the list. It traverses the list and checks if the current node matches the target value. If a match is found, the appropriate pointers are adjusted to bypass the node being removed. If the node being removed is the head or tail, the head or tail pointers are updated accordingly. Finally, the memory occupied by the removed node is deallocated. The 'display' function is used to traverse the list and print its contents.

In the 'main()' function, we create an instance of 'DoublyLinkedList' named 'myList' and append three elements with values 5, 10, and 7, respectively. After appending the nodes, we display the contents of the list, which outputs "5 10 7". We then call the 'remove' function to remove the node with value 10 from the list. After the removal, we display the updated contents of the list, which outputs "5 7". The example demonstrates how to remove a node from a doubly linked list

Section 5.10 - Linked List Traversal

Overview

Linked list traversal refers to the process of visiting each node in a linked list in order to perform some operation or access the data stored in the nodes. The traversal typically starts from the head of the list and iterates through the nodes until the end of the list is reached. During traversal, the data in each node can be processed, displayed, or used for some computation. By following the next pointers of the nodes, the traversal can effectively access all the elements in the linked list. Linked list traversal is an essential operation for analyzing, modifying, or displaying the contents of a linked list, and it allows for efficient and sequential access to the elements stored in the list.

Doubly-Linked List Traversal

Linked list traversal in a doubly linked list is similar to traversal in a singly linked list but with the added capability of traversing in both forward and backward directions. Starting from the head or tail of the list, the traversal can move through the list by following either the next or prev pointers of each node. This bidirectional traversal allows for flexibility in accessing and processing the data in each node. The traversal can be performed to display the elements, perform computations, search for specific values, or modify the data within the nodes. By leveraging the prev and next pointers, linked list traversal in a doubly linked list enables efficient and convenient navigation through the list in both directions, making it a versatile tool for various operations on the list's elements.

Linked List Traversal Example

Below is an example of traversing in a linked list in C++:

```cpp
#include <iostream>

class Node {
public:
    int data;
    Node* prev;
    Node* next;

    Node(int value) : data(value), prev(nullptr), next(nullptr) {}
};

class DoublyLinkedList {
private:
    Node* head;
    Node* tail;

public:
    DoublyLinkedList() : head(nullptr), tail(nullptr) {}

    void append(int value) {
        Node* newNode = new Node(value);

        if (head == nullptr) {
            head = newNode;
            tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }

    void displayForward() {
        Node* current = head;
        while (current != nullptr) {
            std::cout << current->data << " ";
            current = current->next;
        }
        std::cout << std::endl;
    }

    void displayBackward() {
        Node* current = tail;
        while (current != nullptr) {
            std::cout << current->data << " ";
            current = current->prev;
        }
        std::cout << std::endl;
    }
};

int main() {
    DoublyLinkedList myList;

    myList.append(5);
    myList.append(10);
    myList.append(7);

    std::cout << "Forward traversal: ";
    myList.displayForward();  // Output: 5 10 7

    std::cout << "Backward traversal: ";
    myList.displayBackward();  // Output: 7 10 5

    return 0;
}

```

The 'DoublyLinkedList' class represents the doubly linked list data structure. It consists of a head pointer ('head') and a tail pointer ('tail'). The 'append' function is used to append a new node with the specified value at the end of the list, adjusting the pointers accordingly.

The 'displayForward' function is responsible for traversing the list in the forward direction, starting from

the head. It iterates through the list by following the next pointers of each node and prints the data stored in each node.

The 'displayBackward' function performs the traversal in the backward direction, starting from the tail. It iterates through the list by following the prev pointers of each node and prints the data stored in each node.

In the 'main()' function, we create an instance of 'DoublyLinkedList' named 'myList' and append three elements with values 5, 10, and 7, respectively. We then call the 'displayForward' and 'displayBackward' functions to traverse and print the contents of the list in both forward and backward directions. The output demonstrates the successful traversal of the doubly linked list in both directions, displaying "5 10 7" and "7 10 5" respectively. This example highlights how the prev and next pointers in a doubly linked list facilitate bidirectional traversal, allowing for convenient access and processing of the elements stored in the list.

Section 5.11 - Sorting Linked Lists

Sorting for Singly-Linked Lists

Sorting a singly linked list involves rearranging its nodes in a specific order based on the values they hold. Sorting algorithms like merge sort, insertion sort, or bubble sort can be applied to achieve this task. Since a singly linked list only allows traversal in the forward direction, sorting the list can be slightly more challenging compared to a doubly linked list. The process typically involves rearranging the next pointers of the nodes while keeping track of the previous nodes and updating the head pointer as necessary. Sorting a singly linked list improves its organization and allows for more efficient searching and accessing of elements. It is an important operation to enhance the functionality and performance of a singly linked list data structure.

Sorting for Doubly-Linked Lists

Sorting a doubly linked list involves rearranging its nodes in a specific order based on the values they hold. There are various sorting algorithms that can be applied to accomplish this task, such as bubble sort, insertion sort, selection sort, merge sort, or quicksort. The sorting process typically involves comparing adjacent nodes and swapping their positions until the entire list is sorted. The bidirectional nature of the doubly linked list allows for efficient swapping of nodes by adjusting the prev and next pointers. Sorting a doubly linked list provides an organized arrangement of its elements, making it easier to search, access, or perform operations on the list in a structured manner. It is a fundamental operation that contributes to improving the efficiency and usability of the doubly linked list data structure.

Sorting a Linked List Example

Below is an example of sorting a linked list in C++:

```cpp
#include <iostream>

class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

class SinglyLinkedList {
private:
    Node* head;

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

    void insert(int value) {
        Node* newNode = new Node(value);

        if (head == nullptr) {
            head = newNode;
        } else {
            Node* current = head;
            while (current->next != nullptr) {
                current = current->next;
            }
```

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

    void display() {
        Node* current = head;
        while (current != nullptr) {
            std::cout << current->data << " ";
            current = current->next;
        }
        std::cout << std::endl;
    }

    Node* merge(Node* first, Node* second) {
        if (first == nullptr)
            return second;
        if (second == nullptr)
            return first;

        Node* result;
        if (first->data <= second->data) {
            result = first;
            result->next = merge(first->next, second);
        } else {
            result = second;
            result->next = merge(first, second->next);
        }
        return result;
    }

    void split(Node* source, Node** frontRef, Node** backRef) {
        if (source == nullptr || source->next == nullptr) {
            *frontRef = source;
            *backRef = nullptr;
            return;
        }

        Node* slow = source;
        Node* fast = source->next;

        while (fast != nullptr) {
            fast = fast->next;
            if (fast != nullptr) {
                slow = slow->next;
                fast = fast->next;
            }
        }

        *frontRef = source;
        *backRef = slow->next;
        slow->next = nullptr;
    }

    void mergeSort(Node** headRef) {
        Node* head = *headRef;
        Node* a;
        Node* b;

        if (head == nullptr || head->next == nullptr)
            return;

        split(head, &a, &b);

        mergeSort(&a);
        mergeSort(&b);

        *headRef = merge(a, b);
    }

    void sort() {
        mergeSort(&head);
    }
};

int main() {
    SinglyLinkedList myList;

    myList.insert(5);
    myList.insert(10);
    myList.insert(2);
    myList.insert(8);
    myList.insert(3);

    std::cout << "Before sorting: ";
    myList.display();  // Output: 5 10 2 8 3

    myList.sort();

    std::cout << "After sorting: ";
    myList.display();  // Output: 2 3 5 8 10

    return 0;
```

```
120     }
121
```

The ‘SinglyLinkedList’ class represents the singly linked list data structure. It consists of a head pointer (‘head’) that points to the first node in the list. The ‘insert’ function is used to insert a new node with the specified value at the end of the list.

The ‘display’ function traverses the list from the head node to the last node and prints the data stored in each node. The ‘merge’ function is a helper function that merges two sorted linked lists into a single sorted list. The ‘split’ function divides the list into two halves, using the slow and fast pointer approach. The ‘mergeSort’ function recursively splits the list into smaller sublists, sorts them using merge sort, and then merges them back together. The ‘sort’ function serves as an entry point for sorting the linked list. It calls the ‘mergeSort’ function to perform the sorting operation.

In the ‘main()’ function, we create an instance of ‘SinglyLinkedList’ named ‘myList’ and insert five elements with values 5, 10, 2, 8, and 3, respectively. We then call the ‘display’ function to print the contents of the list before sorting. Next, we call the ‘sort’ function to sort the list using merge sort. Finally, we call the ‘display’ function again to print the sorted list. The output demonstrates the successful sorting of the singly linked list, displaying "2 3 5 8 10". This example showcases the implementation of the merge sort algorithm on a singly linked list, effectively arranging the elements in ascending order.

Section 5.12 - Linked List Dummy Nodes

Overview

Dummy nodes, also known as sentinel nodes, are special nodes added to the beginning or end of a linked list. These nodes do not hold any meaningful data but serve as placeholders or markers to simplify the operations performed on the list. Dummy nodes can be used to eliminate special cases and edge conditions when inserting or removing elements, especially at the beginning or end of the list. They provide a consistent structure for traversal and manipulation, ensuring that the list’s head and tail pointers always point to valid nodes. By using dummy nodes, code complexity can be reduced, and operations on linked lists can be streamlined, making the implementation more efficient and easier to manage.

Singly-Linked List Implemention

In the context of singly linked lists, dummy nodes are often used to simplify certain operations, especially at the beginning or end of the list. A common approach is to add a dummy node at the beginning of the list, referred to as the "dummy head." This dummy node acts as a placeholder and allows consistent handling of the list’s head pointer, even when inserting or removing elements. With a dummy head, the head pointer always points to a valid node, eliminating the need for special cases when manipulating the first element. Dummy nodes in singly linked lists enhance code readability and maintain consistency in operations, making it easier to handle edge cases and simplifying the implementation of algorithms and data structures that rely on singly linked lists.

Doubly-Linked List Implementation

In the context of doubly linked lists, dummy nodes are commonly used to simplify operations at both the beginning and end of the list. Similar to singly linked lists, a dummy node can be added at the beginning of the list, called the "dummy head," and another dummy node at the end, known as the "dummy tail." These dummy nodes serve as placeholders and allow for consistent handling of both the head and tail pointers, ensuring that they always point to valid nodes. By using dummy nodes, the implementation of operations such as inserting or removing elements at the beginning or end becomes more streamlined, as special cases and edge conditions are eliminated. Dummy nodes in doubly linked lists provide a standardized structure for traversal and manipulation, enhancing code clarity and reducing complexity. They make it easier to handle boundary cases and simplify the implementation of algorithms and data structures that rely on doubly linked lists.

Dummy Nodes Example

Below is an example of dummy nodes in C++:

```cpp
#include <iostream>

class Node {
public:
    int data;
    Node* prev;
    Node* next;

    Node(int value) : data(value), prev(nullptr), next(nullptr) {}
};

class DoublyLinkedList {
private:
    Node* dummyHead;
    Node* dummyTail;

public:
    DoublyLinkedList() {
        dummyHead = new Node(-1);  // Dummy head with data -1
        dummyTail = new Node(-1);  // Dummy tail with data -1

        dummyHead->next = dummyTail;
        dummyTail->prev = dummyHead;
    }

    void insert(int value) {
        Node* newNode = new Node(value);

        newNode->next = dummyTail;
        newNode->prev = dummyTail->prev;
        dummyTail->prev->next = newNode;
        dummyTail->prev = newNode;
    }

    void display() {
        Node* current = dummyHead->next;
        while (current != dummyTail) {
            std::cout << current->data << " ";
            current = current->next;
        }
        std::cout << std::endl;
    }
};

int main() {
    DoublyLinkedList myList;

    myList.insert(5);
    myList.insert(10);
    myList.insert(2);
    myList.insert(8);
    myList.insert(3);

    std::cout << "Doubly linked list: ";
    myList.display();  // Output: 5 10 2 8 3

    return 0;
}

```

The 'DoublyLinkedList' class represents the doubly linked list data structure. It has two dummy nodes: 'dummyHead' and 'dummyTail'. These dummy nodes act as placeholders and ensure that the head and tail pointers always point to valid nodes. The 'dummyHead' is initially connected to the 'dummyTail' to indicate an empty list.

The 'insert' function inserts a new node with the specified value at the end of the list. It creates a new node, adjusts the pointers of the previous and next nodes accordingly, and connects the new node in between. The 'display' function traverses the list starting from the dummy head and prints the data stored in each node until it reaches the dummy tail, excluding the dummy nodes.

In the 'main()' function, we create an instance of 'DoublyLinkedList' named 'myList' and insert five elements with values 5, 10, 2, 8, and 3, respectively. We then call the 'display' function to print the contents of the doubly linked list. The output demonstrates the successful insertion and display of the elements, showcasing the use of dummy nodes in a doubly linked list.

Section 5.13 - Linked Lists: Recursion

Forward Traversal

Forward traversal in recursion of linked lists refers to the process of traversing a linked list recursively in the forward direction, starting from the head node and visiting each node until reaching the end of the list. In this approach, a recursive function is called on each node, which performs some operation or accesses the data of the current node before recursively calling itself on the next node. This continues until the end of the list is reached, typically indicated by a base case that terminates the recursion. Forward traversal in recursion provides a concise and elegant way to iterate over linked lists, simplifying the code and reducing the need for explicit loop structures. It allows for efficient processing of each node in a linked list and enables various operations, such as printing, searching, or modifying the list, to be implemented recursively.

Reverse Traversal

Reverse traversal in recursion of linked lists involves traversing a linked list in the reverse direction using a recursive approach. Instead of starting from the head node and moving towards the tail, as in forward traversal, reverse traversal begins from the tail node and moves towards the head. This is achieved by recursively calling the reverse traversal function on the next node before performing any operation or accessing the data of the current node. The recursion continues until the head node is reached, which acts as the base case to terminate the traversal. Reverse traversal in recursion allows for efficient processing of nodes in reverse order, enabling operations such as printing in reverse, reversing the list itself, or performing any other operations that require accessing nodes in the opposite direction. It offers a concise and elegant solution to handle reverse traversal scenarios, leveraging the power of recursion in linked list manipulation.

Searching

Searching in linked lists using recursion involves recursively traversing the list to find a specific element or perform a search operation. The recursive search function is typically called on each node, comparing the node's data with the target element. If a match is found, the search terminates, and the appropriate result or node reference is returned. If the current node does not match the target element, the recursion continues by calling the search function on the next node until either a match is found or the end of the list is reached, typically indicated by a base case. Recursive searching in linked lists provides a concise and elegant solution, leveraging the power of recursion to handle search operations efficiently. It allows for flexibility in implementing different search criteria and can be easily extended to handle various search algorithms, such as linear search or binary search in sorted lists, by appropriately adjusting the recursion logic and base case conditions.

Linked List Recursion Example

Below is an example of recursion in linked lists in C++:

```cpp
#include <iostream>

class Node {
public:
    int data;
    Node* next;

    Node(int value) : data(value), next(nullptr) {}
};

class LinkedList {
private:
    Node* head;

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

    void insert(int value) {
        Node* newNode = new Node(value);

        if (head == nullptr) {
            head = newNode;
        } else {
            Node* current = head;
            while (current->next != nullptr) {
                current = current->next;
            }
            current->next = newNode;
        }
    }

    bool searchRecursive(Node* node, int target) {
        if (node == nullptr) {
            return false;  // Base case: end of list reached, element not found
        }
```

```
36
37          if (node->data == target) {
38              return true;  // Base case: element found
39          }
40
41          return searchRecursive(node->next, target);  // Recursive call on next
42      }
43
44      bool search(int target) {
45          return searchRecursive(head, target);
46      }
47  };
48
49  int main() {
50      LinkedList myList;
51      myList.insert(5);
52      myList.insert(10);
53      myList.insert(2);
54      myList.insert(8);
55      myList.insert(3);
56
57      int target = 10;
58      bool found = myList.search(target);
59      if (found) {
60          std::cout << "Element " << target << " found in the linked list." << std::endl;
61      } else {
62          std::cout << "Element " << target << " not found in the linked list." << std::endl;
63      }
64
65      return 0;
66  }
67
```

The 'searchRecursive' function takes two parameters: the current node and the target element to search for. It follows a recursive approach to traverse the list and compare the data of each node with the target. If the target is found, it returns 'true', and if the end of the list is reached without finding the target, it returns 'false'.

The 'search' function acts as a wrapper that initiates the search operation by calling 'searchRecursive' on the 'head' node. In the 'main()' function, we create an instance of 'LinkedList' named 'myList' and insert several elements. We then perform a search for the target element, which is set to 10 in this example. The result is printed based on whether the element is found or not.

The example demonstrates how recursion can be used to search for an element in a linked list efficiently. It allows for a concise and elegant solution, leveraging the recursive nature of the problem and simplifying the search operation.

Section 5.14 - Circular Lists

A circular linked list is a type of linked list where the last node of the list points back to the first node, forming a circular structure. Unlike a traditional linked list where the last node points to null, a circular linked list provides a seamless loop through its nodes. This allows for continuous traversal of the list starting from any node, as each node has a reference to the next node, and the last node points back to the first node. Circular linked lists can be useful in scenarios where cyclic behavior or circular access is required, such as round-robin scheduling algorithms or implementing circular buffers. They offer efficient operations for insertion and deletion at the beginning or end of the list and enable convenient looping or rotation through the list elements.

Section 5.15 - Array-Based Lists

An array-based list, also known as a dynamic array or resizable array, is a data structure that stores elements in a contiguous block of memory. It simulates the behavior of a list or array by dynamically allocating memory and resizing as needed. The elements are stored in a fixed-size array, and when the capacity of the array is reached, a new larger array is created and the elements are copied to the new array. This allows for efficient random access to elements and provides constant-time complexity for accessing elements by index. Array-based lists offer flexibility in adding and removing elements at the end of the list, and they provide a more memory-efficient alternative to

linked lists in many scenarios. However, they may require occasional resizing operations, which can be costly in terms of time and memory.

Section 5.16 - Stack Abstract Data Type (ADT)

The stack abstract data type (ADT) is a collection of elements that follows the Last-In-First-Out (LIFO) principle. It models a real-life stack, where elements are added and removed from the top of the stack. The key operations of a stack include push (adding an element to the top of the stack) and pop (removing the top element from the stack). Additionally, a stack typically provides a peek operation to access the top element without removing it and a isEmpty operation to check if the stack is empty. Stacks are widely used in various applications, such as function call stacks, expression evaluation, and backtracking algorithms. They provide efficient access to the most recently added elements and support a compact and intuitive way to manage data in a last-in-first-out fashion.

Section 5.17 - Stacks Using Linked Lists

Stacks implemented using linked lists are a common way to represent the stack abstract data type (ADT). In this implementation, a linked list is used to store the elements of the stack, with each node representing an element. The top of the stack is represented by the head node of the linked list. When an element is pushed onto the stack, a new node is created and inserted at the beginning of the linked list, becoming the new head node. Similarly, when an element is popped from the stack, the head node is removed, and the next node becomes the new head node. This implementation allows for dynamic memory allocation and efficient push and pop operations, as they can be performed in constant time. Linked list-based stacks provide flexibility in terms of the size of the stack and can handle a variable number of elements. They are particularly useful in scenarios where the stack size is unpredictable or may change dynamically during runtime.

Section 5.18 - Array-Based Stacks

Array-based stacks are a common implementation of the stack abstract data type (ADT) using arrays. In this implementation, an array is used to store the elements of the stack, with a variable indicating the top index of the stack. Elements are pushed onto the stack by incrementing the top index and inserting the element at that position in the array. Similarly, elements are popped from the stack by accessing the element at the top index and decrementing the top index. Array-based stacks provide efficient constant-time complexity for push and pop operations, as accessing elements by index in an array is a constant-time operation. However, they have a fixed capacity determined by the size of the underlying array, and resizing the array can be costly. Array-based stacks are well-suited for scenarios where the maximum number of elements in the stack is known or can be easily determined in advance, providing a more memory-efficient option compared to linked list-based implementations.

Array-Based Stack Example

Below is an example of an array-based stack in C++:

```cpp
#include <iostream>

const int MAX_SIZE = 5;

class Stack {
private:
    int top;
    int arr[MAX_SIZE];

public:
    Stack() : top(-1) {}

    bool isEmpty() {
```

```
        return (top == -1);
    }

    bool isFull() {
        return (top == MAX_SIZE - 1);
    }

    void push(int value) {
        if (isFull()) {
            std::cout << "Stack is full. Cannot push element." << std::endl;
            return;
        }
        top++;
        arr[top] = value;
    }

    int pop() {
        if (isEmpty()) {
            std::cout << "Stack is empty. Cannot pop element." << std::endl;
            return -1;
        }
        int popped = arr[top];
        top--;
        return popped;
    }

    int peek() {
        if (isEmpty()) {
            std::cout << "Stack is empty. Cannot peek element." << std::endl;
            return -1;
        }
        return arr[top];
    }
};

int main() {
    Stack myStack;
    myStack.push(5);
    myStack.push(10);
    myStack.push(7);

    std::cout << "Peek: " << myStack.peek() << std::endl;  // Outputs: 7

    int popped = myStack.pop();
    std::cout << "Popped: " << popped << std::endl;  // Outputs: 7

    myStack.push(3);

    std::cout << "Peek: " << myStack.peek() << std::endl;  // Outputs: 3

    return 0;
}

```

In the 'push' function, a value is added to the stack by incrementing 'top' and assigning the value to the corresponding index in the array. The 'pop' function removes the top element by returning its value and decrementing 'top'. The 'peek' function returns the value of the top element without removing it.

In the 'main' function, we create an instance of the 'Stack' class named 'myStack' and perform stack operations. We push three elements onto the stack, then use the 'peek' function to retrieve the top element (7) without removing it. We then pop an element from the stack (7), and push another element (3). Finally, we use 'peek' again to retrieve the new top element (3).

The example demonstrates how an array-based stack can be implemented in C++. The array acts as a container to store the elements of the stack, and the top index keeps track of the current position of the stack. Array-based stacks provide a simple and efficient way to manage elements using arrays, with constant-time complexity for push, pop, and peek operations.

Section 5.19 - Queue Abstract Data Type (ADT)

The queue abstract data type (ADT) represents a collection of elements that follows the First-In-First-Out (FIFO) principle. It models a real-life queue, where elements are inserted at the back and removed from the front. The main operations of a queue include enqueue (adding an element to the back of the queue) and dequeue (removing the front element from the queue). Additionally, a queue typically provides a peek operation to access the front element without removing it and an isEmpty operation to check if the queue is empty. Queues are commonly used in scenarios where the order of elements matters, such as task scheduling, event handling, and breadth-first search algorithms. They offer efficient insertion and removal of elements at the ends and provide a natural way to manage

data in a first-in-first-out fashion.

Section 5.20 - Queues Using Linked Lists

Queues implemented using linked lists are a common way to represent the queue abstract data type (ADT). In this implementation, a linked list is used to store the elements of the queue, with each node representing an element. The front of the queue is represented by the head node of the linked list, and the back of the queue is represented by the tail node. When an element is enqueued, a new node is created and inserted at the end of the linked list, becoming the new tail node. Similarly, when an element is dequeued, the head node is removed, and the next node becomes the new head node. This implementation allows for efficient insertion and removal of elements at both ends of the queue, as well as constant-time complexity for enqueue and dequeue operations. Linked list-based queues provide flexibility in terms of the size of the queue and can handle a variable number of elements. They are particularly useful in scenarios where the size of the queue may change dynamically and where efficient insertion and removal of elements are required.

Section 5.21 - Array-Based Queues

Overview

Array-based queues are a common implementation of the queue abstract data type (ADT) using arrays. In this implementation, an array is used to store the elements of the queue, with two indices representing the front and back of the queue. Elements are enqueued at the back of the queue by incrementing the back index and inserting the element at that position in the array. Similarly, elements are dequeued from the front of the queue by accessing the element at the front index and incrementing the front index. Array-based queues provide efficient constant-time complexity for enqueue and dequeue operations, as accessing elements by index in an array is a constant-time operation. However, they have a fixed capacity determined by the size of the underlying array, and resizing the array can be costly. Array-based queues are well-suited for scenarios where the maximum number of elements in the queue is known or can be easily determined in advance, providing a more memory-efficient option compared to linked list-based implementations. They are commonly used in applications that require a simple and efficient queue structure with a fixed capacity.

Bounded vs. Unbounded Queue

Bounded and unbounded queues refer to two different types of queues based on their capacity and behavior. A bounded queue has a fixed capacity, meaning it can only hold a limited number of elements. Once the queue is full, any attempt to enqueue additional elements will result in an overflow condition. On the other hand, an unbounded queue has no fixed capacity and can dynamically grow to accommodate any number of elements. Enqueuing elements to an unbounded queue is always possible, as it can allocate additional memory to store new elements. However, this dynamic resizing can be resource-intensive and may result in memory allocation failures in extreme cases. The choice between bounded and unbounded queues depends on the specific requirements of an application. Bounded queues are suitable when the maximum number of elements is known and memory usage needs to be controlled. Unbounded queues, on the other hand, are preferable when the number of elements can vary greatly or when memory constraints are not a concern.

Enqueue and Dequeue Operations

The enqueue and dequeue operations are fundamental operations in a queue data structure. Enqueue refers to the process of adding an element to the back (or rear) of the queue, while dequeue refers to removing an element from the front (or head) of the queue. When enqueueing, the new element is inserted after the last element of the queue, and the rear pointer is updated accordingly. Dequeueing involves removing the element at the front of the queue and updating the front pointer to point to the next element. These operations follow the First-In-First-Out (FIFO) principle, ensuring that the element added first will be the first one to be removed. Enqueue and dequeue

operations are typically performed in constant time complexity, making queues an efficient choice for managing elements in a FIFO manner.

Array-Based Queue Example

Below is an example of an array-based queue example in C++:

```cpp
#include <iostream>
#define MAX_SIZE 5

class ArrayQueue {
private:
    int front, rear;
    int queue[MAX_SIZE];

public:
    ArrayQueue() {
        front = -1;
        rear = -1;
    }

    bool isEmpty() {
        return (front == -1 && rear == -1);
    }

    bool isFull() {
        return (rear == MAX_SIZE - 1);
    }

    void enqueue(int item) {
        if (isFull()) {
            std::cout << "Queue is full. Cannot enqueue element." << std::endl;
            return;
        }
        if (isEmpty()) {
            front = 0;
        }
        rear++;
        queue[rear] = item;
        std::cout << item << " enqueued successfully." << std::endl;
    }

    void dequeue() {
        if (isEmpty()) {
            std::cout << "Queue is empty. Cannot dequeue element." << std::endl;
            return;
        }
        std::cout << queue[front] << " dequeued successfully." << std::endl;
        if (front == rear) {
            front = -1;
            rear = -1;
        } else {
            front++;
        }
    }

    void display() {
        if (isEmpty()) {
            std::cout << "Queue is empty." << std::endl;
            return;
        }
        std::cout << "Queue elements: ";
        for (int i = front; i <= rear; i++) {
            std::cout << queue[i] << " ";
        }
        std::cout << std::endl;
    }
};

int main() {
    ArrayQueue queue;
    queue.enqueue(10);
    queue.enqueue(20);
    queue.enqueue(30);
    queue.display();  // Output: Queue elements: 10 20 30
    queue.dequeue();
    queue.display();  // Output: Queue elements: 20 30
    queue.enqueue(40);
    queue.enqueue(50);
    queue.enqueue(60);  // Output: Queue is full. Cannot enqueue element.
    return 0;
}

```

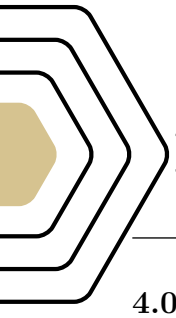
The example demonstrates an array-based queue implementation in C++. The 'ArrayQueue' class maintains a fixed-size array to store the elements of the queue. It provides methods to check if the queue is empty or full, enqueue an element at the rear, dequeue an element from the front, and display the queue elements. The operations are performed in constant time complexity, ensuring efficient handling of elements based on the FIFO principle. The example demonstrates enqueueing and dequeueing elements, as well as handling

cases when the queue is full or empty.

Section 5.22 - Deque Abstract Data Type (ADT)

The deque (double-ended queue) abstract data type (ADT) is a versatile data structure that allows insertion and deletion of elements at both ends. It combines the functionalities of stacks and queues, offering efficient insertion and removal operations at the front and back of the deque. Elements can be added or removed from either end, enabling flexibility in various applications. The deque ADT provides methods such as 'push_front()', 'push_back()', 'pop_front()', 'pop_back()', and 'size()', among others, to manipulate and retrieve elements. Deques can be implemented using arrays or linked lists, with array-based implementations providing constant-time complexity for most operations and linked list implementations providing flexibility in terms of dynamic resizing. The deque ADT is widely used when elements need to be efficiently added or removed from either end, offering a powerful and adaptable data structure for various programming scenarios.

Recursion & Trees



Recursion & Trees

4.0.1 Activities

The following are the activities that are planned for Week 4 of this course.

- Reading Quiz(s) from last week are due on Monday.
- Assignment-2 (Linked List) is due Tuesday.
- Read the zyBook chapter(s) assigned and complete the reading quiz(s) by next Monday.
- Watch week videos Recursion and the Tree data structure.
- Access the GitHub Classroom to get your Assignment-3 repository (assignment due in two weeks).

4.0.2 Lectures

Here are the lectures that can be found for this week:

- Recursion
- Trees
- Binary Search Trees
- Tree Traversal
- BST Operations
- BST Remove Operation

4.0.3 Programming Assignment

The programming assignment for Week 4 is:

- Programming Assignment 3 - Binary Search Tree

4.0.4 Chapter Summary

The first chapter of this week is **Chapter 6: Recursion**.

Section 6.1 - Recursion: Introduction

Recursion is a powerful concept in object-oriented programming (OOP) that allows a function or method to call itself during its execution. It involves breaking down a complex problem into smaller, similar subproblems and solving them iteratively by invoking the same function within itself. Recursion relies on the concept of a base case, which defines the termination condition to stop the recursive calls. Each recursive call works on a smaller subset of the original problem until the base case is reached, at which point the recursion "unwinds" and the results are combined to solve the original problem. Recursion is particularly useful for solving problems that exhibit a recursive structure, such as traversing hierarchical data structures, generating permutations or combinations, and implementing algorithms like quicksort or binary search. It offers an elegant and concise way to solve complex problems and can often result in more readable and maintainable code when used appropriately.

Recursion Example

Below is an example of recursion in C++:

```cpp
#include <iostream>

int factorial(int n) {
// Base case: factorial of 0 is 1
if (n == 0) {
    return 1;
}
// Recursive case: calculate factorial of n by
// multiplying it with factorial of n-1
return n * factorial(n - 1);
}

int main() {
    int number = 5;
    int result = factorial(number);
    std::cout << "Factorial of " << number << " is: " << result << std::endl;
    return 0;
}

```

The example demonstrates the use of recursion to calculate the factorial of a number. The 'factorial' function takes an integer 'n' as input and recursively calculates the factorial by multiplying 'n' with the factorial of 'n-1'. The base case is defined as 'n == 0', where the function returns 1, indicating the factorial of 0. The recursive case performs successive multiplications until the base case is reached. In the main function, the factorial of the number 5 is calculated using the 'factorial' function, and the result is printed. Recursion simplifies the factorial calculation by breaking it down into smaller subproblems until the base case is encountered, offering an elegant and concise solution.

Section 6.2 - Recursive Functions

Recursive functions in C++ are functions that call themselves during their execution. They allow the solving of complex problems by breaking them down into smaller, simpler subproblems. A recursive function consists of two components: the base case, which defines the termination condition to stop the recursion, and the recursive case, where the function calls itself with a modified input. Each recursive call reduces the problem size, bringing it closer to the base case. Recursive functions are particularly useful for solving problems with a recursive structure, such as traversing trees or linked lists, generating permutations, or implementing sorting algorithms like merge sort. Proper design and termination conditions are crucial to ensure the recursion does not lead to infinite loops. Recursive functions offer a concise and elegant way to solve complex problems by leveraging the power of self-reference and the ability to decompose problems into smaller, manageable parts.

Recursive Function Example

Below is an example of a recursive function in C++:

```cpp
#include <iostream>

int recursiveSum(int n) {
    // Base case: if n is 1, return 1
    if (n == 1) {
        return 1;
    }
    // Recursive case: calculate sum of n and the sum of integers from 1 to n-1
    return n + recursiveSum(n - 1);
}

int main() {
    int number = 5;
    int result = recursiveSum(number);
    std::cout << "The sum of integers from 1 to " << number
    << " is: " << result << std::endl;
    return 0;
}

```

The example demonstrates the use of recursion to calculate the sum of integers from 1 to a given number. The 'recursiveSum' function takes an integer 'n' as input and recursively calculates the sum by adding 'n' to the sum of integers from 1 to 'n-1'. The base case is defined as 'n == 1', where the function returns 1. The recursive case performs successive additions until the base case is reached. In the main function, the sum of integers from 1 to the number 5 is calculated using the 'recursiveSum' function, and the result is printed. Recursive functions provide a concise and elegant solution for problems that can be decomposed into smaller subproblems, allowing for a natural and intuitive implementation.

Section 6.3 - Recursive Algorithm: Search

Recursive Search

A recursive search algorithm is a technique that uses recursion to search for a specific element or value in a data structure. It involves breaking down the search problem into smaller subproblems and recursively searching through each subproblem until the target element is found or until the search space is exhausted. The algorithm typically has a base case that defines the termination condition, where it either finds the target element or determines that it does not exist in the data structure. In the recursive case, the algorithm divides the search space into smaller parts and recursively applies the search algorithm to each part. Recursive search algorithms are commonly used in tree and graph traversals, as well as in other data structures. They offer a concise and elegant approach to solving search problems by leveraging the power of recursion and decomposition of the search space.

Recursive Search Function

A recursive search function is an algorithm that uses recursion to search for a specific element or value within a data structure. The function divides the search space into smaller subproblems and recursively applies the search function to each subproblem until the target element is found or the search space is exhausted. It typically includes a base case that defines the termination condition, where the function either finds the target element or determines that it does not exist. In the recursive case, the function splits the search space into smaller parts and recursively invokes itself on each part. Recursive search functions are commonly used in various scenarios, such as searching elements in trees, linked lists, or arrays. They provide an elegant and intuitive way to solve search problems by decomposing the search space and leveraging the power of recursion to explore and locate the desired element efficiently.

Recursively Searching a Sorted List

Recursively searching a sorted list involves using a divide-and-conquer approach to find a target element efficiently. The algorithm starts by comparing the target element with the middle element of the list. If they match, the search is successful. If the target is smaller, the algorithm recursively searches the left half of the list. If the target is larger, the algorithm recursively searches the right half. This process continues until the target element is found or the search space is reduced to zero. By repeatedly dividing the search space in half, the algorithm eliminates large

portions of the list at each step, making it highly efficient for large sorted lists. Recursive searching of sorted lists is a commonly used technique, providing a logarithmic time complexity of $\mathcal{O}\log(n)$ and offering an efficient way to find elements within sorted data structures.

Recursive Search Algorithm Example

Below is an example of using a recursive search algorithm in C++:

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

int binarySearch(const std::vector<int>& sortedList, int target,
                 int left, int right) {
    if (left > right) {
        return -1; // Target element not found
    }

    int mid = (left + right) / 2;

    if (sortedList[mid] == target) {
        return mid; // Target element found
    } else if (sortedList[mid] > target) {
        // Recursively search the left half
        return binarySearch(sortedList, target, left, mid - 1);
    } else {
        // Recursively search the right half
        return binarySearch(sortedList, target, mid + 1, right);
    }
}

int main() {
    std::vector<int> sortedList = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int target = 7;
    int index = binarySearch(sortedList, target, 0, sortedList.size() - 1);

    if (index != -1) {
        std::cout << "Target element found at index: " << index << std::endl;
    } else {
        std::cout << "Target element not found." << std::endl;
    }

    return 0;
}

```

The example demonstrates recursive searching in a sorted list using the binary search algorithm. The 'binarySearch' function takes a sorted list, target element, and the indices of the left and right boundaries. It recursively divides the search space in half by comparing the target element with the middle element of the current range. If the target matches the middle element, the function returns the index. Otherwise, it recursively searches the left or right half of the range based on the comparison result. The algorithm continues until the target element is found or the search space is reduced to zero. In the main function, a sorted list is defined, and the 'binarySearch' function is called to search for the target element. If the target is found, the index is printed; otherwise, a message is displayed indicating that the target element is not found. Recursive searching in a sorted list using the binary search algorithm provides an efficient way to locate elements by repeatedly dividing the search space in half, resulting in a logarithmic time complexity of $\mathcal{O}\log(n)$ and making it suitable for large sorted lists.

Section 6.4 - Adding Output Statements for Debugging

Adding output statements for debugging is a common technique used to track the execution flow and identify issues within a program. By strategically placing print statements or logging statements at key points in the code, developers can observe the values of variables, check the program's state, and trace the control flow during runtime. These output statements can help diagnose problems, verify the correctness of algorithms or conditions, and provide insights into the program's behavior. Debugging output statements are especially useful when dealing with complex logic, loops, or conditionals, as they allow developers to observe intermediate results and identify any unexpected or erroneous behavior. By selectively adding output statements and analyzing the output, developers can gain a better understanding of the program's execution and effectively debug issues, ultimately leading to more robust and reliable software.

Output Statements Example

Below is an example of adding output statements for recursion in C++:

```cpp
#include <iostream>

int factorial(int n) {
    std::cout << "Calculating factorial of: " << n << std::endl;

    if (n == 0) {
        std::cout << "Base case reached: n = 0" << std::endl;
        return 1;
    }

    int result = n * factorial(n - 1);

    std::cout << "Factorial of " << n << " is: " << result << std::endl;

    return result;
}

int main() {
    int number = 5;
    int result = factorial(number);

    std::cout << "Final result: " << result << std::endl;

    return 0;
}

```

The example demonstrates adding output statements for debugging purposes in a recursive function that calculates the factorial of a number. Within the 'factorial' function, output statements are strategically placed to track the execution flow. The statements print the current value of 'n' before each recursive call, indicating the progress of the calculation. When the base case is reached (i.e., 'n' becomes zero), a message is printed to indicate the termination of recursion. After each recursive call, the calculated factorial value for the current 'n' is displayed. In the 'main' function, the 'factorial' function is called with the number 5, and the final result is printed. By observing the output, developers can trace the recursive calls, check the intermediate values of 'n', and verify the correctness of the factorial calculation. Adding such output statements for debugging aids in understanding the behavior of recursive functions, identifying any issues or unexpected behavior, and effectively diagnosing problems.

Section 6.5 - Creating a Recursive Function

Creating a recursive function involves defining a function that calls itself within its own body. Recursive functions are useful when solving problems that can be divided into smaller subproblems of the same nature. The function typically includes a base case, which specifies when the recursion should terminate, and a recursive case, which defines the recursive call(s) to solve the problem incrementally. Recursive functions allow for elegant and concise solutions to certain problems, leveraging the power of self-referential calls to break down complex tasks into simpler ones. However, it is important to ensure that recursive functions have well-defined termination conditions to avoid infinite recursion and unnecessary resource consumption.

Creating a Recusive Function Example

Below is an example of creating a recursive function in C++:

```cpp
#include <iostream>

int fibonacci(int n) {
    // Base cases
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;

    // Recursive case
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n = 6;
    int result = fibonacci(n);
    std::cout << "The " << n << "th Fibonacci number is: "
```

```
        << result << std::endl;

        return 0;
    }

```

The example demonstrates the creation of a recursive function to calculate the nth Fibonacci number. The 'fibonacci' function takes an integer 'n' as input and uses recursive calls to compute the Fibonacci number at position 'n'. The base cases are defined for 'n' equal to 0 and 1, where the function returns the respective Fibonacci numbers. In the recursive case, the function calls itself with 'n - 1' and 'n - 2' to calculate the previous two Fibonacci numbers and adds them together to determine the current Fibonacci number. In the 'main' function, the 'fibonacci' function is called with 'n = 6', and the result is printed. The recursive nature of the function allows for a concise implementation to calculate Fibonacci numbers, where each number is derived by summing the two previous numbers. Recursive functions are particularly useful in scenarios where a problem can be broken down into smaller instances of the same problem, leading to a more elegant and readable code solution.

Section 6.6 - Recursive Math Functions

Recursive math functions are functions that utilize recursion to solve mathematical problems. These functions often involve breaking down a complex mathematical problem into smaller subproblems of the same nature, solving each subproblem recursively, and combining the results to obtain the final solution. Recursive math functions are commonly used to solve problems such as computing factorials, calculating Fibonacci numbers, exponentiation, or finding combinations and permutations. By leveraging the power of recursion, these functions can provide elegant and concise solutions to mathematical problems, reducing the complexity of the code and improving readability. However, it is important to define appropriate base cases and ensure that the recursive calls converge towards the base cases to prevent infinite recursion and ensure termination.

Recursive Math Function Example

Below is an example of recursive math functions in C++:

```
    #include <iostream>

    int factorial(int n) {
        // Base case: factorial of 0 is 1
        if (n == 0)
            return 1;

        // Recursive case: multiply n by factorial of (n - 1)
        return n * factorial(n - 1);
    }

    int main() {
        int number = 5;
        int result = factorial(number);
        std::cout << "The factorial of " << number << " is: "
        << result << std::endl;

        return 0;
    }

```

The example demonstrates the use of a recursive math function to calculate the factorial of a given number. The 'factorial' function takes an integer 'n' as input and recursively multiplies 'n' by the factorial of '(n - 1)' until the base case of 'n' equal to 0 is reached. The base case specifies that the factorial of 0 is 1. By making recursive calls and breaking down the problem into smaller subproblems, the function calculates the factorial of the given number. In the 'main' function, the 'factorial' function is called with 'number = 5', and the result is printed. Recursive math functions provide an elegant and concise way to solve mathematical problems by decomposing them into smaller, manageable tasks. They allow for a more intuitive representation of the problem and often result in more readable code.

Section 6.7 - Recursive Exploration of all Possibilities

Exploring all possibilities using recursion involves designing a recursive function to systematically generate and explore all possible combinations, permutations, or configurations of a given problem. The recursive function typically explores different choices or options at each step and recursively explores all subsequent possibilities until a base case is reached. The base case defines when the recursion should terminate, and the function returns or processes the final result. By exhaustively exploring all possibilities through recursion, we can systematically analyze and solve problems that involve generating or examining all potential outcomes. Recursive exploration of possibilities is particularly useful in tasks such as generating permutations, solving combinatorial problems, backtracking, or searching through a decision tree. However, it is essential to design the recursive function with care, defining appropriate base cases and ensuring that the recursive calls converge towards the base cases to avoid infinite recursion and unnecessary computation.

Recursive Exploration Example

Below is an example of recursive exploration in C++:

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

void generateSubsets(const std::vector<int>& set,
                     std::vector<int>& currentSubset, int index) {
    // Base case: when we reach the end of the set
    if (index == set.size()) {
        // Process the current subset
        for (int num : currentSubset) {
            std::cout << num << " ";
        }
        std::cout << std::endl;
        return;
    }

    // Recursive case: explore two possibilities
    // - include the current element or exclude it
    // Include the current element
    currentSubset.push_back(set[index]);
    generateSubsets(set, currentSubset, index + 1);

    // Exclude the current element
    currentSubset.pop_back();
    generateSubsets(set, currentSubset, index + 1);
}

int main() {
    std::vector<int> set = {1, 2, 3};
    std::vector<int> currentSubset;
    generateSubsets(set, currentSubset, 0);

    return 0;
}

```

The example demonstrates how to use recursion to explore all possible subsets of a given set. The 'generateSubsets' function takes the set, a vector to store the current subset, and the current index as parameters. The base case is when the index reaches the end of the set, where the current subset is processed and printed. In the recursive case, the function explores two possibilities at each step: including the current element in the subset or excluding it. By making recursive calls with updated parameters, the function generates all possible subsets by systematically exploring different combinations of elements. In the 'main' function, the 'generateSubsets' function is called with a set of {1, 2, 3}, and all the possible subsets are printed. Recursive exploration of all possibilities allows us to examine and generate various configurations or combinations efficiently.

Section 6.8 - Stack Overflow

Stack overflow in recursion occurs when the recursive function consumes all available space on the call stack, leading to a runtime error. This typically happens when the recursion does not reach a base case or termination condition, causing an infinite recursion loop. As each recursive function call pushes a new frame onto the call stack, repeated recursive calls without proper termination can exhaust the available stack space. This results in a stack overflow error, where the program is unable to allocate more memory on the stack to accommodate additional

function calls. To prevent stack overflow, it is crucial to ensure that recursive functions have well-defined base cases and termination conditions that allow the recursion to stop. Additionally, tail recursion and iterative approaches can be considered to optimize recursive algorithms and avoid stack overflow issues.

Stack Overflow Example

Below is an example of stack overflow in C++:

```cpp
#include <iostream>

void recursiveFunction() {
    recursiveFunction();  // Recursive call without a base case
}

int main() {
    recursiveFunction();

    return 0;
}

```

In this example, the 'recursiveFunction' is called without a base case or termination condition. As a result, the function enters into an infinite recursion loop, continuously making recursive calls without any stopping condition. Eventually, the recursive calls consume all available space on the call stack, leading to a stack overflow error. When executed, the program encounters a runtime error and terminates due to the stack overflow caused by the infinite recursion. It serves as a reminder of the importance of defining proper base cases or termination conditions in recursive functions to avoid stack overflow errors and ensure the termination of the recursion.

Section 6.9 - Recursively Output Permutations

Recursively outputting permutations involves generating and printing all possible arrangements of a given set of elements using a recursive approach. The algorithm works by swapping elements in the set, fixing one element at a time, and recursively permuting the remaining elements. At each step, the algorithm explores different choices by swapping elements and proceeds with recursive calls until a base case is reached. The base case occurs when only one element remains, indicating that a permutation has been formed. The algorithm then outputs the current permutation. By systematically generating permutations through recursion, all possible arrangements of the elements are explored and printed. However, it is essential to handle duplicate elements and ensure proper termination of the recursion to avoid unnecessary computations and infinite recursion loops.

Recursively Output Permutations Example

Below is an example of recursively outputting permutations in C++:

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

    void generatePermutations(std::vector<int>& nums, int index) {
        // Base case: when all elements have been fixed
        if (index == nums.size() - 1) {
            for (int num : nums) {
                std::cout << num << " ";
            }
            std::cout << std::endl;
            return;
        }

        // Recursive case: generate permutations by fixing one element at a time
        for (int i = index; i < nums.size(); ++i) {
            std::swap(nums[index], nums[i]);
            generatePermutations(nums, index + 1);
            std::swap(nums[index], nums[i]);  // Restore the original order
        }
    }

    int main() {
        std::vector<int> nums = {1, 2, 3};
        generatePermutations(nums, 0);

        return 0;
    }

```

This example demonstrates how to recursively output permutations of a given set of numbers. The 'generatePermutations' function takes a vector of numbers and the current index as parameters. In the base case, when the index reaches the last element, the function outputs the current permutation. In the recursive case, the function swaps the current element with all subsequent elements, fixes it, and recursively generates permutations for the remaining elements. By systematically swapping elements and making recursive calls, the function generates all possible permutations of the given set. In the 'main' function, the 'generatePermutations' function is called with a set of {1, 2, 3}, and all the permutations are printed. The example demonstrates how recursion can be used to generate and output permutations efficiently by exploring different choices and combinations of elements.

Section 6.10 - Recursive Definitions

Recursive Algorithms

Recursive algorithms are algorithms that solve a problem by breaking it down into smaller, simpler instances of the same problem. They employ the concept of self-reference, where a function calls itself to solve a smaller subproblem, eventually reaching a base case that does not require further recursion. Recursive algorithms follow the divide-and-conquer approach, where a larger problem is divided into smaller, more manageable subproblems, and the results are combined to obtain the final solution. These algorithms are often elegant and concise, as they leverage the power of recursion to solve complex problems by solving smaller versions of themselves. However, it is essential to ensure that recursive algorithms have well-defined base cases, proper termination conditions, and avoid unnecessary recomputation to prevent infinite recursion and optimize performance.

Recursive Functions

Recursive functions are functions that call themselves within their own definition to solve a problem by breaking it down into smaller, simpler instances of the same problem. They allow for the repetition of a specific set of instructions until a certain condition is met, known as the base case. Recursive functions leverage the concept of self-reference to solve complex problems by reducing them to simpler subproblems. Each recursive call operates on a smaller portion of the problem until reaching the base case, where the recursion stops and the results are combined to obtain the final solution. Recursive functions are widely used when dealing with problems that exhibit a recursive structure, as they provide an elegant and intuitive way to tackle such problems. However, it is crucial to define proper base cases and ensure termination conditions to prevent infinite recursion and ensure the correctness and efficiency of recursive functions.

Section 6.11 - Recursive Algorithms

Popular recursive algorithms include the factorial function, which calculates the product of all positive integers up to a given number by recursively multiplying the number with the factorial of its predecessor until reaching the base case of 0 or 1. The Fibonacci sequence is another well-known recursive algorithm, where each number in the sequence is the sum of the two preceding numbers, with the base cases of 0 and 1. The recursive implementation of quicksort divides the input array into subarrays based on a chosen pivot, recursively sorting the subarrays until reaching subarrays of size 1 or 0. The binary search algorithm uses recursion to efficiently search for a target element in a sorted array by recursively dividing the array into halves and discarding the half that does not contain the target. These are just a few examples of recursive algorithms, which demonstrate the power and versatility of recursion in solving various computational problems by breaking them down into simpler instances.

The second chapter of this week is Chapter 7: Trees.

Section 7.1 - Binary Trees

Binary Tree Basics

Binary trees are hierarchical data structures consisting of nodes connected by edges. Each node in a binary tree can have at most two child nodes, referred to as the left child and the right child. The topmost node in the tree is called the root. Binary trees exhibit a recursive structure, as each node can be viewed as the root of its own subtree. The left subtree of a node contains nodes with smaller values, while the right subtree contains nodes with greater values. Binary trees can be used to represent hierarchical relationships and are widely used in various applications such as searching, sorting, and data representation. Traversing a binary tree involves visiting each node in a specific order, such as preorder, inorder, or postorder traversal. The structure and properties of binary trees play a fundamental role in understanding more advanced data structures and algorithms.

Depth, Level, & Height

In the context of binary trees, depth refers to the distance between a node and the root, counting the number of edges in the path. The root node has a depth of 0, and the depth increases by 1 as we move towards the leaves. Level, on the other hand, refers to the depth of a node plus one. So, the level of the root node is 1, and the level increases by 1 as we move down the tree. Height, also known as the depth of the tree, is the maximum depth of any node in the tree. It represents the length of the longest path from the root to a leaf node. Height is a measure of the overall size of the binary tree and provides insights into the tree's balance and performance characteristics. Understanding and computing the depth, level, and height of binary trees are essential for analyzing and working with these data structures efficiently.

Special Types of Binary Trees

Special types of binary trees include full, complete, and perfect trees. A full binary tree is a tree in which every node has either zero or two children, meaning there are no nodes with only one child. Each level of a full binary tree is completely filled, except possibly for the last level, which is filled from left to right. A complete binary tree is a binary tree in which all levels, except possibly the last one, are completely filled, and the nodes on the last level are filled from left to right. It may have nodes with only one child, but they must be towards the left side of the last level. A perfect binary tree is a binary tree in which all levels are completely filled with the maximum number of nodes possible, which is $2^h - 1$, where h is the height of the tree. In a perfect binary tree, every internal node has two children. Understanding these special types of binary trees helps in analyzing and designing efficient algorithms that exploit their structural properties.

Binary Trees Example

Below is an example of binary trees in C++:

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

// Node class for binary tree
class Node {
public:
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

// Function to check if a binary tree is complete
bool isCompleteBinaryTree(Node* root) {
    if (root == nullptr)
        return true;

    queue<Node*> q;
    bool flag = false;

    // Perform level order traversal
    q.push(root);
    while (!q.empty()) {
        Node* curr = q.front();
        q.pop();

```

```
            // Check if the node has any missing child
            if (curr->left) {
                if (flag)
                    return false;
                q.push(curr->left);
            }
            else
                flag = true;

            if (curr->right) {
                if (flag)
                    return false;
                q.push(curr->right);
            }
            else
                flag = true;
        }

        return true;
    }

    // Main function
    int main() {
        Node* root = new Node(1);
        root->left = new Node(2);
        root->right = new Node(3);
        root->left->left = new Node(4);
        root->left->right = new Node(5);

        if (isCompleteBinaryTree(root))
            cout << "The binary tree is complete.";
        else
            cout << "The binary tree is not complete.";

        return 0;
    }

```

In this example, we have implemented a complete binary tree and provided a function 'isCompleteBinaryTree()' to check if the given binary tree is complete. The function uses a queue-based level order traversal approach to check for missing children in each level. If any node is encountered with a missing child before reaching the last level, the tree is deemed incomplete. The program outputs whether the binary tree is complete or not based on the result of the function.

The example demonstrates the implementation of a complete binary tree in C++ and the usage of a level order traversal-based algorithm to check its completeness. By using a queue to explore the tree level by level, we can determine if any nodes are missing children before reaching the last level. This example showcases how the complete binary tree property can be verified algorithmically, providing a practical application of the concept.

Section 7.2 - Applications of Trees

File Systems

Binary trees find applications in various domains, and one notable area is file systems. File systems organize and manage data stored on storage devices such as hard drives or solid-state drives. They often use a hierarchical structure to represent directories and files, which can be effectively modeled using binary trees. Each node in the tree represents a directory, and its children nodes represent subdirectories or files within that directory. The tree structure allows for efficient navigation, searching, and manipulation of files and directories. Binary trees are particularly useful in file systems for tasks like directory traversal, file organization, file searching, and maintaining the hierarchical relationships between directories and files. They enable efficient file system operations, contributing to the smooth functioning and management of data within a file system.

Binary Space Partitioning

Binary Space Partitioning (BSP) trees find applications in various areas where efficient spatial partitioning and search algorithms are required. One notable application is in computer graphics, particularly in rendering and collision detection algorithms. BSP trees are used to partition a 3D space into regions, dividing it based on planes. This partitioning allows for efficient visibility determination and culling in rendering algorithms, enabling

faster rendering of complex scenes. Additionally, BSP trees are used in collision detection algorithms to efficiently determine whether objects or polygons intersect or collide with each other. By recursively subdividing the space using binary splits, BSP trees provide an effective data structure for spatial partitioning and enable efficient spatial queries in various applications, including computer graphics, robotics, and computational geometry.

Applications of Binary Trees Example

Below is an example of an application of binary trees in C++:

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

// Binary tree node for representing directories or files
struct TreeNode {
    string name;
    bool isDirectory;
    TreeNode* left;
    TreeNode* right;

    TreeNode(string n, bool isDir) {
        name = n;
        isDirectory = isDir;
        left = nullptr;
        right = nullptr;
    }
};

// Function to perform a depth-first traversal of the file system tree
void DFS(TreeNode* node, int level) {
    if (node == nullptr) return;

    cout << string(level * 4, ' ') << "- " << node->name << endl;

    DFS(node->left, level + 1);
    DFS(node->right, level + 1);
}

int main() {
    TreeNode* root = new TreeNode("Root", true);
    TreeNode* dir1 = new TreeNode("Dir1", true);
    TreeNode* dir2 = new TreeNode("Dir2", true);
    TreeNode* file1 = new TreeNode("File1.txt", false);
    TreeNode* file2 = new TreeNode("File2.txt", false);

    root->left = dir1;
    root->right = dir2;
    dir1->left = file1;
    dir2->right = file2;

    DFS(root, 0);

    // Clean up memory
    delete file1;
    delete file2;
    delete dir1;
    delete dir2;
    delete root;

    return 0;
}

```

This example demonstrates the representation of a file system using a binary tree. Each node in the tree represents a directory or a file, and the left and right child pointers represent subdirectories or files within a directory. The 'DFS' function performs a depth-first traversal of the tree, printing the names of the directories and files with appropriate indentation to indicate their levels. The example showcases the hierarchical structure of a file system and how it can be effectively modeled and traversed using a binary tree data structure.

Section 7.3 - Binary Search

Linear Search vs. Binary Search

Binary search is a search algorithm that efficiently finds the position of a target value within a sorted array. It follows a divide-and-conquer approach by repeatedly dividing the search space in half, eliminating one half of the remaining elements in each iteration. In contrast, linear search sequentially checks each element of the array until it either finds the target value or exhausts the entire array. While linear search has a time complexity of $\mathcal{O}(n)$, where n is the number of elements in the array, binary search has a time complexity of $\mathcal{O}(\log(n))$. This significant difference in time complexity makes binary search much faster for large arrays, as it eliminates half of the remaining elements at each step. Binary search is highly efficient for searching in sorted arrays and provides a significant improvement over linear search, especially when dealing with large datasets.

Binary Search Algorithm

The binary search algorithm is a search algorithm used to find the position of a target value within a sorted array. It works by repeatedly dividing the search space in half, comparing the middle element with the target value, and then narrowing down the search to the appropriate half. If the middle element is equal to the target, the search is successful. If the middle element is greater than the target, the search continues in the left half of the array. If the middle element is smaller, the search continues in the right half. This process is repeated until the target value is found or the search space is reduced to zero. Binary search has a time complexity of $\mathcal{O}(\log(n))$, making it highly efficient for searching in sorted arrays. It is a fundamental algorithm in computer science and is widely used in various applications where efficient searching is required.

Binary Search Efficiency

The binary search algorithm is highly efficient in searching for a target value within a sorted array. It follows a divide-and-conquer approach by continuously dividing the search space in half, which significantly reduces the number of elements to be checked in each iteration. As a result, the binary search has a time complexity of $\mathcal{O}(\log(n))$, where n is the number of elements in the array. This logarithmic time complexity means that the binary search can handle large datasets efficiently, even when the number of elements grows exponentially. Compared to linear search, which has a time complexity of $\mathcal{O}(n)$, binary search offers a significant improvement in efficiency, especially for larger arrays. This makes it a preferred choice when dealing with sorted data and the need for fast search operations.

Binary Search Example

Below is an example of a binary search in C++:

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

int binarySearch(int arr[], int target, int low, int high) {
    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == target)
            return mid;

        if (arr[mid] < target)
            low = mid + 1;

        else
            high = mid - 1;
    }

    return -1;
}

int main() {
    int arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
    int target = 23;
    int size = sizeof(arr) / sizeof(arr[0]);

    int index = binarySearch(arr, target, 0, size - 1);

    if (index != -1)
        cout << "Element found at index " << index << endl;
    else
        cout << "Element not found" << endl;

    return 0;
}

```

In this example, the binary search algorithm is used to search for a target element (23) within a sorted array. The 'binarySearch' function implements the binary search logic by repeatedly dividing the search space and updating the low and high indices until the target element is found or the search space is exhausted. The

main function initializes the array and the target element, calculates the size of the array, and calls the 'binarySearch' function. If the target element is found, it prints the index; otherwise, it displays a "not found" message. This example demonstrates the efficiency of the binary search algorithm in finding a target element in a sorted array.

Section 7.4 - Binary Search Tree

Binary Search Trees

A binary search tree (BST) is a binary tree data structure where each node has a key/value and satisfies the following properties: the left child of a node has a key less than the node's key, and the right child has a key greater than the node's key. This ordering property allows for efficient searching, insertion, and deletion operations. Binary search trees are particularly useful for organizing and searching data, as they provide a way to store elements in a sorted order, enabling fast lookup operations. The BST structure allows for efficient average and best-case performance, with a time complexity of $\mathcal{O}(\log(n))$ for these operations, but it can degrade to $\mathcal{O}(n)$ in the worst-case scenario when the tree becomes unbalanced. Therefore, maintaining a balanced BST is essential to ensure optimal performance.

Searching

Searching in a binary search tree (BST) involves comparing the target value with the key of the current node and traversing left or right based on the comparison. The search starts at the root node and continues down the tree until the target value is found or a leaf node is reached. The BST's structure allows for efficient searching by exploiting the ordering property, as it eliminates the need to search through irrelevant subtrees. If the target value is less than the current node's key, the search proceeds to the left subtree; if it is greater, the search proceeds to the right subtree. This recursive process continues until the target value is found or the search reaches a leaf node, indicating that the value is not present in the BST. The time complexity of searching in a balanced BST is $\mathcal{O}(\log(n))$, making it an efficient approach for finding elements in sorted data sets.

BST Search Runtime

The runtime of searching in a binary search tree (BST) is determined by the height of the tree. In a balanced BST, the height is logarithmic in relation to the number of nodes, resulting in an average and best-case runtime of $\mathcal{O}(\log(n))$, where n is the number of nodes in the tree. This logarithmic time complexity is achieved because at each level of the tree, the search space is halved. However, in the worst-case scenario where the BST is highly unbalanced, such as a skewed tree, the height can be equal to the number of nodes, resulting in a linear runtime of $\mathcal{O}(n)$. Therefore, it is crucial to maintain the balance of the BST to ensure efficient searching operations.

Successors & Predecessors

In the context of binary search trees (BSTs), successors and predecessors refer to the nodes that come immediately after or before a given node, respectively, in the sorted order of the tree. The successor of a node is the smallest node with a key greater than the given node's key, while the predecessor is the largest node with a key smaller than the given node's key. Finding successors and predecessors can be useful in various operations, such as in-order traversal, finding the minimum and maximum elements, and performing range queries. The process of finding successors and predecessors typically involves traversing the tree based on the comparison of keys, starting from the root node and moving towards the desired direction until the successor or predecessor is found.

Binary Search Tree Example

Below is an example of binary search trees in C++:

```cpp
#include <iostream>

struct Node {
    int key;
    Node* left;
    Node* right;
};
```

```cpp

Node* createNode(int key) {
    Node* newNode = new Node();
    newNode->key = key;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

Node* insert(Node* root, int key) {
    if (root == nullptr) {
        return createNode(key);
    }
    if (key < root->key) {
        root->left = insert(root->left, key);
    } else if (key > root->key) {
        root->right = insert(root->right, key);
    }
    return root;
}

Node* search(Node* root, int key) {
    if (root == nullptr || root->key == key) {
        return root;
    }
    if (key < root->key) {
        return search(root->left, key);
    }
    return search(root->right, key);
}

void inorderTraversal(Node* root) {
    if (root == nullptr) {
        return;
    }
    inorderTraversal(root->left);
    std::cout << root->key << " ";
    inorderTraversal(root->right);
}

int main() {
    Node* root = nullptr;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    std::cout << "Inorder traversal of the BST: ";
    inorderTraversal(root);

    int key = 40;
    Node* result = search(root, key);
    if (result != nullptr) {
        std::cout << "\nElement " << key << " found in the BST.";
    } else {
        std::cout << "\nElement " << key << " not found in the BST.";
    }

    return 0;
}

```

The provided example demonstrates the implementation of a binary search tree (BST) in C++. The 'Node' struct represents a node in the tree, with a key value, and left and right pointers. The 'createNode' function creates a new node with the given key. The 'insert' function recursively inserts a new node into the BST based on the key value, maintaining the ordering property. The 'search' function searches for a specific key in the BST, returning the corresponding node if found or 'nullptr' if not found. The 'inorderTraversal' function performs an in-order traversal of the BST and prints the keys in sorted order. The 'main' function creates a BST, inserts some elements, performs an in-order traversal, and searches for a specific element.

Section 7.5 - BST Search Algorithm

The binary search tree search algorithm is a method used to find a specific key within a binary search tree (BST). Starting from the root node, the algorithm compares the key with the current node's key. If the keys match, the search is successful, and the node is returned. If the key is less than the current node's key, the algorithm continues

the search in the left subtree. If the key is greater, the search proceeds in the right subtree. This process is repeated recursively until the key is found or until reaching a leaf node, indicating that the key does not exist in the tree. The search algorithm exploits the property of the BST, where all keys in the left subtree are smaller than the current node, and all keys in the right subtree are larger, allowing for efficient search operations.

BST Search Algorithm Example

Below is an example of a binary search tree algorithm in C++:

```cpp
#include <iostream>

// Binary Search Tree Node
struct Node {
    int key;
    Node* left;
    Node* right;

    Node(int value) : key(value), left(nullptr), right(nullptr) {}
};

// Binary Search Tree Search
Node* searchBST(Node* root, int target) {
    // Base case: root is null or target is found
    if (root == nullptr || root->key == target)
        return root;

    // If target is smaller, search in the left subtree
    if (target < root->key)
        return searchBST(root->left, target);

    // If target is larger, search in the right subtree
    return searchBST(root->right, target);
}

int main() {
    // Creating a binary search tree
    Node* root = new Node(5);
    root->left = new Node(3);
    root->right = new Node(8);
    root->left->left = new Node(2);
    root->left->right = new Node(4);
    root->right->left = new Node(7);
    root->right->right = new Node(9);

    // Searching for a key in the binary search tree
    int target = 4;
    Node* result = searchBST(root, target);

    // Displaying the result
    if (result)
        std::cout << "Key " << target << " found in the binary search tree." << std::endl;
    else
        std::cout << "Key " << target << " not found in the binary search tree." << std::endl;

    return 0;
}

```

In this example, we demonstrate the binary search tree search algorithm using a simple binary search tree. The 'searchBST' function takes a root node and a target key as input and performs a recursive search in the binary search tree. Starting from the root, the function compares the target key with the current node's key and continues the search in the left or right subtree based on the key's value. The function returns the node if the key is found or 'nullptr' if it is not present in the tree. Finally, we showcase the usage of the algorithm by searching for a specific key and displaying the result.

Section 7.6 - BST Insert Algorithm

Overview

The binary search tree insert algorithm is used to insert a new node into a binary search tree while maintaining its binary search tree property. The algorithm starts from the root of the tree and compares the key of the new node with the key of the current node. If the new node's key is less than the current node's key, the algorithm moves to the left child of the current node. If the new node's key is greater, the algorithm moves to the right child. This

process continues until an appropriate position for the new node is found (i.e., a null child pointer is encountered). Once the appropriate position is found, a new node is created and connected to the tree. The binary search tree insert algorithm ensures that the resulting tree maintains the binary search tree property, where all nodes in the left subtree are less than the current node, and all nodes in the right subtree are greater.

BST Insert Algorithm Complexity

The algorithm complexity of a binary search tree insert algorithm is typically $\mathcal{O}(\log(n))$ in the average case and $\mathcal{O}(n)$ in the worst case, where n is the number of nodes in the tree. The average case complexity arises from the balanced nature of the binary search tree, where each level approximately halves the search space. However, in the worst case scenario where the tree is skewed, such as when inserting nodes in sorted order, the tree can degenerate into a linked list, resulting in a linear search time. To maintain the balanced nature of the tree and ensure efficient insertions, various techniques like self-balancing binary search trees, such as AVL trees or red-black trees, can be employed. These techniques guarantee a logarithmic height of the tree and provide a more balanced tree structure, resulting in a consistent $\mathcal{O}(\log(n))$ complexity for insertions.

BST Insert Algorithm Example

Below is an example of a binary search tree insert algorithm in C++:

```cpp
#include <iostream>

struct Node {
    int key;
    Node* left;
    Node* right;
};

Node* insert(Node* root, int key) {
    if (root == nullptr) {
        Node* newNode = new Node;
        newNode->key = key;
        newNode->left = nullptr;
        newNode->right = nullptr;
        return newNode;
    }

    if (key < root->key) {
        root->left = insert(root->left, key);
    } else if (key > root->key) {
        root->right = insert(root->right, key);
    }

    return root;
}

void inOrderTraversal(Node* root) {
    if (root != nullptr) {
        inOrderTraversal(root->left);
        std::cout << root->key << " ";
        inOrderTraversal(root->right);
    }
}

int main() {
    Node* root = nullptr;

    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    std::cout << "In-order traversal of the binary search tree: ";
    inOrderTraversal(root);
    std::cout << std::endl;

    return 0;
}

```

This example demonstrates the binary search tree insert algorithm. We define a 'Node' struct representing a node in the tree, which contains a key value, as well as left and right child pointers. The 'insert' function recursively inserts a new node with the given key into the appropriate position in the tree. The 'inOrderTraversal' function performs an in-order traversal of the tree, printing the keys in sorted order. In the 'main' function, we create a binary search tree by inserting several keys, and then perform an in-order traversal to display the keys.

Section 7.7 - BST Remove Algorithm

Overview

The binary search tree remove algorithm is used to delete a node from a binary search tree while maintaining its properties. The algorithm starts by searching for the node to be removed in the tree based on its key value. Once the node is found, there are three cases to consider:

1. If the node has no children, it can be simply deleted from the tree.
2. If the node has only one child, the child node replaces the deleted node in the tree.
3. If the node has two children, the algorithm finds the node's in-order successor or predecessor (the node with the next smallest or next largest key, respectively), swaps their key values, and then recursively removes the successor/predecessor from its original position. This ensures that the binary search tree properties are maintained.

Overall, the binary search tree remove algorithm is a complex process that involves carefully handling different cases to ensure the integrity of the tree. It requires finding the node to be removed, considering its children, and potentially rearranging the tree structure to maintain the binary search tree properties.

BST Remove Algorithm Complexity

The complexity of the binary search tree remove algorithm depends on the height of the tree, which is influenced by the tree's structure and the order of node removal. In the worst-case scenario, where the tree is skewed and has a height of N (resembling a linked list), the time complexity of the remove algorithm is $\mathcal{O}(N)$. However, in a balanced binary search tree, where the height is logarithmic to the number of nodes ($\log(N)$), the removal operation takes $\mathcal{O}(\log(N))$ time. This logarithmic complexity arises from the process of searching for the node to be removed and then potentially rearranging the tree structure. Overall, the complexity of the binary search tree remove algorithm is efficient when the tree is balanced but can degrade to linear time when the tree is highly skewed.

BST Remove Algorithm Example

Below is an example of a binary search tree remove algorithm in C++:

```cpp
#include <iostream>

struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int value) : val(value), left(nullptr), right(nullptr) {}
};

TreeNode* findMin(TreeNode* node) {
    while (node->left != nullptr) {
        node = node->left;
    }
    return node;
}

TreeNode* removeNode(TreeNode* root, int key) {
    if (root == nullptr) {
        return root;
    }
    if (key < root->val) {
        root->left = removeNode(root->left, key);
    } else if (key > root->val) {
        root->right = removeNode(root->right, key);
    } else {
        if (root->left == nullptr) {
            TreeNode* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            TreeNode* temp = root->left;
            delete root;
            return temp;
        }
        TreeNode* temp = findMin(root->right);
        root->val = temp->val;
        root->right = removeNode(root->right, temp->val);
```

```
    }
    return root;
}

void inorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    inorderTraversal(root->left);
    std::cout << root->val << " ";
    inorderTraversal(root->right);
}

int main() {
    TreeNode* root = new TreeNode(8);
    root->left = new TreeNode(3);
    root->right = new TreeNode(10);
    root->left->left = new TreeNode(1);
    root->left->right = new TreeNode(6);
    root->right->right = new TreeNode(14);
    root->left->right->left = new TreeNode(4);
    root->left->right->right = new TreeNode(7);
    root->right->right->left = new TreeNode(13);

    std::cout << "Binary Search Tree before removal: ";
    inorderTraversal(root);
    std::cout << std::endl;

    int key = 6;
    root = removeNode(root, key);

    std::cout << "Binary Search Tree after removing " << key << ": ";
    inorderTraversal(root);
    std::cout << std::endl;

    return 0;
}

```

In this example, we have implemented the binary search tree remove algorithm. The 'removeNode' function takes the root of the tree and a key value to be removed. It recursively searches for the node with the given key and performs the necessary rearrangements to maintain the binary search tree properties. The 'findMin' function helps in finding the minimum value in the right subtree when the node to be removed has both left and right children. Finally, we demonstrate the algorithm by removing a node with the key value of 6 from the binary search tree and printing the updated tree using an inorder traversal.

Section 7.8 - BST Inorder Traversal

Binary search tree inorder traversal is a recursive algorithm used to visit all the nodes of a binary search tree in a specific order. In inorder traversal, we first visit the left subtree, then the root node, and finally the right subtree. This traversal ensures that the nodes are visited in ascending order when the tree represents sorted data. It allows us to retrieve the elements of the binary search tree in sorted order, making it useful for tasks such as printing the elements, checking for sortedness, or creating a sorted array from the tree. The algorithm follows a left-root-right pattern and can be implemented recursively or iteratively.

BST Inorder Traversal Example

Below is an example of a binary search tree inorder traversal algorithm in C++:

```
#include <iostream>

// Binary Search Tree Node
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) : data(value), left(nullptr), right(nullptr) {}
};

// Inorder Traversal
void inorderTraversal(Node* root) {
    if (root == nullptr)
        return;

    inorderTraversal(root->left);
```

```
    std::cout << root->data << " ";
    inorderTraversal(root->right);
}

int main() {
    // Create a sample Binary Search Tree
    Node* root = new Node(4);
    root->left = new Node(2);
    root->right = new Node(6);
    root->left->left = new Node(1);
    root->left->right = new Node(3);
    root->right->left = new Node(5);
    root->right->right = new Node(7);

    // Perform inorder traversal
    std::cout << "Inorder Traversal: ";
    inorderTraversal(root);

    return 0;
}

```

In this example, we create a binary search tree with integer values and perform the inorder traversal. The 'inorderTraversal' function recursively visits the nodes in the left-root-right order and prints their values. The output will be the sorted sequence of elements from the binary search tree. In this case, the inorder traversal will print: 1 2 3 4 5 6 7.

Section 7.9 - BST Height & Insertion Order

The height of a binary search tree refers to the maximum number of edges from the root to any leaf node, influencing the efficiency of operations. A balanced tree with a logarithmic height allows for faster search, insert, and delete operations. However, the insertion order of nodes can impact the tree's balance, with an ordered insertion leading to a skewed structure and increased height. Random or balanced insertion strategies are recommended for achieving optimal performance by maintaining a lower height and a more evenly distributed tree structure.

BST Height & Insertion Order Example

Below is an example of a binary search tree involving height and insertion order in C++:

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

class Node {
public:
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);
    }

    return root;
}

void inorderTraversal(Node* root) {
    if (root == nullptr) {
        return;
    }

    inorderTraversal(root->left);
    cout << root->data << " ";
```

```
    inorderTraversal(root->right);
}

int main() {
    Node* root = nullptr;

    // Insertion order: 5, 2, 8, 1, 3, 6, 9
    root = insert(root, 5);
    root = insert(root, 2);
    root = insert(root, 8);
    root = insert(root, 1);
    root = insert(root, 3);
    root = insert(root, 6);
    root = insert(root, 9);

    cout << "Inorder Traversal: ";
    inorderTraversal(root);

    return 0;
}

```

The example demonstrates the insertion order's impact on the binary search tree structure. In this case, the numbers 5, 2, 8, 1, 3, 6, and 9 are inserted in the given order. The resulting tree structure shows that the nodes are placed according to their values, with smaller values on the left and larger values on the right. The inorder traversal of the tree verifies that the nodes are visited in ascending order: 1, 2, 3, 5, 6, 8, 9. The example highlights how the insertion order influences the resulting binary search tree and affects the order of traversal.

Section 7.10 - BST Parent Node Pointers

In a binary search tree, parent node pointers can be used to efficiently navigate and perform operations on the tree. Each node in the binary search tree contains a pointer to its parent node, allowing for easy traversal up the tree. This enables operations such as finding the parent of a given node, determining the root of the tree, or moving from one node to another level higher. The parent node pointers enhance the efficiency of various tree operations and provide a convenient way to access and manipulate the tree structure.

BST Parent Node Pointers

Below is an example of binary search tree parent node pointers in C++:

```
#include <iostream>

struct Node {
    int data;
    Node* left;
    Node* right;
    Node* parent;
};

class BinarySearchTree {
private:
    Node* root;

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

    void insert(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        newNode->left = nullptr;
        newNode->right = nullptr;
        newNode->parent = nullptr;

        if (root == nullptr) {
            root = newNode;
        } else {
            Node* current = root;
            while (true) {
                if (value < current->data) {
                    if (current->left == nullptr) {
                        current->left = newNode;
                        newNode->parent = current;
                        break;
                    } else {
                        current = current->left;
                    }
```

```
                } else {
                    if (current->right == nullptr) {
                        current->right = newNode;
                        newNode->parent = current;
                        break;
                    } else {
                        current = current->right;
                    }
                }
            }
        }
    }

    // Other functions like search, remove, etc.

    void displayParentPointers() {
        displayParentPointersHelper(root);
    }

private:
    void displayParentPointersHelper(Node* node) {
        if (node != nullptr) {
            std::cout << "Node: " << node->data;
            if (node->parent != nullptr) {
                std::cout << ", Parent: " << node->parent->data;
            }
            std::cout << std::endl;
            displayParentPointersHelper(node->left);
            displayParentPointersHelper(node->right);
        }
    }
};

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

    bst.displayParentPointers();

    return 0;
}

```

In this example, we define a 'Node' struct that contains data, left and right child pointers, and a parent pointer. The 'BinarySearchTree' class uses this 'Node' struct to implement a binary search tree with parent node pointers. The 'insert' function is modified to assign the parent pointer while inserting nodes. The 'displayParentPointers' function is added to demonstrate accessing and displaying the parent node pointers for each node in the tree. When we run the program, it will print the nodes along with their respective parent nodes.

The example demonstrates the use of parent node pointers in a binary search tree implemented in C++. It showcases the insertion of nodes while maintaining the parent pointers, and a function to display the parent pointers of each node. This allows for easy navigation and analysis of the tree's structure.

Section 7.11 - BST: Recursion

BST Recursive Search Algorithm

Binary search tree recursive search algorithms involve recursively traversing the tree to find a specific key or determine if it exists within the tree. The recursive approach begins at the root node and compares the key with the current node's key. If the key matches, the search is successful. If the key is less than the current node's key, the algorithm continues the search in the left subtree. Conversely, if the key is greater, the algorithm proceeds to the right subtree. This process continues until the key is found or the algorithm reaches a leaf node (indicating the key does not exist in the tree). The recursive nature of these algorithms simplifies the search logic by utilizing the recursive function calls to traverse the tree in an efficient manner.

BST Get Parent Algorithm

The binary search tree get parent algorithm is used to retrieve the parent node of a given node in a binary search tree. Starting from the root node, the algorithm compares the key of the current node with the target key. If the key matches, the parent node is found. If the key is less than the current node's key, the algorithm continues the search in the left subtree, while if the key is greater, it proceeds to the right subtree. By keeping track of the parent node during the search process, the algorithm can return the parent node when the target key is found. This algorithm allows efficient retrieval of the parent node of a given node in a binary search tree, facilitating various operations and modifications on the tree structure.

Recursive BST Insertion and Removal

The recursive binary search tree (BST) insertion algorithm is used to insert a new node into a BST while maintaining the BST properties. Starting from the root node, the algorithm compares the value of the new node with the value of the current node. If the value is less, the algorithm recursively inserts the new node in the left subtree. If the value is greater, the algorithm recursively inserts the new node in the right subtree. This process continues until an appropriate position is found for the new node, at which point it is inserted. The recursive BST removal algorithm is used to remove a node from a BST while preserving the BST properties. It follows a similar approach as the insertion algorithm, recursively traversing the tree to find the target node. Once the node is found, the algorithm handles three cases: if the target node has no children, it is simply removed; if the target node has one child, it is replaced by its child; if the target node has two children, it is replaced by its in-order successor or predecessor, and then the successor/predecessor node is recursively removed. These recursive algorithms ensure the proper insertion and removal of nodes in a binary search tree.

Recursive BST Algorithms

Below is an example of a recursive binary search tree algorithm in C++:

```cpp
#include <iostream>

struct Node {
    int data;
    Node* left;
    Node* right;
};

Node* insert(Node* root, int value) {
    if (root == nullptr) {
        Node* newNode = new Node();
        newNode->data = value;
        newNode->left = nullptr;
        newNode->right = nullptr;
        return newNode;
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }

    return root;
}

Node* findMin(Node* node) {
    while (node->left != nullptr) {
        node = node->left;
    }
    return node;
}

Node* remove(Node* root, int value) {
    if (root == nullptr) {
        return nullptr;
    }

    if (value < root->data) {
        root->left = remove(root->left, value);
    } else if (value > root->data) {
        root->right = remove(root->right, value);
    } else {
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        Node* minRight = findMin(root->right);
```

```
            root->data = minRight->data;
            root->right = remove(root->right, minRight->data);
        }

        return root;
    }

    void inorderTraversal(Node* root) {
        if (root != nullptr) {
            inorderTraversal(root->left);
            std::cout << root->data << " ";
            inorderTraversal(root->right);
        }
    }

    int main() {
        Node* root = nullptr;
        root = insert(root, 5);
        insert(root, 3);
        insert(root, 7);
        insert(root, 1);
        insert(root, 4);
        insert(root, 6);
        insert(root, 9);

        std::cout << "Inorder traversal: ";
        inorderTraversal(root);
        std::cout << std::endl;

        root = remove(root, 3);
        root = remove(root, 7);

        std::cout << "Inorder traversal after removal: ";
        inorderTraversal(root);
        std::cout << std::endl;

        return 0;
    }

```

This example demonstrates the recursive insertion and removal algorithms for a binary search tree (BST). The 'insert' function inserts nodes into the BST based on their values, recursively traversing the tree until finding the appropriate position. The 'remove' function removes nodes from the BST while maintaining the BST properties, handling cases of nodes with no children, one child, and two children. The 'inorderTraversal' function is also included to print the elements of the BST in ascending order. The example creates a BST, inserts several nodes, performs removals, and then displays the resulting BST using the inorder traversal.

Section 7.12 - Tries

Overview

A trie, also known as a prefix tree, is a tree-based data structure used for efficient retrieval and storage of strings. It is particularly useful for solving problems related to string matching and searching. The key feature of a trie is that it represents the entire alphabet at each level of the tree, with each node storing a single character. This allows for fast lookup and search operations, making it suitable for tasks such as autocomplete, spell-checking, and IP routing. Tries offer a time complexity of $\mathcal{O}(L)$ for search, insert, and delete operations, where L is the length of the string being operated on, making them a powerful data structure for handling string-based data efficiently.

Trie Insert Algorithm

The true insert algorithm for a trie involves recursively traversing the trie based on the characters of the input string. Starting from the root node, it checks if the current character exists as a child node. If it does, the algorithm moves to the corresponding child node and continues the traversal. If the character is not found, a new node is created and added as a child to the current node. This process is repeated until all characters of the input string are processed. At the end of the traversal, the last node is marked as a terminal node to indicate the completion of the inserted word. The true insert algorithm efficiently constructs a trie by utilizing the recursive nature of the data structure, ensuring that all characters of the input string are appropriately represented and stored for efficient retrieval and search operations.

Trie Remove Algorithm

The trie remove algorithm involves recursively traversing the trie to find the node representing the target string to be removed. Starting from the root node, the algorithm follows the characters of the target string, checking if each character exists as a child node. If any character is not found, it means the target string does not exist in the trie, and the algorithm terminates. If all characters are found, the algorithm marks the last node representing the target string as a non-terminal node to indicate its removal. If this node has no other children, it is removed from its parent, and the parent's children are adjusted accordingly. This process is repeated recursively, moving up the trie, removing any unnecessary nodes that are no longer part of any other words. The trie remove algorithm ensures that the trie structure is maintained correctly after removing a string, optimizing the space and ensuring efficient retrieval and search operations.

Trie Algorithm Example

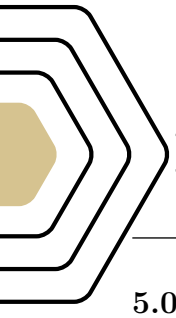
Below is an example of trie algorithms in C++:

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

struct TrieNode {
    unordered_map<char, TrieNode*> children;
    bool isTerminal;

    TrieNode() {
        isTerminal = false;
    }
};

class Trie {
    TrieNode* root;

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

    void insert(string word) {
        TrieNode* current = root;
        for (char ch : word) {
            if (current->children.find(ch) == current->children.end()) {
                current->children[ch] = new TrieNode();
            }
            current = current->children[ch];
        }
        current->isTerminal = true;
    }

    bool remove(string word) {
        return removeUtil(root, word, 0);
    }

private:
    bool removeUtil(TrieNode* node, const string& word, int index) {
        if (index == word.length()) {
            if (!node->isTerminal) {
                return false; // Word doesn't exist in trie
            }
            node->isTerminal = false;
            return node->children.empty(); // Check if node has no other children
        }

        char ch = word[index];
        if (node->children.find(ch) == node->children.end()) {
            return false; // Word doesn't exist in trie
        }

        TrieNode* child = node->children[ch];
        bool shouldRemoveChild = removeUtil(child, word, index + 1);

        if (shouldRemoveChild) {
            node->children.erase(ch);
            delete child;
            return node->children.empty();
        }

        return false;
    }
};

int main() {
    Trie trie;
    trie.insert("apple");
    trie.insert("banana");
    trie.insert("orange");

    cout << "Before removal: " << endl;
    cout << "Contains 'apple': " << trie.remove("apple") << endl;
```

```
73        cout << "Contains 'banana': " << trie.remove("banana") << endl;
74        cout << "Contains 'orange': " << trie.remove("orange") << endl;
75
76        cout << "After removal: " << endl;
77        cout << "Contains 'apple': " << trie.remove("apple") << endl;
78        cout << "Contains 'banana': " << trie.remove("banana") << endl;
79        cout << "Contains 'orange': " << trie.remove("orange") << endl;
80
81        return 0;
82    }
83
```

This example demonstrates the implementation of a trie data structure in C++, including the removal of words from the trie. The 'Trie' class provides an 'insert' method to add words to the trie and a 'remove' method to remove words. The 'remove' method uses a recursive helper function, 'removeUtil', to traverse the trie and find the node representing the target word. If the word exists in the trie, it is marked as a non-terminal node and removed from the trie structure if it has no other children. The example shows the removal of words from the trie and verifies their existence before and after removal.

Balanced Trees



Balanced Trees

5.0.1 Activities

The following are the activities that are planned for Week 5 of this course.

- Watch week videos
- Read Ch8 of the Data Structures zybook and take the in chapter quizzes(due next Monday)
- Programming Assignment 3 is Due next Tuesday. No new homeworks this week
- No new homework posted this week

5.0.2 Lectures

Here are the lectures that can be found for this week:

- Balanced Trees
- Red Black Tree Insert Operation

The lecture notes for this week are:

- Design & B-Trees Lecture Notes

5.0.3 Programming Assignment

There programming assignment for Week 5 is:

- Programming Assignment 3 - Binary Search Tree

5.0.4 Chapter Summary

The chapter of this week is **Chapter 8: Balanced Trees**.

Section 8.1 - Red-Black Tree: A Balanced Tree

Red-Black trees are a type of self-balancing binary search tree that maintain balance through a set of rules and operations. Each node in a Red-Black tree is assigned a color, either red or black, and follows specific rules that ensure the tree remains balanced. These rules include maintaining the black height property, where the number of black nodes on any path from the root to a leaf is the same, and ensuring that no red node has a red child. Red-Black trees offer efficient operations for insertion, deletion, and searching, with a guaranteed worst-case time complexity of $\mathcal{O}(\log(n))$. The self-balancing nature of Red-Black trees makes them suitable for a wide range of applications where maintaining balance and efficient operations are crucial.

These rules of Red-Black trees employ that they remain balanced and prevent long paths or imbalances, guaranteeing an efficient search, insertion, and deletion operations with a worst-case time complexity of $\mathcal{O}(\log(n))$. The rules for these trees are as follows:

- Every node in a Red-Black tree is either red or black.
- The root node of the tree is always black.
- All leaves (null or empty nodes) are considered black.
- If a node is red, both its children are black.
- Every path from a node to its descendant leaves contains the same number of black nodes. This is known as the black height property.
- Red nodes cannot have red children. A red node can only have black children.
- Null or empty nodes are considered black and do not affect the black height property.

Section 8.2 - Red-Black Tree: Rotations

Overview

Rotations in Red-Black trees are fundamental operations used to maintain balance and restructure the tree during insertion and deletion operations. There are two types of rotations: left rotation and right rotation. A left rotation is performed to move a node's right child up and the node down, effectively rotating them towards the left. Similarly, a right rotation moves a node's left child up and the node down, rotating them towards the right. These rotations preserve the Red-Black tree properties and help in maintaining a balanced tree structure. They are essential for adjusting the tree during operations to ensure efficient search, insertion, and deletion operations while keeping the tree balanced.

Left Rotation Algorithm

The left rotation algorithm in Red-Black trees is used to balance the tree and maintain the Red-Black properties during insertion and deletion operations. It involves reorganizing the structure of the tree by rotating a node and its right child towards the left. During the left rotation, the right child of the node becomes the new parent, while the original node becomes the left child of the new parent. The left child of the right child becomes the right child of the original node. This rotation helps in adjusting the tree's structure to ensure that the Red-Black properties, such as color balance and ordered key values, are maintained. It is a crucial operation in Red-Black trees to maintain a balanced and efficient data structure.

Left Rotation Algorithm Example

Below is an example of the left rotation algorithm in C++:

```
1  struct Node {
```

```
    int key;
    Node* left;
    Node* right;
    bool isRed;
};

// Left rotation function
Node* leftRotate(Node* node) {
    Node* newParent = node->right;
    node->right = newParent->left;
    newParent->left = node;
    return newParent;
}

```

In this example, we have a 'Node' struct representing a node in a Red-Black tree. The 'leftRotate' function performs a left rotation on a given node. It reassigns the parent and child pointers to rotate the node towards the left. The 'newParent' node becomes the new parent, and the original node becomes the left child of the new parent. This operation helps in rebalancing the tree while preserving the Red-Black properties.

The left rotation algorithm in Red-Black trees is implemented through the 'leftRotate' function. It rearranges the structure of the tree by rotating a node and its right child towards the left. The new parent becomes the right child's left child, and the original node becomes the left child of the new parent. This operation plays a crucial role in maintaining balance and ensuring the Red-Black properties are satisfied in the tree.

Right Rotation Algorithm

The right rotation algorithm in Red-Black trees is a fundamental operation used to rebalance the tree while preserving the Red-Black properties. It involves rotating a node and its left child towards the right. The left child becomes the new parent, with the original node as its right child. Additionally, the right child of the new parent becomes the left child of the original node. By performing this rotation, the structure of the tree is adjusted to maintain balance and ensure that the Red-Black properties, such as maintaining the black height and avoiding consecutive red nodes, are satisfied. The right rotation algorithm is crucial in maintaining the integrity and stability of Red-Black trees during insertions and deletions.

Right Rotation Algorithm Example

Below is an example of the right rotation algorithm in C++:

```
struct Node {
    int data;
    Node* left;
    Node* right;
    Node* parent;
    bool isRed;
};

// Perform a right rotation on the given node
void rightRotate(Node*& root, Node* node) {
    Node* pivot = node->left;
    node->left = pivot->right;

    if (node->left != nullptr)
        node->left->parent = node;

    pivot->parent = node->parent;

    if (node->parent == nullptr)
        root = pivot;
    else if (node == node->parent->left)
        node->parent->left = pivot;
    else
        node->parent->right = pivot;

    pivot->right = node;
    node->parent = pivot;
}

```

The provided example demonstrates the right rotation algorithm in a Red-Black tree. The 'rightRotate' function performs a right rotation on a given node by reassigning pointers and adjusting parent-child relationships. This operation helps maintain balance and uphold the Red-Black properties. In the example, a node with a left child is rotated to the right, resulting in the left child becoming the new parent and the original node as its right child. The example showcases the usage of the right rotation algorithm in constructing and manipulating Red-Black trees.

Section 8.3 - Red-Black Tree Insertion

Insertion in a Red-Black tree involves a process to maintain the tree's balance and adherence to Red-Black properties. The algorithm begins with a standard binary search tree insertion, followed by adjustments to ensure the tree remains a valid Red-Black tree. After inserting a new node, color violations and structural imbalances are resolved through a series of rotations and recoloring operations. These adjustments aim to preserve the key properties of Red-Black trees, such as black height consistency and the absence of consecutive red nodes. The insertion process guarantees an efficient and balanced Red-Black tree structure suitable for fast search, insertion, and deletion operations.

Insertion Algorithm Example

Below is an example of the insertion algorithm in C++:

```cpp
#include <iostream>
#include <memory>

enum class Color { Red, Black };

struct Node {
    int key;
    Color color;
    std::shared_ptr<Node> left;
    std::shared_ptr<Node> right;
    std::shared_ptr<Node> parent;

    Node(int k) : key(k), color(Color::Red), left(nullptr),
    right(nullptr), parent(nullptr) {}
};

void RedBlackTree::Insert(int key) {
    auto node = std::make_shared<Node>(key);

    // Perform binary search tree insertion
    // (Code for searching and finding the appropriate position for insertion)

    // Perform color adjustments and rotations to
    // maintain Red-Black tree properties
    InsertFixup(node);

    // Adjust the root, if necessary
    while (root->parent != nullptr) {
        root = root->parent;
    }
}

```

In this example, a Red-Black tree is implemented using a 'Node' struct and a 'RedBlackTree' class. The 'Insert' function performs a standard binary search tree insertion, followed by the 'InsertFixup' function that adjusts the tree to maintain Red-Black properties. The 'LeftRotate' and 'RightRotate' functions are used for performing left and right rotations, respectively. The example demonstrates the basic structure and flow of inserting elements into a Red-Black tree, ensuring the tree remains balanced and adheres to the Red-Black properties.

Section 8.4 - Red-Black Tree: Removal

In Red-Black trees, removal of nodes involves a complex process to maintain the tree's balance and preserve its Red-Black properties. The removal algorithm typically follows the steps of a standard binary search tree deletion, but with additional adjustments and rotations to ensure the resulting tree remains balanced. The removal process includes cases for handling different scenarios such as removing a node with no children, removing a node with one child, or removing a node with two children. Through careful restructuring, recoloring, and rotation operations, the Red-Black tree is modified to satisfy the Red-Black properties while preserving the search tree structure. This ensures efficient and consistent removal of nodes while maintaining the desired balance and characteristics of the Red-Black tree.

Removal Algorithm Example

Below is an example of the removal algorithm in C++:

```cpp
// Remove a node with value 'value' from the Red-Black tree
void removeNode(int value) {
    Node* nodeToRemove = searchNode(value); // Find the node to remove
    if (nodeToRemove == nullptr) {
        return; // Node not found, exit
    }
    deleteNode(nodeToRemove); // Delete the node from the tree
}

// Delete a node from the Red-Black tree
void deleteNode(Node* node) {
    Node* y = node; // Temporary variable for the node to be deleted or moved
    Node* x = nullptr; // Temporary variable for the node that replaces 'y'
    bool yOriginalColor = y->color; // Save the original color of 'y'

    // Case 1: Node to be deleted has no children or only one child
    if (node->left == nullptr) {
        x = node->right;
        transplant(node, node->right);
    }
    else if (node->right == nullptr) {
        x = node->left;
        transplant(node, node->left);
    }
    else {
        // Case 2: Node to be deleted has two children
        y = minimum(node->right); // Find the successor of 'node'
        yOriginalColor = y->color;
        x = y->right;
        if (y->parent == node) {
            x->parent = y; // Update 'x' parent
        }
        else {
            transplant(y, y->right);
            y->right = node->right;
            y->right->parent = y;
        }
        transplant(node, y);
        y->left = node->left;
        y->left->parent = y;
        y->color = node->color;
    }

    delete node; // Delete the node from memory

    // Case 3: Fix any violations and maintain Red-Black tree properties
    if (yOriginalColor == BLACK) {
        deleteFixup(x);
    }
}

// Other necessary functions for Red-Black tree implementation

```

In the provided example, we have a Red-Black tree implementation in C++. The 'removeNode' function is responsible for removing a node with a specified value from the tree. The function first searches for the node to remove using the 'searchNode' function. If the node is found, it calls the 'deleteNode' function to delete the node from the tree. The 'deleteNode' function performs the standard binary search tree deletion steps and then performs additional adjustments and rotations to maintain the Red-Black tree properties. These additional operations ensure that the tree remains balanced and satisfies the Red-Black properties even after the removal of a node.

Code Complexity, Binary Search Tree, Sorting Methods

Code Complexity, Binary Search Tree, Sorting Methods

6.0.1 Activities

The following are the activities that are planned for Week 6 of this course.

- Watch week videos.
- Read Ch. 9 and Ch. 10 of the Data Structures zyBook and take the in chapter quizzes (due next Monday).
- BST Homework is due Tuesday.
- Start on Sorting Homework (Due next Tuesday).
- Make sure your Proctorio extension is installed and working. — Take verification quiz to make sure your browser is set up and ready for the exam next week.

6.0.2 Lectures

Here are the lectures that can be found for this week:

- Complexity
- BST Operation Complexity
- Hard Problems
- Bubble Sort
- Merge Sort
- Item Sort

The lectures for this week are:

- Complexity & Sorting Algorithms Lecture Notes

6.0.3 Programming Assignment

The programming assignment for Week 6 is:

- Programming Assignment 4 - Sorting
- Sorting Algorithms Interview Notes

6.0.4 Chapter Summary

The first chapter of this week is **Chapter 9: Searching & Algorithm Analysis**.

Section 9.1 - Constant Time Operations

Constant time operations in the context of object-oriented programming refer to operations that execute in a fixed and predictable amount of time, regardless of the size or complexity of the data being processed. These operations provide efficient and consistent performance, allowing developers to perform tasks with a time complexity of $\mathcal{O}(1)$. By implementing constant time operations, developers can optimize their code for quick and reliable execution, resulting in improved scalability and responsiveness in their object-oriented programs.

Section 9.2 - Growth of Functions & Complexity

Upper & Lower Bounds

Upper and lower bounds play a crucial role in analyzing the growth rates and complexities of functions and algorithms. An upper bound represents the worst-case growth rate of a function, while a lower bound represents the best-case growth rate. The big $\mathcal{O}$ notation ($\mathcal{O}$ notation) is used to express upper bounds, while the big Ω notation (Omega notation) represents lower bounds. By combining both upper and lower bounds, the tightest possible bound, denoted by Θ notation (Theta notation), can be determined. Analyzing these bounds enables us to understand how the performance of algorithms and functions scales with input size, aiding in algorithm selection, optimization, and predicting efficiency in different scenarios.

Notation	General Form	Meaning
$\mathcal{O}$	$T(N) = \mathcal{O}(f(N))$	A positive constant c exists such that, for all $N \geq 1$, $T(N) \leq c * f(N)$
Ω	$T(N) = \Omega(f(N))$	A positive constant c exists such that, for all $N \geq 1$, $T(N) \geq c * f(N)$
Θ	$T(N) = \Omega(f(N))$	$T(N) = \mathcal{O}(f(N))$ and $T(N) = \Omega(f(N))$

Section 9.3 - $\mathcal{O}$ Notation

Big $\mathcal{O}$ Notation

Big $\mathcal{O}$ notation is a fundamental concept in computer science used to analyze the efficiency and scalability of algorithms. It provides a standardized way to express the upper bound on the worst-case time complexity of an algorithm in terms of the input size. Big $\mathcal{O}$ notation represents the growth rate of an algorithm as a function of the input size, allowing comparisons between different algorithms and helping in algorithm selection. It disregards constant factors and lower-order terms, focusing solely on the dominant term that determines the algorithm's scalability. Big $\mathcal{O}$ notation provides a concise and abstract representation of algorithmic complexity, enabling engineers and developers to understand and reason about the efficiency of algorithms without getting caught up in implementation details.

Big $\mathcal{O}$ Notation of Composite Functions

When dealing with composite functions, Big $\mathcal{O}$ notation provides a concise way to express the overall growth rate of the composed functions. The Big $\mathcal{O}$ notation of composite functions is determined by considering the dominant term in each function and evaluating how they interact with one another. If two functions $f(n)$ and $g(n)$ have respective Big $\mathcal{O}$ notations of $\mathcal{O}(f(n))$ and $\mathcal{O}(g(n))$, the Big $\mathcal{O}$ notation of the composite function is determined by taking the maximum growth rate among the two. In other words, if the growth rate of $f(n)$ is faster than $g(n)$, the composite function's Big $\mathcal{O}$ notation will be $\mathcal{O}(f(n))$. This approach allows for a simplified representation of the

overall complexity of a composite function and facilitates analysis of algorithmic efficiency when multiple functions are involved.

Composite Function	Big $\mathcal{O}$ Notation
$c \cdot \mathcal{O}(f(N))$	$\mathcal{O}(f(N))$
$c + \mathcal{O}(f(N))$	$\mathcal{O}(f(N))$
$g(N) \cdot \mathcal{O}(f(N))$	$\mathcal{O}(g(N) * f(N))$
$g(N) + \mathcal{O}(f(N))$	$\mathcal{O}(g(N) + f(N))$

Runtime Growth Rate

Runtime growth rate refers to how the execution time of an algorithm increases as the input size grows. It is commonly measured using Big $\mathcal{O}$ notation, which expresses the worst-case upper bound on the growth rate of an algorithm. The runtime growth rate provides insights into the scalability and efficiency of an algorithm, allowing developers to understand how the algorithm performs with larger inputs. By analyzing the runtime growth rate, developers can make informed decisions about algorithm selection, identify potential performance bottlenecks, and optimize their code to improve overall efficiency. Understanding the runtime growth rate is crucial for designing algorithms that can handle larger and more complex datasets effectively.

Common Big $\mathcal{O}$ Complexities

Common Big $\mathcal{O}$ complexities are used to describe the performance characteristics of algorithms as the input size increases. Some of the most common Big $\mathcal{O}$ complexities include $\mathcal{O}(1)$ O(1), $\mathcal{O}(\log(n))$ (logarithmic time), $\mathcal{O}(n)$ (linear time), $\mathcal{O}(n\log(n))$ (linearithmic time), $\mathcal{O}(n^2)$ (quadratic time), $\mathcal{O}(2^n)$ (exponential time), and $\mathcal{O}(n!)$ (factorial time). These complexities represent different rates of growth and have a significant impact on the efficiency of algorithms. Algorithms with constant time complexity execute in a fixed and predictable time regardless of input size, while linear, quadratic, and exponential time complexities indicate a proportional increase in execution time with the input size. Understanding these common Big $\mathcal{O}$ complexities helps in analyzing and comparing algorithms, selecting appropriate solutions for specific problem domains, and optimizing code to achieve efficient algorithmic performance.

Notation	Name
$\mathcal{O}(1)$	Constant
$\mathcal{O}(\log(N))$	Logarithmic
$\mathcal{O}(N)$	Linear
$\mathcal{O}(N\log(N))$	Linearithmic
$\mathcal{O}(N^2)$	Quadratic
$\mathcal{O}(c^N)$	Exponential

Common Big $\mathcal{O}$ Complexity Examples

Below are examples of some common Big $\mathcal{O}$ complexities in C++:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

// O(1) - Constant time complexity
void printFirstElement(const std::vector<int>& data) {
    if (!data.empty())
        std::cout << "First element: " << data[0] << std::endl;
}

// O(log n) - Logarithmic time complexity
void binarySearch(const std::vector<int>& data, int target) {
    int low = 0;
    int high = data.size() - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (data[mid] == target) {
            std::cout << "Found at index " << mid << std::endl;
            return;
        }
        else if (data[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
    std::cout << "Not found!" << std::endl;
}

// O(n) - Linear time complexity
void linearSearch(const std::vector<int>& data, int target) {
    for (int i = 0; i < data.size(); i++) {
        if (data[i] == target) {
```

```
            std::cout << "Found at index " << i << std::endl;
            return;
        }
    }
    std::cout << "Not found!" << std::endl;
}

// O(n log n) - Linearithmic time complexity
void mergeSort(std::vector<int>& data) {
    if (data.size() <= 1)
        return;

    std::vector<int> left(data.begin(), data.begin() + data.size() / 2);
    std::vector<int> right(data.begin() + data.size() / 2, data.end());

    mergeSort(left);
    mergeSort(right);

    std::merge(left.begin(), left.end(), right.begin(),
    right.end(), data.begin());
}

// O(n^2) - Quadratic time complexity
void bubbleSort(std::vector<int>& data) {
    for (int i = 0; i < data.size() - 1; i++) {
        for (int j = 0; j < data.size() - i - 1; j++) {
            if (data[j] > data[j + 1])
                std::swap(data[j], data[j + 1]);
        }
    }
}

// O(2^n) - Exponential time complexity
int fibonacci(int n) {
    if (n <= 1)
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

// O(n!) - Factorial time complexity
int factorial(int n) {
    if (n <= 1)
        return 1;
    return n * factorial(n - 1);
}

int main() {
    std::vector<int> data = { 5, 3, 1, 4, 2 };
    int target = 4;

    printFirstElement(data); // O(1)
    binarySearch(data, target); // O(log n)
    linearSearch(data, target); // O(n)
    mergeSort(data); // O(n log n)
    bubbleSort(data); // O(n^2)
    int fib = fibonacci(5); // O(2^n)
    int fact = factorial(5); // O(n!)

    std::cout << "Fibonacci: " << fib << std::endl;
    std::cout << "Factorial: " << fact << std::endl;

    return 0;
}

```

The examples provided showcase different common Big $\mathcal{O}$ complexities in C++. They illustrate algorithms with varying growth rates as the input size increases. The examples include constant time complexity $\mathcal{O}(1)$, logarithmic time complexity $\mathcal{O}(log(n))$, linear time complexity $\mathcal{O}(n)$, linearithmic time complexity $\mathcal{O}(n \log(n))$, quadratic time complexity $\mathcal{O}(n^2)$, exponential time complexity $\mathcal{O}(2^n)$, and factorial time complexity $\mathcal{O}(n!)$. Understanding these complexities is crucial for analyzing algorithmic efficiency, selecting appropriate solutions, and optimizing code. By considering the performance characteristics of algorithms, developers can make informed decisions to design efficient and scalable solutions.

Section 9.4 - Algorithm Analysis

Worst-Case Algorithm Analysis

Worst-case algorithm analysis is a method used to assess the maximum possible running time of an algorithm for a given input size. It involves identifying the input that would lead to the algorithm's worst performance and analyzing its execution time under that scenario. By focusing on the worst-case scenario, this approach provides a guarantee on the upper bound of the algorithm's running time. Worst-case analysis is widely used in algorithm design and evaluation, as it allows programmers to make informed decisions about algorithm selection, optimization, and predicting the algorithm's behavior in real-world scenarios. It helps ensure that the algorithm performs efficiently and reliably even in unfavorable or challenging inputs.

Counting Constant Time Operations

Counting constant time operations involves determining the number of basic operations executed by an algorithm, where each operation takes a constant amount of time regardless of the input size. It entails identifying and quantifying individual operations such as arithmetic operations, assignments, comparisons, or accessing elements in an array. By counting these operations, we can evaluate the efficiency of an algorithm in terms of its time complexity and analyze how it scales with different input sizes. Counting constant time operations provides a quantitative measure to compare and reason about the efficiency of algorithms, enabling developers to make informed decisions regarding algorithm selection and optimization.

Runtime Analysis of Nested Loops

Runtime analysis of nested loops involves analyzing the time complexity of algorithms that contain multiple nested loops. This analysis helps determine how the execution time of the algorithm grows as the input size increases. By examining the nesting structure and the number of iterations performed by each loop, we can derive the overall complexity of the nested loops. The runtime analysis considers the number of iterations and the operations within each loop to estimate the overall efficiency of the algorithm. Understanding the runtime analysis of nested loops is essential for identifying potential performance bottlenecks, optimizing code, and designing efficient algorithms for handling larger input sizes.

Algorithm Analysis Example

Below is an example of analyzing an algorithm in C++:

```cpp
bool searchValue(const std::vector<std::vector<int>>& array, int target) {
    // Outer loop iterates through each row
    for (const auto& row : array) {
        // Inner loop iterates through each element in the row
        for (const auto& element : row) {
            if (element == target)
                return true;
        }
    }
    return false;
}

```

The provided example demonstrates the concepts of worst-case algorithm analysis, counting constant time operations, and runtime analysis of nested loops. The worst-case analysis involves considering the scenario where the target element is at the last position or absent in the array, resulting in the algorithm traversing through all elements. Counting constant time operations involves recognizing that each comparison operation ('element == target') within the nested loops takes a constant amount of time. The runtime analysis of the nested loops determines that the time complexity of the algorithm is $\mathcal{O}(m * n)$, where m is the number of rows and n is the number of columns in the array. Understanding these concepts allows us to evaluate the algorithm's efficiency, optimize code if necessary, and predict its performance as the size of the input array changes.

Section 9.5 - Searching & Algorithms

Algorithms

Algorithms are step-by-step procedures or sets of rules used to solve problems, perform computations, or achieve specific tasks. They are fundamental to computer science and play a vital role in software development. Algorithms can be designed to handle a wide range of tasks, such as sorting data, searching for elements, graph traversal, optimization, and more. The efficiency of an algorithm is evaluated through its time complexity (how long it takes to run) and space complexity (how much memory it requires). Efficient algorithms are designed to minimize resource usage and deliver fast and scalable solutions. The study of algorithms involves analyzing their properties, designing new algorithms, and selecting the most appropriate algorithmic approach for a given problem. Ultimately, algorithms enable the automation and optimization of processes, providing the foundation for various computational tasks and applications.

Algorithm Runtime

Algorithm runtime refers to the amount of time it takes for an algorithm to execute and produce a result. It is a critical aspect of algorithm analysis, as it helps evaluate the efficiency and performance of algorithms. The runtime of an algorithm is influenced by factors such as the input size, the complexity of the operations performed, and the algorithmic design itself. Analyzing algorithm runtime involves estimating how the execution time increases as the input size grows, typically using Big $\mathcal{O}$ notation. Understanding algorithm runtime allows developers to identify bottlenecks, optimize code, and make informed decisions about algorithm selection to ensure efficient and scalable solutions. By striving for optimal runtime, developers can create algorithms that deliver faster results and handle larger data sets effectively.

Section 9.6 - Analyzing the Time Complexity of Recursive Algorithms

---

Recurrence Relations

Recurrence relations are mathematical equations used to describe the time complexity of recursive algorithms. They provide a way to express the runtime of an algorithm in terms of the runtime of smaller subproblems. Recurrence relations are often used when analyzing the time complexity of divide-and-conquer algorithms, dynamic programming algorithms, or any algorithm that exhibits recursive behavior. By solving the recurrence relation, we can obtain an explicit formula or a bound on the time complexity of the algorithm. Recurrence relations help in understanding how the algorithm's runtime grows with the input size and aid in comparing and evaluating different recursive algorithms. They are a powerful tool for reasoning about the time complexity of recursive algorithms and guiding algorithmic design decisions.

Recursion Trees

Recursion trees are graphical representations used to analyze the time complexity of recursive algorithms. They provide a visual depiction of the recursive calls and their respective sizes during the execution of the algorithm. Each node in the tree represents a recursive call, and the edges represent the recursive relationship between the calls. The tree's depth corresponds to the number of recursive calls made, and the branches represent the sizes of the subproblems at each level. By examining the recursion tree, we can analyze the total number of nodes (recursive calls) and the work done at each node to determine the algorithm's time complexity. This analysis helps in understanding the growth pattern of the algorithm as the input size increases and provides insights into optimizing the algorithm or making design choices to improve its efficiency. Recursion trees provide a clear and visual representation of the recursive algorithm's execution, aiding in the analysis and understanding of its time complexity.

The second chapter of this week is **Chapter 10: Sorting Algorithms**.

Section 10.1 - Sorting: Introduction

Sorting in object-oriented programming (OOP) involves organizing a collection of objects or data structures in a particular order based on specific criteria. OOP provides various techniques and algorithms to achieve efficient

sorting, aiming to improve code reusability and maintainability. By encapsulating sorting functionality within classes or methods, OOP enables the creation of reusable and modular sorting components. This approach enhances code readability, simplifies maintenance, and allows for easy adaptation of sorting algorithms based on changing requirements. Additionally, OOP encourages the use of interfaces and inheritance, facilitating the implementation of different sorting strategies while adhering to common contracts or base classes.

Section 10.2 - Quicksort

Quicksort

Quicksort is a highly efficient sorting algorithm commonly used in computer science. It follows a divide-and-conquer strategy, recursively dividing an array or list into smaller subarrays based on a pivot element, and then sorting these subarrays independently. The key idea behind Quicksort is to select a pivot element, typically the last or middle element of the array, and partition the remaining elements into two groups: those smaller than the pivot and those greater than the pivot. This process is repeated recursively on the two resulting subarrays until the entire array is sorted. Quicksort's average and best-case time complexity is $\mathcal{O}(n \log(n))$, making it one of the fastest sorting algorithms in practice. However, in its worst-case scenario, when the pivot selection is unbalanced, the time complexity can degrade to $\mathcal{O}(n^2)$. Various optimizations, such as randomized pivot selection or using different partitioning schemes, can be employed to mitigate the worst-case scenario.

Partitioning Algorithm

The partitioning algorithm is a crucial step in many sorting algorithms, including Quicksort. It is responsible for rearranging elements in an array or list so that all elements smaller than a chosen pivot element are placed to its left, and all elements larger than the pivot are placed to its right. The partitioning process involves maintaining two pointers, one starting from the left end and the other from the right end of the array. These pointers traverse towards each other, swapping elements when necessary, until they meet. This ensures that the pivot element is in its final sorted position, with smaller elements on its left and larger elements on its right. The partitioning algorithm enables efficient sorting by dividing the problem into smaller subproblems, allowing for subsequent recursive sorting on the resulting subarrays. It is a fundamental component of many efficient sorting algorithms, contributing to their overall performance and effectiveness.

Recursively Sorting Partitions

Recursively sorting partitions is a common approach used in various sorting algorithms, such as Quicksort and Merge Sort. This technique involves dividing an array or list into smaller partitions or subarrays and recursively applying the sorting algorithm to each of these partitions. By recursively sorting smaller subarrays, the algorithm gradually reduces the problem size until it reaches base cases where the subarrays are either empty or contain a single element, which are inherently sorted. The sorted subarrays are then merged or combined in a manner that guarantees the final sorted result. This recursive approach leverages the divide-and-conquer strategy, allowing for efficient sorting of larger datasets by breaking them down into more manageable pieces and leveraging the sorted results to obtain the overall sorted array.

Quicksort Runtime

The runtime of Quicksort, on average and in the best case, is considered very efficient, with a time complexity of $\mathcal{O}(n \log(n))$. This means that the sorting algorithm can efficiently sort a collection of n elements by making approximately $\log(n)$ recursive partitioning steps, each of which requires linear time to perform the necessary element comparisons and swaps. However, it's important to note that Quicksort's worst-case time complexity is $\mathcal{O}(n^2)$, which occurs when the chosen pivot is consistently the smallest or largest element in the array, resulting in unbalanced partitions. To mitigate this, randomized pivot selection or other techniques can be employed to ensure a more balanced partitioning, reducing the likelihood of the worst-case scenario. In practice, Quicksort is often a preferred choice due to its average-case efficiency and adaptability to various datasets.

Quick Sort Example

Below is an example of a quicksort example in C++:

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

int partition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    std::vector<int> arr = { 7, 2, 1, 6, 8, 5, 3, 4 };
    int size = arr.size();

    quickSort(arr, 0, size - 1);

    std::cout << "Sorted array: ";
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}

```

The provided example demonstrates the Quick Sort algorithm implemented in C++. Quick Sort is a divide-and-conquer algorithm that works by selecting a pivot element and partitioning the array into two subarrays based on the pivot. The partitioning step places the pivot element in its correct position, with smaller elements to its left and larger elements to its right. The algorithm recursively applies this partitioning process to the subarrays until the entire array is sorted. The example showcases how the Quick Sort algorithm can be used to sort a vector of integers. By selecting appropriate pivot elements and efficiently partitioning the array, Quick Sort achieves an average-case time complexity of $\mathcal{O}(\log(n))$. The example demonstrates the sorted array as the output, highlighting the effectiveness of the Quick Sort algorithm for sorting data efficiently.

Section 10.3 - Merge Sort

Merge Sort Overview

Merge Sort is a popular sorting algorithm that follows the divide-and-conquer paradigm. It works by recursively dividing the input array into smaller subarrays until they become single elements. Then, it merges the subarrays in a sorted order to obtain a fully sorted array. The algorithm utilizes the merge operation, which combines two sorted arrays into a single sorted array. Merge Sort has a time complexity of $\mathcal{O}(n \log(n))$ in the worst, best, and average cases, making it efficient for large input sizes. It is known for its stability, as it maintains the relative order of equal elements. Merge Sort is widely used due to its reliable performance and ability to handle large datasets. However, it requires additional space for the temporary arrays used during the merge process, which can be a consideration when dealing with limited memory resources. Overall, Merge Sort is a reliable and efficient sorting algorithm for a wide range of applications.l

Merge Sort Partitioning

Merge Sort partitioning is the process by which the input array is divided into smaller subarrays during the execution of the Merge Sort algorithm. It follows a recursive approach, repeatedly halving the array until each subarray contains only one element. This process is known as the "divide" step in the divide-and-conquer paradigm. Merge Sort partitions the array by calculating the middle index and recursively applying the partitioning process to the left and right halves of the array. This division continues until the subarrays contain only one element, at which point the merging step begins. Partitioning is a key component of Merge Sort that enables the subsequent merging of sorted subarrays, ultimately leading to the generation of a fully sorted array.

Merge Sort Algorithm

Merge Sort is a widely used sorting algorithm that employs a divide-and-conquer approach to efficiently sort an array of elements. The algorithm divides the input array into smaller subarrays until each subarray consists of only one element. Then, it merges adjacent subarrays in a sorted manner until the entire array is sorted. The merging process compares the elements from the subarrays and combines them into a single sorted subarray. Merge Sort has a time complexity of $\mathcal{O}(n \log(n))$ in the worst, best, and average cases, making it highly efficient for large input sizes. It is known for its stability, meaning that it maintains the relative order of equal elements. Merge Sort is a reliable and widely used sorting algorithm due to its efficient performance, stability, and scalability.

Merge Sort Algorithm Example

Below is an example of the merge sort algorithm in C++:

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

void merge(std::vector<int>& arr, int left, int middle, int right) {
    int leftSize = middle - left + 1;
    int rightSize = right - middle;

    std::vector<int> leftArr(leftSize);
    std::vector<int> rightArr(rightSize);

    for (int i = 0; i < leftSize; i++) {
        leftArr[i] = arr[left + i];
    }

    for (int j = 0; j < rightSize; j++) {
        rightArr[j] = arr[middle + 1 + j];
    }

    int i = 0, j = 0, k = left;

    while (i < leftSize && j < rightSize) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

    while (i < leftSize) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }

    while (j < rightSize) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}

void mergeSort(std::vector<int>& arr, int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;

        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);

        merge(arr, left, middle, right);
    }
}

int main() {
    std::vector<int> arr = { 7, 2, 1, 6, 8, 5, 3, 4 };
    int size = arr.size();

    mergeSort(arr, 0, size - 1);
```

```
61
62      std::cout << "Sorted array: ";
63      for (int i = 0; i < size; i++) {
64          std::cout << arr[i] << " ";
65      }
66      std::cout << std::endl;
67
68      return 0;
69  }
70
```

The provided example demonstrates the Merge Sort algorithm implemented in C++. Merge Sort is a divide-and-conquer algorithm that efficiently sorts an array by recursively dividing it into smaller subarrays and then merging them in a sorted manner. The example showcases how the Merge Sort algorithm can be used to sort a vector of integers. By dividing the array, merging sorted subarrays, and eventually producing a fully sorted array, Merge Sort achieves an average-case time complexity of $\mathcal{O}(n \log(n))$. The example displays the sorted array as the output, highlighting the effectiveness of the Merge Sort algorithm in sorting data efficiently.

Section 10.4 - Bubble Sort

Bubble Sort is a simple and intuitive sorting algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order. It iterates through the array multiple times, with each pass pushing the largest unsorted element to the end. The algorithm continues until the entire array is sorted. Bubble Sort has a time complexity of $\mathcal{O}(n^2)$ in the worst, best, and average cases. While Bubble Sort is easy to understand and implement, it is not the most efficient sorting algorithm for large datasets. However, it can be useful for sorting small arrays or when simplicity is prioritized over efficiency.

Bubble Sort Algorithm Example

Below is an example of the bubble sort algorithm example in C++:

```
1   #include <iostream>
2   #include <vector>
3
4   void bubbleSort(std::vector<int>& arr) {
5       int n = arr.size();
6       bool swapped;
7
8       for (int i = 0; i < n - 1; i++) {
9           swapped = false;
10
11          for (int j = 0; j < n - i - 1; j++) {
12              if (arr[j] > arr[j + 1]) {
13                  std::swap(arr[j], arr[j + 1]);
14                  swapped = true;
15              }
16          }
17
18          if (!swapped) {
19              // If no swapping occurred in the inner loop,
20              // the array is already sorted
21              break;
22          }
23      }
24  }
25
26  int main() {
27      std::vector<int> arr = { 7, 2, 1, 6, 8, 5, 3, 4 };
28      int size = arr.size();
29
30      bubbleSort(arr);
31
32      std::cout << "Sorted array: ";
33      for (int i = 0; i < size; i++) {
34          std::cout << arr[i] << " ";
35      }
36      std::cout << std::endl;
37
38      return 0;
39  }
40
```

The provided example demonstrates the Bubble Sort algorithm implemented in C++. Bubble Sort is a simple

and inefficient sorting algorithm that repeatedly compares adjacent elements and swaps them if they are out of order. The algorithm iterates through the array multiple times, gradually pushing the larger elements towards the end of the array. Bubble Sort has a time complexity of $\mathcal{O}(n^2)$ in the worst, best, and average cases, making it inefficient for large datasets. However, it is easy to understand and implement, making it suitable for small arrays or educational purposes. The example showcases the sorted array as the output, highlighting the step-by-step comparison and swapping of elements that Bubble Sort performs to achieve a sorted result.

Section 10.5 - Selection Sort

Selection Sort is a simple sorting algorithm that repeatedly selects the smallest element from the unsorted portion of the array and swaps it with the element at the beginning of the sorted portion. It maintains two subarrays within the array: the left portion represents the sorted elements, while the right portion represents the unsorted elements. In each iteration, the algorithm finds the smallest element from the unsorted portion and places it in its correct position in the sorted portion. This process continues until the entire array is sorted. Selection Sort has a time complexity of $\mathcal{O}(n^2)$ in the worst, best, and average cases, making it inefficient for large datasets. However, it requires minimal additional memory and performs well for small arrays or when memory usage is a concern.

Selection Sort Algorithm Example

Below is an example of the selection sort algorithm in C++:

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

void selectionSort(std::vector<int>& arr) {
    int n = arr.size();

    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;

        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        std::swap(arr[i], arr[minIndex]);
    }
}

int main() {
    std::vector<int> arr = { 7, 2, 1, 6, 8, 5, 3, 4 };
    int size = arr.size();

    selectionSort(arr);

    std::cout << "Sorted array: ";
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}

```

The provided example demonstrates the Selection Sort algorithm implemented in C++. Selection Sort is a simple and inefficient sorting algorithm that repeatedly selects the smallest element from the unsorted portion of the array and swaps it with the element at the beginning of the sorted portion. The algorithm maintains two subarrays within the array, with the left portion representing the sorted elements and the right portion representing the unsorted elements. Selection Sort has a time complexity of $\mathcal{O}(n^2)$ in the worst, best, and average cases, making it inefficient for large datasets. However, it requires minimal additional memory and performs adequately for small arrays or situations where memory usage is a concern. The example showcases the sorted array as the output, demonstrating the step-by-step selection and swapping of elements that Selection Sort performs to achieve the desired sorting result.

Section 10.6 - Insertion Sort

Insertion Sort is a simple and efficient sorting algorithm that builds the final sorted array one element at a time. It starts with an initially empty sorted portion and gradually inserts each unsorted element into its correct position within the sorted portion. The algorithm iterates through the unsorted portion, comparing each element with the elements in the sorted portion and shifting them to the right if they are greater. This process continues until all elements are inserted into their appropriate positions, resulting in a fully sorted array. Insertion Sort has a time complexity of $\mathcal{O}(n^2)$ in the worst, best, and average cases. However, it performs exceptionally well on small datasets or partially sorted arrays, making it a suitable choice for such scenarios. Insertion Sort is an in-place algorithm, meaning it does not require additional memory beyond the input array, making it memory-efficient.

Insertion Sort Algorithm Example

Below is an example of the insertion sort algorithm in C++:

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

void insertionSort(std::vector<int>& arr) {
    int n = arr.size();

    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = key;
    }
}

int main() {
    std::vector<int> arr = { 7, 2, 1, 6, 8, 5, 3, 4 };
    int size = arr.size();

    insertionSort(arr);

    std::cout << "Sorted array: ";
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}

```

The provided example demonstrates the Insertion Sort algorithm implemented in C++. Insertion Sort is a simple and efficient sorting algorithm that builds the final sorted array one element at a time. It starts with an initially empty sorted portion and gradually inserts each unsorted element into its correct position within the sorted portion. The algorithm iterates through the unsorted portion, comparing each element with the elements in the sorted portion and shifting them to the right if they are greater. Insertion Sort has a time complexity of $\mathcal{O}(n^2)$ in the worst, best, and average cases. However, it performs exceptionally well on small datasets or partially sorted arrays. The example showcases the sorted array as the output, illustrating the step-by-step insertion of elements and the shifting of values that Insertion Sort performs to achieve the desired sorting result.

Section 10.7 - Shell Sort

Shell Sort is an efficient and adaptive sorting algorithm that improves upon the Insertion Sort algorithm by reducing the number of comparisons and data movements. It divides the array into smaller subarrays and performs Insertion Sort on each subarray. The key idea is to sort elements that are far apart, then progressively reduce the gap between elements being compared and swapped. This process is known as "diminishing increment sorting." Shell Sort utilizes a sequence of gaps, typically generated using the Knuth sequence, to determine the gap sizes. The algorithm continues to decrease the gap until it reaches 1, where it performs a final Insertion Sort pass. Shell Sort has a time complexity of $\mathcal{O}(n \log(n))$ in the best case, and its worst-case time complexity depends on the chosen

gap sequence. Overall, Shell Sort is a versatile algorithm that performs well on medium-sized arrays and offers better performance than Insertion Sort.

Shell Sort Algorithm Example

Below is an example of the shell sort algorithm in C++:

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

void shellSort(std::vector<int>& arr) {
    int n = arr.size();

    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;

            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }

            arr[j] = temp;
        }
    }
}

int main() {
    std::vector<int> arr = { 7, 2, 1, 6, 8, 5, 3, 4 };
    int size = arr.size();

    shellSort(arr);

    std::cout << "Sorted array: ";
    for (int i = 0; i < size; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}

```

The provided example demonstrates the Shell Sort algorithm implemented in C++. Shell Sort is an efficient sorting algorithm that improves upon Insertion Sort by dividing the array into smaller subarrays and performing Insertion Sort on each subarray. The algorithm utilizes a sequence of gaps, typically generated using the Knuth sequence, to determine the gap sizes. It starts with a large gap and progressively reduces the gap until it reaches 1, where it performs a final Insertion Sort pass. Shell Sort has a time complexity of $\mathcal{O}(n \log(n))$ in the best case, and its worst-case time complexity depends on the chosen gap sequence. The example showcases the sorted array as the output, demonstrating the step-by-step sorting process and the reduction of gaps that Shell Sort performs to achieve an efficiently sorted result.

Section 10.8 - Radix Sort

Radix Sort is a linear-time sorting algorithm that operates on the individual digits or characters of the elements being sorted. It groups the elements by their least significant digit, then by the next digit, and so on until all digits have been considered. This sorting process is typically performed using counting sort as a subroutine. Radix Sort is effective for sorting numbers, strings, or other data types where elements have multiple digits or characters. The time complexity of Radix Sort depends on the number of digits or characters in the elements being sorted, and it can be expressed as $\mathcal{O}(d * (n + k))$, where d is the number of digits or characters, n is the number of elements, and k is the range of values for each digit or character. Radix Sort is efficient for large datasets, especially when the number of digits or characters is relatively small.

Radix Sort Algorithm Example

Below is an example of the radix sort algorithm in C++:

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

    int getMax(const std::vector<int>& arr) {
        int max = arr[0];
```

```
        for (int i = 1; i < arr.size(); i++) {
            if (arr[i] > max) {
                max = arr[i];
            }
        }
        return max;
    }

    void countingSort(std::vector<int>& arr, int exp) {
        const int radix = 10;
        int n = arr.size();
        std::vector<int> count(radix, 0);
        std::vector<int> output(n, 0);

        for (int i = 0; i < n; i++) {
            count[(arr[i] / exp) % radix]++;
        }

        for (int i = 1; i < radix; i++) {
            count[i] += count[i - 1];
        }

        for (int i = n - 1; i >= 0; i--) {
            output[count[(arr[i] / exp) % radix] - 1] = arr[i];
            count[(arr[i] / exp) % radix]--;
        }

        for (int i = 0; i < n; i++) {
            arr[i] = output[i];
        }
    }

    void radixSort(std::vector<int>& arr) {
        int max = getMax(arr);

        for (int exp = 1; max / exp > 0; exp *= 10) {
            countingSort(arr, exp);
        }
    }

    int main() {
        std::vector<int> arr = { 170, 45, 75, 90, 802, 24, 2, 66 };
        int size = arr.size();

        radixSort(arr);

        std::cout << "Sorted array: ";
        for (int i = 0; i < size; i++) {
            std::cout << arr[i] << " ";
        }
        std::cout << std::endl;

        return 0;
    }

```

The provided example demonstrates the Radix Sort algorithm implemented in C++. Radix Sort is a linear-time sorting algorithm that operates on the individual digits of the elements being sorted. It utilizes a counting sort subroutine to group the elements by their least significant digit, then by the next digit, and so on until all digits have been considered. Radix Sort is efficient for sorting numbers or strings where elements have multiple digits or characters. The example showcases the sorted array as the output, illustrating the step-by-step sorting process performed by Radix Sort. The algorithm finds the maximum value in the array to determine the number of digits, then performs counting sort for each digit position. Radix Sort has a time complexity of $\mathcal{O}(d * (n + k))$, where d is the number of digits, n is the number of elements, and k is the range of values for each digit.

Section 10.9 - Overview of Fast Sorting Algorithms

Fast sorting algorithms are efficient algorithms that aim to sort data quickly, especially for large datasets. These algorithms, such as QuickSort, Merge Sort, and Heap Sort, employ different techniques to achieve fast and reliable sorting. QuickSort utilizes a divide-and-conquer strategy by selecting a pivot element and partitioning the array into subarrays based on the pivot. Merge Sort divides the array into smaller subarrays, recursively sorts them, and then merges them to obtain the final sorted array. Heap Sort constructs a binary heap from the input array and repeatedly extracts the maximum element to build the sorted array. These fast sorting algorithms typically have a time complexity of $\mathcal{O}(n \log(n))$ in the average and worst cases, making them suitable for sorting large datasets

efficiently. They often outperform slower sorting algorithms like Insertion Sort and Bubble Sort, especially when dealing with significant amounts of data.

Sorting Algorithm's Average Runtime Complexity

Sorting Algorithm	Average Case Runtime Complexity	Fast?
Selection Sort	$\mathcal{O}(N^2)$	No
Insertion Sort	$\mathcal{O}(N^2)$	No
Shell Sort	$\mathcal{O}(N^{1.5})$	No
Quick Sort	$\mathcal{O}(N \log(N))$	Yes
Merge Sort	$\mathcal{O}(N \log(N))$	Yes
Heap Sort	$\mathcal{O}(N \log(N))$	Yes
Radix Sort	$\mathcal{O}(N)$	Yes

Best Sorting Algorithm's Runtime Complexity

Sorting Algorithm	Best Case Runtime Complexity	Average Case Runtime Complexity	Worst Case Runtime Complexity
Quicksort	$\mathcal{O}(N \log(N))$	$\mathcal{O}(N \log(N))$	$\mathcal{O}(N^2)$
Merge Sort	$\mathcal{O}(N \log(N))$	$\mathcal{O}(N \log(N))$	$\mathcal{O}(N \log(N))$
Heap Sort	$\mathcal{O}(N)$	$\mathcal{O}(N \log(N))$	$\mathcal{O}(N \log(N))$
Radix Sort	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$

B-Tree, Sets, Midterm

B-Tree, Sets, Midterm

7.0.1 Activities

The following are the activities that are planned for Week 7 of this course.

- Sorting Programming Assignment is Due Wednesday.
- Your midterm exam is open on Friday from 10am-10pm MT, make sure you take both parts of the exam (you don't have to take them back to back necessarily, both exams must be completed by 10pm MT).
- Midterm Exam will cover all the material you have learned so far up to and including Chapter 10 of the book (Chapter 11 and 12 are excluded).
- Read Ch. 11 and Ch. 12 of the Data Structures zyBook and take the in chapter quizzes (due next Monday).
- Start on BTree Programming Assignment (due next Tuesday) - this assignment is EXTRA CREDIT.

7.0.2 Lectures

Here are the lectures that can be found for this week:

- Operation Times
- Indiana Jones & The Temple Of B-Tree
- B-Tree Invariants
- B-Tree Find Operation
- B-Tree Demo
- B-Tree Insert Operation
- B-Tree Remove Operation

The lecture notes for this week are:

- Design & B-Trees Lecture Notes

7.0.3 Programming Assignment

The programming assignment for Week 7 is:

- Programming Assignment 5 - BTrees

7.0.4 Exam

The exam for this week is:

- Exam 1
- Exam 1 Notes

7.0.5 Chapter Summary

The first chapter of this week is **Chapter 11: B-Trees**.

Section 11.1 - B-Trees

Introduction to B-Trees

B-Trees are self-balancing search trees that provide efficient storage and retrieval of data in computer science. They are commonly used in databases and file systems due to their ability to handle large amounts of data and maintain optimal performance. B-Trees have a hierarchical structure with a variable number of child nodes per parent, allowing them to maintain balance and minimize the number of disk accesses required for data retrieval and updates. Their balanced nature ensures efficient search operations, making B-Trees a fundamental data structure for organizing and managing data in various applications. B-Trees adhere to the following rules:

- All keys in a B-tree must be distinct.
- All leaf nodes must be at the same level.
- An internal node with N keys must have N+1 children.
- Keys in a node are stored in sorted order from smallest to largest.
- Each key in a B-tree internal node has one left subtree and one right subtree. All left subtree keys are $<$ that key, and all right subtree keys are $>$ that key.

Higher Order B-Trees

Higher-order B-Trees, also known as B+ Trees, are an extension of traditional B-Trees that optimize for disk access and efficient data retrieval in large-scale databases and file systems. Like B-Trees, they maintain a hierarchical structure, but they differ in the way they handle data storage. Higher-order B-Trees store data only in the leaf nodes, while the internal nodes serve as index nodes, containing key-value pairs that guide the search process. This design allows for faster sequential data access and efficient range queries, as the leaf nodes are linked together in a sorted manner. The higher-order B-Tree's ability to store multiple key-value pairs in a single node, often referred to as fanout, significantly reduces the number of disk accesses required for operations, improving overall performance in scenarios with large datasets.

2-3-4 B-Trees

2-3-4 B-Trees, also known as 2-4 B-Trees, are a balanced search tree variant that efficiently organizes and retrieves data in computer science. They are a multiway tree structure where each internal node can have either two, three, or four child nodes, hence the name. The key characteristic of 2-3-4 B-Trees is that they maintain balance by redistributing keys and nodes during insertions and deletions, ensuring an evenly distributed tree structure. This balance helps optimize search operations by reducing the height of the tree and minimizing the number of disk accesses required. 2-3-4 B-Trees are commonly used in databases and file systems to provide efficient data storage and retrieval, especially in scenarios with large datasets and frequent updates.

B-Trees Example

Below is an example of B-Trees in C++:

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

class BTreeNode {
    std::vector<int> keys;
    std::vector<BTreeNode*> children;
    bool leaf;

public:
    BTreeNode(bool isLeaf) {
        leaf = isLeaf;
    }

    void insert(int key) {
        // Insert key into the appropriate position in the keys vector
        // and adjust the children vector if necessary.
    }

    void traverse() {
```

```
            // Traverse the B-Tree in order and print the keys
        }
    };

    class BTree {
        BTreeNode* root;

    public:
        BTree() {
            root = nullptr;
        }

        void insert(int key) {
            if (root == nullptr) {
                root = new BTreeNode(true);
                root->insert(key);
            } else {
                // Insert key into the B-Tree by traversing the nodes
            }
        }

        void traverse() {
            if (root != nullptr) {
                root->traverse();
            }
        }
    };

    int main() {
        BTree bTree;
        bTree.insert(10);
        bTree.insert(20);
        bTree.insert(5);
        bTree.insert(15);

        bTree.traverse();

        return 0;
    }

```

The provided example presents a basic implementation of a B-Tree in C++, showcasing the insertion and traversal operations. B-Trees are essential data structures used in databases and file systems for efficient storage and retrieval of large datasets. The B-Tree maintains balance and optimizes disk access, making it suitable for managing and organizing data effectively. By recursively traversing nodes and sorting keys, B-Trees enable fast search operations and provide a balanced hierarchical structure for optimal performance in various applications.

Section 11.2 - 2-3-4 B-Tree Search Algorithm

The 2-3-4 B-Tree search algorithm is a method for finding a specific key within a 2-3-4 B-Tree data structure. The algorithm starts at the root node and compares the target key with the keys stored in the node. If the key is found, the search is successful. If the target key is less than the smallest key in the node, the algorithm recursively follows the leftmost child. If the target key is greater than the largest key in the node, the algorithm follows the rightmost child. If the target key falls between two keys in the node, the algorithm recursively follows the child corresponding to the key range in which the target key lies. The process continues until the key is found or a leaf node is reached, indicating the key is not present in the B-Tree. This search algorithm ensures efficient and balanced key retrieval in a 2-3-4 B-Tree, making it suitable for large-scale data storage and retrieval scenarios.

2-3-4 B-Tree Search Algorithm Example

Below is an example of the 2-3-4 B-Tree search algorithm in C++:

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

class BTreeNode {
    std::vector<int> keys;
    std::vector<BTreeNode*> children;
    bool leaf;

public:
    BTreeNode(bool isLeaf) {
        leaf = isLeaf;
    }
```

```
13
14      BTreeNode* search(int key) {
15          int i = 0;
16          while (i < keys.size() && key > keys[i]) {
17              i++;
18          }
19
20          if (keys[i] == key) {
21              return this;
22          }
23
24          if (leaf) {
25              return nullptr;
26          }
27
28          return children[i]->search(key);
29      }
30  };
31
32  class BTree {
33      BTreeNode* root;
34
35  public:
36      BTree() {
37          root = nullptr;
38      }
39
40      BTreeNode* search(int key) {
41          if (root == nullptr) {
42              return nullptr;
43          } else {
44              return root->search(key);
45          }
46      }
47  };
48
49  int main() {
50      BTree bTree;
51      // Populate the B-Tree with keys
52
53      // Example keys
54      std::vector<int> keys = {10, 20, 30, 40, 50};
55
56      // Insert keys into the B-Tree
57      for (int key : keys) {
58          bTree.insert(key);
59      }
60
61      int targetKey = 30;
62      BTreeNode* result = bTree.search(targetKey);
63
64      if (result != nullptr) {
65          std::cout << "Key " << targetKey
66          << " found in the B-Tree!" << std::endl;
67      } else {
68          std::cout << "Key " << targetKey
69          << " not found in the B-Tree." << std::endl;
70      }
71
72      return 0;
73  }
74
```

The example showcases the implementation and utilization of the 2-3-4 B-Tree search algorithm in C++. By traversing the nodes and comparing keys, the algorithm enables efficient search for a specific key within the B-Tree structure. The code demonstrates the process by populating a B-Tree with keys and performing a search for a target key. This algorithm is useful for retrieving data from large datasets, making 2-3-4 B-Trees an essential data structure for efficient data organization and management in diverse applications.

Section 11.3 - 2-3-4 B-Tree Insert Algorithm

2-3-4 B-Tree Insertions & Split Operations

2-3-4 B-Tree insertions and split operations play a crucial role in maintaining the balance and integrity of the B-Tree structure. When inserting a new key into a 2-3-4 B-Tree, the algorithm begins by finding the appropriate leaf node where the key should be placed. If the node has room for the new key, it is simply inserted in sorted order.

However, if the node is already full, a split operation is performed. During a split operation, the node is divided into two, with the middle key being promoted to the parent node. The left side contains the smaller keys, and the right side contains the larger keys. The parent node then adjusts its child pointers accordingly. If the parent node is also full, the split operation cascades up the tree until a suitable node with space is found or a new root is created. These insertions and split operations maintain the balance and ensure optimal performance of the 2-3-4 B-Tree, enabling efficient storage and retrieval of data in various applications.

Split Operation Algorithm

The split operation algorithm in the context of 2-3-4 B-Trees is responsible for maintaining the balance and structure of the tree when a node becomes full during an insertion. The algorithm starts by identifying the middle key of the full node, which will be promoted to the parent node. Then, the node is split into two separate nodes: one containing the smaller keys and the other containing the larger keys. The child pointers of the original node are appropriately redistributed between the two new nodes. If the parent node is also full, the split operation cascades up the tree until a suitable node with available space is found or a new root is created. This split operation ensures that the 2-3-4 B-Tree remains balanced and maintains its efficient search and retrieval capabilities when handling large amounts of data.

Inserting A Key Into A Leaf Node

When inserting a key into a leaf node in the context of a 2-3-4 B-Tree, the algorithm first finds the appropriate leaf node where the key should be placed. If the leaf node has space to accommodate the new key, it is simply inserted in sorted order. However, if the leaf node is already full, a split operation is triggered. During the split operation, the leaf node is divided into two, with the middle key being promoted to the parent node. The new key is then inserted in the appropriate sorted position in either the left or right split node. This ensures that the leaf nodes remain balanced and maintain the sorted order of keys, preserving the efficient search and retrieval characteristics of the 2-3-4 B-Tree data structure.

B-Tree Insert With Preimptive Split

A B-Tree Insert with preemptive split is an optimization technique used during key insertion in B-Trees to proactively prevent potential future splits. In this context, when inserting a key, the algorithm first traverses the tree to find the appropriate leaf node for insertion. However, if the leaf node is full, instead of immediately splitting the node, a preemptive split is performed. This involves splitting the node into two separate nodes, with the middle key being promoted to the parent node. The new key to be inserted is then compared to the promoted key. If it is smaller, it is inserted into the left split node, and if it is larger, it is inserted into the right split node. This preemptive split optimizes the B-Tree's structure in advance, minimizing the need for future splits and ensuring efficient storage and retrieval operations in scenarios with frequent insertions.

2-3-4 B-Tree Insert Algorithm Example

Below is an example of the 2-3-4 B-Tree insert algorithm in C++:

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

class BTreeNode {
    std::vector<int> keys;
    std::vector<BTreeNode*> children;
    bool leaf;

public:
    BTreeNode(bool isLeaf) {
        leaf = isLeaf;
    }

    void insertNonFull(int key) {
        int i = keys.size() - 1;
        if (leaf) {
            keys.push_back(0);
            while (i >= 0 && key < keys[i]) {
                keys[i + 1] = keys[i];
                i--;
            }
            keys[i + 1] = key;
        } else {
            while (i >= 0 && key < keys[i]) {
                i--;
            }
            if (children[i + 1]->keys.size() == 3) {
                splitChild(i + 1, children[i + 1]);
                if (key > keys[i + 1]) {
                    i++;
                }
            }
```

```
            children[i + 1]->insertNonFull(key);
        }
    }

    void splitChild(int index, BTreeNode* child) {
        BTreeNode* newNode = new BTreeNode(child->leaf);
        keys.insert(keys.begin() + index, child->keys[1]);
        child->keys.resize(1);

        if (!child->leaf) {
            newNode->children.assign(child->children.begin() + 2,
            child->children.end());
            child->children.resize(2);
        }

        children.insert(children.begin() + index + 1, newNode);
    }
};

class BTree {
    BTreeNode* root;

public:
    BTree() {
        root = nullptr;
    }

    void insert(int key) {
        if (root == nullptr) {
            root = new BTreeNode(true);
            root->keys.push_back(key);
        } else {
            if (root->keys.size() == 3) {
                BTreeNode* newNode = new BTreeNode(false);
                newNode->children.push_back(root);
                root = newNode;
                newNode->splitChild(0, root);
                newNode->insertNonFull(key);
            } else {
                root->insertNonFull(key);
            }
        }
    }
};

int main() {
    BTree bTree;
    // Insert keys into the B-Tree
    bTree.insert(10);
    bTree.insert(20);
    bTree.insert(5);
    bTree.insert(15);

    // Perform operations on the B-Tree

    return 0;
}

```

The example showcases the implementation of the 2-3-4 B-Tree Insert algorithm with preemptive split in C++. The code demonstrates how keys are inserted into the B-Tree, with preemptive splits applied to prevent future overflows. The algorithm ensures that the B-Tree remains balanced and optimized for efficient data storage and retrieval. The example provides a basic framework for creating and manipulating a 2-3-4 B-Tree, enabling the insertion of keys and potential splits as necessary. This algorithm and implementation are essential for maintaining the integrity and performance of B-Trees when handling large datasets and frequent insertions.

Section 11.4 - 2-3-4 B-Tree Rotations & Fusion

Rotation Concepts

Rotation concepts in the context of 2-3-4 trees refer to the techniques used to maintain the balance and structure of the tree when inserting or deleting nodes. Rotations involve reorganizing the nodes by adjusting their positions and relationships. In the case of 2-3-4 trees, rotations include left rotations, right rotations, and fusion. Left and right rotations involve rotating nodes within a single level of the tree, while fusion involves merging two nodes

into a single node. These rotation operations ensure that the tree remains balanced and adheres to the rules of a 2-3-4 tree, optimizing data storage and retrieval operations by avoiding excessive height and promoting efficient key distribution.

2-3-4 B-Tree Rotation Example

Below is an example of a 2-3-4 B-Tree rotation in C++:

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

class TreeNode {
    std::vector<int> keys;
    std::vector<TreeNode*> children;

public:
    TreeNode() {}

    bool isLeaf() {
        return children.empty();
    }

    void leftRotate(int index) {
        TreeNode* child = children[index];
        TreeNode* sibling = children[index + 1];

        // Moving keys and children
        child->keys.push_back(keys[index]);
        keys[index] = sibling->keys[0];
        sibling->keys.erase(sibling->keys.begin());

        if (!child->isLeaf()) {
            child->children.push_back(sibling->children[0]);
            sibling->children.erase(sibling->children.begin());
        }
    }
};

int main() {
    TreeNode* root = new TreeNode();

    // Insert keys and children into the tree

    // Perform a left rotation
    root->leftRotate(0);

    // Perform other operations on the tree

    return 0;
}

```

The provided example illustrates the implementation of a left rotation operation in the context of a 2-3-4 tree using C++. Left rotations play a vital role in maintaining the balance and structure of the tree by redistributing keys and children between nodes. This rotation operation ensures that the tree remains balanced and adheres to the properties of a 2-3-4 tree, promoting efficient data storage and retrieval. The example provides a framework for creating and manipulating a 2-3-4 tree, showcasing the usage of left rotations and their impact on the tree's structure.

Fusion

Fusion, in the context of a 2-3-4 tree, refers to the operation of merging two adjacent nodes into a single node. When fusion occurs, a key from the parent node is moved down to the merged node, effectively combining the keys and children of the two nodes. This process helps maintain the balance of the tree and ensures that the number of keys in each node adheres to the rules of a 2-3-4 tree. Fusion is typically performed when deleting a key from a node, and the resulting node becomes underfull. By fusing nodes, the tree avoids excessive height and optimizes its structure for efficient data storage and retrieval operations.

Non-Root Fusion

In the context of a 2-3-4 tree, non-root fusion refers to the fusion operation performed on a non-root node when deleting a key that causes the node to become underfull. When a key is deleted from a non-root node and the resulting node has fewer keys than the minimum required, a fusion operation is triggered. In this operation, the underfull node fuses with its adjacent sibling node by transferring a key from the parent node into the merged node. The keys and children of the two nodes are combined, and the parent node is updated accordingly. Non-root fusion helps maintain the balance and structure of the tree while ensuring that the number of keys in each node remains within the required range. By performing non-root fusions, the 2-3-4 tree optimizes its structure for efficient data storage and retrieval operations.

2-3-4 B-Tree Fusion Example

Below is an example of a 2-3-4 B-Tree fusion in C++:

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

class TreeNode {
    std::vector<int> keys;
    std::vector<TreeNode*> children;

public:
    TreeNode() {}

    bool isLeaf() {
        return children.empty();
    }

    void fuseNodes(int index) {
        TreeNode* leftChild = children[index];
        TreeNode* rightChild = children[index + 1];

        // Move key from parent to leftChild
        leftChild->keys.push_back(keys[index]);

        // Move keys from rightChild to leftChild
        for (const auto& key : rightChild->keys) {
            leftChild->keys.push_back(key);
        }

        // Move children from rightChild to leftChild
        for (const auto& child : rightChild->children) {
            leftChild->children.push_back(child);
        }

        // Erase key and rightChild
        keys.erase(keys.begin() + index);
        children.erase(children.begin() + index + 1);

        delete rightChild;
    }
};

int main() {
    TreeNode* root = new TreeNode();

    // Insert keys and children into the tree

    // Perform a fusion operation
    root->fuseNodes(1);

    // Perform other operations on the tree

    return 0;
}

```

The provided example demonstrates the implementation of a fusion operation in the context of a 2-3-4 tree using C++. Fusion operations are essential for maintaining the balance and structure of the tree by merging adjacent nodes into a single node. This process redistributes keys and children, ensuring that the tree remains balanced and adheres to the properties of a 2-3-4 tree. By performing fusions, the tree optimizes its structure for efficient data storage and retrieval. The example provides a framework for creating and manipulating a 2-3-4 tree, showcasing the usage of fusion operations and their impact on the tree's structure.

Section 11.5 - 2-3-4 Tree Removal

Merge Algorithm

The merge algorithm for a 2-3-4 B-Tree refers to the process of merging two adjacent nodes in the tree when they become underfull. When a node is underfull, meaning it has fewer keys than the minimum required, the merge algorithm is invoked. In this algorithm, the underfull node combines with its adjacent sibling node by moving a key from the parent node into the merged node. The keys and children of the two nodes are merged, forming a single node with a reduced height. The parent node is then updated to remove the key and the reference to the

sibling node. The merge algorithm helps maintain the balanced structure of the 2-3-4 B-Tree, optimizing its storage efficiency and preserving the efficient search and retrieval properties of the data structure.

2-3-4 B-Tree Merge Algorithm Example

Below is an example of the 2-3-4 B-Tree merge algorithm in C++:

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

class TreeNode {
    std::vector<int> keys;
    std::vector<TreeNode*> children;

public:
    TreeNode() {}

    bool isLeaf() {
        return children.empty();
    }

    void mergeNodes(int index) {
        TreeNode* leftChild = children[index];
        TreeNode* rightChild = children[index + 1];

        // Move key from parent to leftChild
        leftChild->keys.push_back(keys[index]);

        // Move keys from rightChild to leftChild
        for (const auto& key : rightChild->keys) {
            leftChild->keys.push_back(key);
        }

        // Move children from rightChild to leftChild
        for (const auto& child : rightChild->children) {
            leftChild->children.push_back(child);
        }

        // Erase key and rightChild
        keys.erase(keys.begin() + index);
        children.erase(children.begin() + index + 1);

        delete rightChild;
    }
};

int main() {
    TreeNode* root = new TreeNode();

    // Insert keys and children into the tree

    // Perform a merge operation
    root->mergeNodes(1);

    // Perform other operations on the tree

    return 0;
}

```

The provided example demonstrates the implementation of the merge algorithm for a 2-3-4 B-Tree using C++. The merge algorithm is essential for maintaining the balanced structure of the tree by merging underfull nodes with their adjacent sibling nodes. By combining the keys and children of the nodes, the merge operation reduces the height of the tree and ensures that the number of keys in each node adheres to the requirements of the 2-3-4 B-Tree. This algorithm optimizes the storage efficiency and preserves the efficient search and retrieval properties of the data structure. The example provides a framework for creating and manipulating a 2-3-4 B-Tree, showcasing the usage of the merge algorithm and its impact on the tree's structure.

Remove Algorithm

The remove algorithm in a 2-3-4 B-Tree refers to the process of removing a key from the tree while maintaining its balanced structure. When a key is to be removed, the remove algorithm is invoked. The algorithm first finds the node containing the key and removes it. If the removed key is from an internal node, it is replaced with the largest key from the left child or the smallest key from the right child, ensuring that the tree properties are preserved. If the removal causes an underfull node, fusion or redistribution operations may be performed to maintain the balance of the tree. The remove algorithm in a 2-3-4 B-Tree ensures that the tree remains balanced and optimized for efficient data storage and retrieval operations even after key removal.

2-3-4 Remove Algorithm Example

Below is an example of the 2-3-4 B-Tree remove algorithm in C++:

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

class TreeNode {
    std::vector<int> keys;
    std::vector<TreeNode*> children;

public:
    TreeNode() {}

    bool isLeaf() {
        return children.empty();
    }

    void removeKey(int key) {
        int keyIndex = findKeyIndex(key);

        if (keyIndex != -1) {
            // Key found in current node
            if (isLeaf()) {
                // Remove key from leaf node
                keys.erase(keys.begin() + keyIndex);
            } else {
                // Replace key with predecessor or successor
                TreeNode* predecessorChild = children[keyIndex];
                TreeNode* successorChild = children[keyIndex + 1];

                if (predecessorChild->keys.size() >= 2) {
                    int predecessorKey = removePredecessor(predecessorChild);
                    keys[keyIndex] = predecessorKey;
                } else if (successorChild->keys.size() >= 2) {
                    int successorKey = removeSuccessor(successorChild);
                    keys[keyIndex] = successorKey;
                } else {
                    // Fusion operation
                    mergeNodes(keyIndex);
                    children[keyIndex]->removeKey(key);
                }
            }
        } else {
            // Key not found in current node
            int childIndex = findChildIndex(key);
            if (childIndex != -1) {
                // Recursively remove key from appropriate child node
                children[childIndex]->removeKey(key);
            }
        }
    }

    int removePredecessor(TreeNode* node) {
        if (node->isLeaf()) {
            int predecessorKey = node->keys.back();
            node->keys.pop_back();
            return predecessorKey;
        } else {
            return removePredecessor(node->children.back());
        }
    }

    int removeSuccessor(TreeNode* node) {
        if (node->isLeaf()) {
            int successorKey = node->keys.front();
            node->keys.erase(node->keys.begin());
            return successorKey;
        } else {
            return removeSuccessor(node->children.front());
        }
    }

    void mergeNodes(int index) {
        // Merge current node with adjacent sibling
        // (implementation of the merge algorithm)
    }

    int findKeyIndex(int key) {
        // Find index of the key in the current node
        // Return -1 if key not found
    }

    int findChildIndex(int key) {
        // Find index of the child node where the key might be located
        // Return -1 if appropriate child not found
    }
};

int main() {
    TreeNode* root = new TreeNode();

```

```
89          // Insert keys and children into the tree
90
91          // Perform a remove operation
92          root->removeKey(10);
93
94          // Perform other operations on the tree
95
96          return 0;
97      }
98
```

The remove algorithm in a 2-3-4 B-Tree handles the deletion of keys while maintaining the tree's balance. It searches for the key in the current node and removes it from a leaf node directly or replaces it with a predecessor or successor in an internal node. If there are insufficient keys in the predecessor or successor, fusion operations are performed. If the key is not found in the current node, the algorithm recursively searches for the appropriate child node to remove the key. The remove algorithm ensures a balanced structure and optimized data storage and retrieval by maintaining the required properties of the 2-3-4 B-Tree.

The second chapter of this is **Chapter 12: Sets**.

Section 12.1 - Set Abstract Data Type

Set Abstract Data Type

The set abstract data type represents a collection of unique elements without any particular order or associated values. It provides operations for adding elements to the set, removing elements from it, checking if an element is present, and performing common set operations such as union, intersection, and difference. The set ADT is characterized by its ability to efficiently handle membership queries and perform set operations in near-constant time, making it a valuable tool in various applications involving data manipulation, mathematical modeling, and algorithm design.

Element Keys & Removal

In the context of sets, element keys refer to the values that uniquely identify each element within the set. Unlike other data structures like dictionaries or maps, sets do not have associated keys or values. Instead, elements in a set are considered unique based on their inherent value or properties. When removing elements from a set, the operation is typically performed based on the value of the element itself rather than a specific key. This allows for efficient removal of elements without the need for key-based lookup or retrieval. The absence of keys simplifies the implementation and operations of sets, making them a convenient choice for managing collections of distinct elements.

Searching & Subsets

In the context of sets, searching refers to the process of determining whether a particular element is present in the set. Set implementations typically provide efficient search operations, allowing for constant-time lookup. By utilizing appropriate data structures like hash sets or binary search trees, sets can quickly determine membership, indicating whether an element exists within the set or not. On the other hand, subsets in the context of sets involve determining whether one set is entirely contained within another set. This operation checks if all elements of one set are also present in another set. Subset operations can be performed by iterating over the elements of one set and checking their presence in the other set, which can be done efficiently by leveraging appropriate data structures and algorithms. The ability to search for elements and determine subsets makes sets valuable for various applications, such as data filtering, data validation, and solving combinatorial problems.

Sets Example

Below is an example of sets in C++:

```
1  #include <iostream>
2  #include <unordered_set>
3
4  int main() {
5      std::unordered_set<int> set1 = {1, 2, 3, 4, 5};
6      std::unordered_set<int> set2 = {3, 4, 5};
```

```
    std::unordered_set<int> set3 = {6, 7, 8};

    // Searching
    int element = 3;
    if (set1.find(element) != set1.end()) {
        std::cout << "Element " << element << " is present in set1."
        << std::endl;
    } else {
        std::cout << "Element " << element << " is not present in set1."
        << std::endl;
    }

    // Subsets
    if (std::includes(set1.begin(), set1.end(), set2.begin(), set2.end())) {
        std::cout << "set2 is a subset of set1." << std::endl;
    } else {
        std::cout << "set2 is not a subset of set1." << std::endl;
    }

    if (std::includes(set1.begin(), set1.end(), set3.begin(), set3.end())) {
        std::cout << "set3 is a subset of set1." << std::endl;
    } else {
        std::cout << "set3 is not a subset of set1." << std::endl;
    }

    return 0;
}
```

In this example, we use the 'unordered_set' container from the C++ Standard Library to represent sets. First, we create 'set1', 'set2', and 'set3' with different elements. We then demonstrate searching by checking if an element ('3' in this case) exists in 'set1' using the 'find()' function. If the element is found, we output that it is present; otherwise, we output that it is not present. Next, we showcase subsets by using the 'includes()' algorithm from the Standard Library. It takes the iterators of the first and last elements of two ranges and checks if the first range ('set1') includes all elements of the second range ('set2' and 'set3'). We output whether each set is a subset or not. This example demonstrates the searching capability of sets by finding specific elements and the subset functionality by checking if one set is a subset of another.

Section 12.2 - Set Operations

Union, Intersection, & Difference

In the context of sets, union, intersection, and difference are fundamental operations used to manipulate and combine sets. The union of two sets results in a new set that contains all unique elements from both sets. It combines the elements of both sets without duplication. The intersection of two sets yields a new set that contains only the elements that are common to both sets. It represents the overlap or shared elements between the sets. Lastly, the difference operation returns a new set that contains the elements present in one set but not in the other. It represents the elements that are unique to each set. These set operations provide powerful tools for data analysis, merging datasets, filtering, and solving various mathematical and logical problems involving sets.

Filter & Map

In the context of sets, filters and maps are operations used to transform and manipulate the elements within a set. The filter operation applies a condition or predicate to each element of the set and returns a new set that includes only the elements satisfying the condition. It allows for selective extraction or removal of elements based on specific criteria. On the other hand, the map operation applies a transformation function to each element of the set and generates a new set containing the transformed elements. It enables the modification or conversion of each element according to a defined rule. Filters and maps provide flexible ways to extract relevant information, perform data transformations, and derive new sets from existing ones, making them useful tools in data processing, data cleaning, and data analysis tasks involving sets.

Set Operations Example

Below is an example of set operations in C++:

```
#include <iostream>
```

```
#include <unordered_set>
#include <algorithm>

int main() {
    std::unordered_set<int> originalSet = {1, 2, 3, 4, 5};

    // Filter: Keep only even numbers
    std::unordered_set<int> filteredSet;
    std::copy_if(originalSet.begin(), originalSet.end(),
    std::inserter(filteredSet, filteredSet.begin()), [](int num) {
        return num % 2 == 0;
    });

    // Map: Multiply each element by 2
    std::unordered_set<int> mappedSet;
    std::transform(originalSet.begin(), originalSet.end(),
    std::inserter(mappedSet, mappedSet.begin()), [](int num) {
        return num * 2;
    });

    // Output the filtered set
    std::cout << "Filtered Set: ";
    for (const auto& element : filteredSet) {
        std::cout << element << " ";
    }
    std::cout << std::endl;

    // Output the mapped set
    std::cout << "Mapped Set: ";
    for (const auto& element : mappedSet) {
        std::cout << element << " ";
    }
    std::cout << std::endl;

    return 0;
}

```

In this example, we start with an original set containing integers. We demonstrate the filter operation by creating a new set called 'filteredSet'. We use 'copy_if()' algorithm from the C++ Standard Library to iterate over the original set and only copy elements that satisfy a condition. In this case, we keep only even numbers in the filtered set by using a lambda function as the predicate.

Next, we showcase the map operation by creating another set called 'mappedSet'. We utilize the 'transform()' algorithm to iterate over the original set and apply a transformation function to each element. The lambda function multiplies each element by 2, generating a new set with the transformed values. Finally, we output the filtered set and mapped set to verify the results. The filtered set contains only the even numbers from the original set, while the mapped set contains all elements of the original set multiplied by 2.

This example demonstrates how filters and maps can be applied to sets to selectively extract elements based on a condition or transform the elements according to a defined rule, allowing for flexible data manipulation and processing.

Section 12.3 - Static & Dynamic Set Operations

Static and dynamic sets are two different approaches to representing sets in computer science. A static set refers to a fixed-size set where the number of elements and their values cannot change once the set is defined. It is typically implemented using arrays or fixed-size data structures. Static sets are efficient in terms of memory usage and provide fast access to elements but lack flexibility in terms of dynamic operations. On the other hand, dynamic sets are resizable sets that can grow or shrink in size during program execution. They are typically implemented using linked lists, dynamic arrays, or tree structures. Dynamic sets allow for efficient insertion, deletion, and modification of elements but may require additional memory overhead and have slightly slower access times compared to static sets. The choice between static and dynamic sets depends on the specific requirements of the problem and the trade-offs between memory efficiency and dynamic operations.

Static & Dynamic Set Operations Example

Below is an example of static and dynamic set operations in C++:

```
#include <iostream>

// Static Set
const int MAX_SIZE = 5;
```

```
    int staticSet[MAX_SIZE] = {1, 2, 3, 4, 5};

    // Dynamic Set
    class DynamicSet {
    private:
        int* elements;
        int size;
        int capacity;

    public:
        DynamicSet() {
            capacity = 5;
            size = 0;
            elements = new int[capacity];
        }

        void insert(int element) {
            if (size < capacity) {
                elements[size++] = element;
            }
        }

        void remove(int element) {
            for (int i = 0; i < size; i++) {
                if (elements[i] == element) {
                    for (int j = i; j < size - 1; j++) {
                        elements[j] = elements[j + 1];
                    }
                    size--;
                    break;
                }
            }
        }

        void printSet() {
            for (int i = 0; i < size; i++) {
                std::cout << elements[i] << " ";
            }
            std::cout << std::endl;
        }
    };

    int main() {
        // Static Set
        std::cout << "Static Set: ";
        for (int i = 0; i < MAX_SIZE; i++) {
            std::cout << staticSet[i] << " ";
        }
        std::cout << std::endl;

        // Dynamic Set
        DynamicSet dynamicSet;
        dynamicSet.insert(1);
        dynamicSet.insert(2);
        dynamicSet.insert(3);
        dynamicSet.insert(4);
        dynamicSet.insert(5);
        dynamicSet.remove(3);
        std::cout << "Dynamic Set: ";
        dynamicSet.printSet();

        return 0;
    }

```

In this example, we demonstrate static and dynamic sets. The static set is implemented using a fixed-size array, where the elements are defined at compile-time and cannot be modified. We initialize the static set with five elements (1, 2, 3, 4, 5) and simply print the elements.

On the other hand, the dynamic set is implemented using a class called 'DynamicSet'. It uses a dynamic array to store the elements, allowing for resizing and modification. The class provides methods to insert and remove elements from the set, as well as print the set. In the 'main()' function, we create a dynamic set, insert five elements (1, 2, 3, 4, 5), remove the element 3, and print the resulting set.

This example demonstrates the difference between static and dynamic sets. The static set has a fixed size and its elements are defined statically, while the dynamic set is resizable and allows for dynamic insertion and removal of elements.

Priority Queue, Heap, Treap

Priority Queue, Heap, Treap

8.0.1 Activities

The following are the activities that are planned for Week 8 of this course.

- Read Chapter 13 in zyBooks and work on book chapter activities. (Due Next Monday)
- BTree homework due Tuesday (Extra Credit)
- Watch video lectures
- Work on Priority Queue homework (Due Next Tuesday)

8.0.2 Lectures

Here are the lectures that can be found for this week:

- Intro to Abstract Data Types
- ADT Examples
- Commond ADT Operations
- Priority Queues
- Heaps
- Treaps

8.0.3 Programming Assignment

The programming assignment for Week 8 is:

- Programming Assignment 6 - Priority Queue

8.0.4 Chapter Summary

The chapter of this week is **Chapter 13: Heaps & Treaps**.

Section 13.1 - Heaps

Overview

Heaps are a fundamental data structure in computer science used for efficient management of priority queues. A heap is a binary tree where each node satisfies the heap property, which means that the value of each node is greater (or smaller) than or equal to the values of its children. Heaps can be implemented as arrays, allowing constant time access to the maximum (or minimum) element. They are commonly used in algorithms such as heap sort, Dijkstra's algorithm, and priority queue implementations, providing efficient operations for inserting, deleting, and retrieving the highest (or lowest) priority element.

Max Heap

Max heaps are a type of heap data structure where the value of each node is greater than or equal to the values of its children. They are typically implemented using an array, with the root node being the largest element. The insert operation in a max heap involves adding a new element to the bottom of the heap and then percolating it up by comparing it with its parent and swapping if necessary until the heap property is satisfied. The remove operation removes the maximum element from the heap, which is always the root node, and replaces it with the last element in the array. The element is then percolated down by comparing it with its children and swapping if necessary to maintain the heap property. Percolating refers to the process of moving an element up or down the heap to maintain the heap property after an insert or remove operation.

Min Heap

Min heaps are a type of heap data structure where the value of each node is smaller than or equal to the values of its children. Similar to max heaps, min heaps are often implemented using an array, with the root node being the smallest element. The insert operation in a min heap involves adding a new element to the bottom of the heap and percolating it up by comparing it with its parent and swapping if necessary until the heap property is satisfied. The remove operation removes the minimum element from the heap, which is always the root node, and replaces it with the last element in the array. The element is then percolated down by comparing it with its children and swapping if necessary to maintain the heap property. Percolating refers to the process of moving an element up or down the heap to maintain the heap property after an insert or remove operation.

Section 13.2 - Heaps Using Arrays

Overview

Heaps can be efficiently stored in arrays, providing a compact representation and enabling constant time access to elements. In a binary heap, the elements are stored in a complete binary tree structure, where each level is filled from left to right, except for the last level which may be partially filled from the left. The heap property is maintained by carefully arranging the elements in the array based on their relationships with their parent and children nodes. The parent-child relationship is determined by the indices of the array, where for any given index i, the left child is located at index $2i + 1$ and the right child is at index $2i + 2$. Conversely, for any child node at index j, its parent can be found at index $\textbf{floor}((j - 1)/2)$. This array representation allows for efficient heap operations such as insert, remove, and percolating, by performing simple arithmetic computations on indices to navigate and manipulate the array elements.

Percolate Algorithm

The percolate algorithm in heaps refers to the process of moving an element up or down the heap to maintain the heap property after an insert or remove operation. When an element is inserted or removed, it may violate the heap property, which states that the value of each node should be greater (in the case of a max heap) or smaller (in the case of a min heap) than or equal to the values of its children. Percolation involves comparing the element with its parent (in the case of percolating up) or with its children (in the case of percolating down) and swapping them if necessary. This process is repeated iteratively until the heap property is satisfied throughout the entire heap, ensuring the integrity and efficiency of heap operations.

Percolate Algorithm Example

Percolate Up

Below is an example of the percolate up algorithm in a max heap in the context of C++:

```
void maxHeapPercolateUp(int arr[], int index) {
    int parent = (index - 1) / 2;
    while (index > 0 && arr[index] > arr[parent]) {
        std::swap(arr[index], arr[parent]);
        index = parent;
        parent = (index - 1) / 2;
    }
}

```

Percolate Down

Conversely, below is an example of the percolate down algorithm in a max heap in the context of C++:

```
void maxHeapPercolateDown(int arr[], int index, int size) {
    int maxIndex = index;
    int leftChild = 2 * index + 1;
    int rightChild = 2 * index + 2;

    if (leftChild < size && arr[leftChild] > arr[maxIndex]) {
        maxIndex = leftChild;
    }

    if (rightChild < size && arr[rightChild] > arr[maxIndex]) {
        maxIndex = rightChild;
    }

    if (maxIndex != index) {
        std::swap(arr[index], arr[maxIndex]);
        maxHeapPercolateDown(arr, maxIndex, size);
    }
}

```

In max heaps, the percolate up algorithm compares an element with its parent and swaps them if the element is greater, ensuring it moves up to its correct position. This process is repeated until the element is in its proper place or reaches the root. Conversely, the percolate down algorithm compares an element with its children, selecting the larger child and swapping them if the child is greater, allowing the element to move down to its appropriate position. This continues until the element is in its correct place or has no greater children. These algorithms maintain the max heap property, facilitating efficient operations like insertions, removals, and maintaining the proper ordering of elements based on their values.

Section 13.3 - Heap Sort

Heapify

The heapify operation is used to convert an array into a heap, either max heap or min heap, in linear time. It rearranges the elements of the array, transforming it into a complete binary tree that satisfies the heap property. The operation begins by selecting the last non-leaf node in the array and performing a percolate down operation on each node, starting from this node and moving upwards towards the root. This ensures that every subtree rooted at a particular node is a valid heap. By repeating this process for all non-leaf nodes, the entire array is transformed

into a heap. The heapify operation is efficient and commonly employed to build a heap from an unordered array or to restore the heap property after an element is modified.

Heapify Algorithm Example

Below is an example of the heapify algorithm in the context of C++:

```cpp
void maxHeapify(int arr[], int n, int i) {
    int largest = i;
    int leftChild = 2 * i + 1;
    int rightChild = 2 * i + 2;

    if (leftChild < n && arr[leftChild] > arr[largest])
        largest = leftChild;

    if (rightChild < n && arr[rightChild] > arr[largest])
        largest = rightChild;

    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        maxHeapify(arr, n, largest);
    }
}

```

The above is an example of the heapify algorithm implementation in C++.

Heap Sort

Heap sort is an efficient comparison-based sorting algorithm that utilizes the properties of a heap data structure. The algorithm begins by building a max heap from the given array, transforming it into a partially sorted structure. Next, it repeatedly extracts the maximum element from the heap (which is always the root), swaps it with the last element of the heap, and reduces the heap size. This process ensures that the maximum element "bubbles down" to its correct position. By iteratively applying this step, the array becomes fully sorted in ascending order. Heap sort has a time complexity of $\mathcal{O}(n \log(n))$, where n is the number of elements in the array, making it an optimal choice for large datasets.

Heap Sort Algorithm Example

Below is an example of the heap sort algorithm in the context of C++:

```cpp
void heapSort(int arr[], int n) {
    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        maxHeapify(arr, n, i);

    // Extract elements from the heap one by one
    for (int i = n - 1; i > 0; i--) {
        std::swap(arr[0], arr[i]);
        maxHeapify(arr, i, 0);
    }
}

```

The above is an example of the heap sort algorithm implementation in C++.

Section - 13.4 Priority Queue Abstract Data Type (ADT)

Overview

The Priority Queue Abstract Data Type (ADT) is a fundamental data structure that allows elements to be stored with associated priority values. It operates on the principle of priority, where elements with higher priority are given precedence over elements with lower priority. The key operations supported by a priority queue include inserting elements with their respective priorities, accessing or retrieving the element with the highest priority, and removing the element with the highest priority. Priority queues can be implemented using various data structures such as heaps, arrays, or linked lists, and they find applications in algorithms where efficient management of priorities is crucial, such as task scheduling, graph algorithms, and event-driven simulations. The priority queue ADT provides

a flexible and efficient way to handle elements based on their priorities, allowing for efficient and ordered processing of tasks or events.

Common Priority Queue Operations

Common priority queue operations include inserting an element with its associated priority, accessing or retrieving the element with the highest priority, and removing the element with the highest priority. Insertion involves adding an element to the priority queue while maintaining the order based on its priority. Accessing the highest priority element allows retrieving the element without removing it from the queue. Removal of the highest priority element removes and returns the element with the highest priority from the priority queue, ensuring that the next highest priority element becomes the new highest priority. These operations are the key building blocks for managing priorities and efficiently processing elements in a priority queue data structure.

Operation	Description
Dequeue(PQueue)	Returns and removes the item at the front of PQueue.
Enqueue(PQueue, x)	Inserts x after all equal or higher priority items.
GetLength(PQueue)	Returns the number of items in PQueue.
IsEmpty(PQueue)	Returns true if PQueue has no items.
Peek(PQueue)	Returns but does not remove the item at the front of PQueue.

Enqueueing With Priority

Enqueueing items with priority refers to the process of adding elements to a data structure, such as a priority queue, while associating each element with a specific priority value. This operation involves inserting the element into the data structure in a way that preserves the order based on the assigned priorities. Elements with higher priorities are typically placed ahead of elements with lower priorities, ensuring that the highest priority element is readily accessible. Enqueueing items with priority is crucial in scenarios where elements need to be processed or accessed based on their importance or urgency, allowing for efficient handling and retrieval of items according to their assigned priorities.

Priority Queue Operation	Heap Functionality	Worst Case Runtime
Enqueue	Insert	$\mathcal{O}(\log(N))$
Dequeue	Remove	$\mathcal{O}(\log(N))$
GetLength	Return number of nodes	$\mathcal{O}(1)$
IsEmpty	Return true if no nodes in heap	$\mathcal{O}(1)$
Peek	Return value of root node	$\mathcal{O}(1)$

Section 13.5 - Treaps

Overview

Treaps are a combination of binary search trees (BSTs) and binary heaps, merging the advantages of both data structures. Each node in a treap holds a key-value pair, with keys following the BST property and priorities assigned randomly. The priority determines the heap property, where the priority of any node is greater than or equal to the priorities of its children. Treaps maintain the probabilistic balance between the BST and heap properties, resulting in a balanced binary search tree with expected logarithmic time complexity for key-based operations. Insertion and deletion in treaps involve performing rotations to maintain the BST and heap properties. Treaps find applications in various scenarios, including priority queues, range queries, and randomized binary search tree implementations.

Search

Search in treaps follows the principles of binary search trees, leveraging the ordering of keys to efficiently locate specific elements. Starting from the root, the search algorithm compares the target key with the key at the current node. If the keys match, the desired element is found. If the target key is smaller, the search continues in the left subtree. If the target key is larger, the search proceeds in the right subtree. This process continues recursively until either the target key is found or a leaf node is reached, indicating that the element does not exist in the treap. The randomized nature of treaps ensures that the search operation maintains an expected logarithmic time complexity, making it an efficient data structure for retrieval operations.

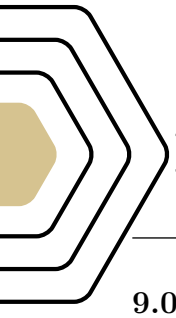
Insert

Insertion in treaps involves adding a new element, represented by a key-value pair, to the treap data structure. The algorithm begins by performing a standard binary search tree insertion, placing the new element in its appropriate position based on the key values. After the element is inserted, the heap property of the treap may be violated since the priorities are assigned randomly. To restore the heap property, rotations are performed to move the newly inserted element upwards until its priority satisfies the heap property. The rotations adjust the tree structure while maintaining the order of the keys. The insertion process in treaps ensures that the resulting data structure remains balanced and offers an expected logarithmic time complexity for subsequent search, delete, and other operations. Treaps provide an efficient way to insert elements while combining the benefits of binary search trees and binary heaps.

Delete

Deletion in treaps involves removing a specific element from the data structure while maintaining the treap's binary search tree and heap properties. The algorithm begins by searching for the element to be deleted, following the same process as the search operation. Once the element is found, it is removed from the treap by performing rotations to restore the binary search tree and heap properties. The rotations reorganize the tree while ensuring that the priorities and keys remain consistent. The exact rotations performed depend on the relative priorities of the nodes involved. The deletion process ensures that the resulting treap remains balanced and preserves the expected logarithmic time complexity for subsequent search and other operations. Treaps provide an efficient way to remove elements from a data structure while maintaining the benefits of both binary search trees and binary heaps.

Hash Tables



Hash Tables

9.0.1 Activities

The following are the activities that are planned for Week 9 of this course.

- Read zyBooks Chapter 14 and do activities (due Monday of Week 10)
- Watch lecture videos
- Homework on Hash Tables (due Tuesday of Week 10)
- Work on Project Proposal (due Thursday of Week 10)

9.0.2 Lectures

Here are the lectures that can be found for this week:

- Hashes
- Hash Implementation
- Collision Mitigation
- Open Addressing
- Hash Tables & Sets

The lecture notes for this week are:

- Hashes Lecture Notes

9.0.3 Programming Assignment

The programming assignment for Week 9 is:

- Programming Assignment 7 - Hash Tables
- Hash Table Interview Notes

9.0.4 Chapter Summary

The chapter of this week is **Chapter 14: Hash Tables**.

Section 14.1 - Hash Tables

Overview

A hash table is a data structure that allows efficient storage and retrieval of key-value pairs. It uses a hash function to convert keys into array indices, providing direct access to values in constant time. The hash function distributes keys uniformly across the array, minimizing collisions. Collisions occur when multiple keys map to the same index, and they are resolved using techniques like chaining or open addressing. Hash tables offer fast insertion, deletion, and retrieval operations, making them ideal for applications that require efficient data lookup, such as symbol tables, caches, and databases.

Operations

Hash tables support efficient operations such as insertion, deletion, and retrieval. During insertion, the key is hashed to determine the index where the value is stored, handling collisions if necessary. Deletion involves locating the key and removing its associated value. Retrieval utilizes the hash function to find the index and returns the value stored at that location. These operations typically have an average time complexity of $\mathcal{O}(1)$, making hash tables effective for fast data storage and retrieval. However, in the worst-case scenario with frequent collisions, the time complexity can degrade to $\mathcal{O}(n)$.

Collisions

Collisions in hash tables occur when multiple keys map to the same index. To handle collisions, hash tables employ techniques like chaining, where a linked list is used to store multiple key-value pairs at the same index. Alternatively, open addressing probes for an alternative location within the array when collisions happen. This can be done using linear probing, quadratic probing, or double hashing. Another approach called Robin Hood Hashing reduces clustering by swapping entries to minimize the difference between the desired and actual positions. These collision resolution methods ensure efficient storage and retrieval of data in hash tables, accommodating varying trade-offs between memory usage and lookup performance.

Section 14.2 - Chaining

Overview

Chaining is a collision resolution technique used in hash tables. In chaining, each bucket or slot in the hash table contains a linked list or another data structure. When a collision occurs and multiple keys map to the same index, the new key-value pairs are appended to the linked list at that index. During retrieval, the hash function is used to locate the bucket, and a linear search or other method is employed within the linked list to find the desired key-value pair. Chaining provides an efficient way to handle collisions by allowing multiple entries to coexist at the same index, ensuring constant-time average performance for insertion, deletion, and retrieval operations in hash tables.

Searching

Searching in a hash table involves applying the hash function to the key to determine the index where the value might be stored. Once the index is identified, the search algorithm follows the collision resolution mechanism implemented in the hash table. In the case of chaining, a linear search or other suitable method is performed within the linked list at the corresponding index to locate the desired key-value pair. This process allows for efficient retrieval of values in constant time, making hash tables an effective data structure for fast searching operations.

Hash Table Search Algorithm

Below is an example of searching in a hash table in the context of C++:

```cpp
// Function to search for a key in the hash table
std::string HashSearch(const std::unordered_map<int, std::list<std::string>>&
                       hashTable, int key) {
    // Find the bucket list corresponding to the key's hash
    auto bucketList = hashTable.at(Hash(key));

    // Search for the key in the bucket list
    auto itemNode = ListSearch(bucketList, key);

    // If the item is found, return its data
    if (itemNode != nullptr) {
        return itemNode->data;
    }
    else {
        return "null";
    }
}

```

Removing

Removing items from a hash table involves locating the key in the table and deleting the associated key-value pair. The removal process typically follows the same steps as searching, where the hash function is used to find the index or bucket where the key might be stored. Once the bucket is identified, the collision resolution mechanism is applied, such as traversing a linked list or probing in open addressing, to locate the specific key-value pair. Once found, the item is removed from the table. Removing items from a hash table ensures efficient deletion operations, with an average time complexity of $\mathcal{O}(1)$.

Hash Table Removing Algorithm

Below is an example of removing in a hash table in the context of C++:

```cpp
// Function to remove an item from the hash table
void HashRemove(std::unordered_map<int, std::list<Item>>& hashTable,
                const Item& item) {
    // Find the bucket list corresponding to the item's key's hash
    auto& bucketList = hashTable[Hash(item.key)];

    // Search for the item in the bucket list
    auto itemNode = ListSearch(bucketList, item.key);

    // If the item is found, remove it from the list
    if (itemNode != nullptr) {
        ListRemove(bucketList, itemNode);
    }
}

```

Inserting

Inserting items into a hash table involves mapping the key to its corresponding index or bucket using a hash function and then storing the key-value pair at that location. The hash function ensures uniform distribution of keys across the hash table, minimizing collisions. If a collision occurs and another key already exists at the computed index, collision resolution techniques like chaining or open addressing are employed to handle it. Insertion in a hash table typically has an average time complexity of $\mathcal{O}(1)$, making it an efficient operation for adding new key-value pairs to the table.

Hash Table Inserting Algorithm

Below is an example of inserting in a hash table in the context of C++:

```cpp
// Function to insert an item into the hash table
void HashInsert(std::unordered_map<int, std::list<Item>>& hashTable,
                const Item& item) {
    // Check if the item's key already exists in the hash table
    if (HashSearch(hashTable, item.key) == nullptr) {
        // Find the bucket list corresponding to the item's key's hash
        auto& bucketList = hashTable[Hash(item.key)];

        // Allocate a new linked list node
        auto node = new Item;
        node->next = nullptr;
        node->data = item;

```

```
            // Append the node to the bucket list
            bucketList.push_back(*node);
        }
    }

```

Section 14.3 - Linear Probing

Overview

Linear probing is a collision resolution technique used in hash tables. When a collision occurs and a key cannot be inserted at its original hashed index, linear probing sequentially searches for the next available slot in the hash table by incrementing the index. The key is inserted at the first unoccupied location found. This process allows for a direct mapping of keys to array indices without the need for additional data structures. However, linear probing can lead to clustering, where consecutive occupied slots form groups, potentially impacting search performance. Despite this drawback, linear probing offers simplicity and cache-friendly behavior, and it can be an effective strategy for resolving collisions in hash tables.

Empty Bucket Types

Empty bucket types, also known as tombstones or placeholders, are used in hash tables as a mechanism to indicate previously occupied slots that have been subsequently emptied. These empty bucket markers serve the purpose of distinguishing between genuinely empty slots and those that were previously occupied but have been deleted or removed. By marking these emptied slots with a special value or flag, the hash table can maintain its integrity during collision resolution and search operations. Empty bucket types allow for efficient re-use of slots, avoiding unnecessary collisions and improving the performance of hash table operations.

Searching

Searching with linear probing is a method used in hash tables to locate a key-value pair. When searching, the hash function is applied to the key to determine the initial index in the array. If the key is not found at that index, linear probing is used to sequentially check the subsequent indices until the key is found or an empty slot is encountered. This approach allows for a direct mapping of keys to array indices, enabling efficient search operations. However, linear probing can lead to clustering, where consecutive occupied slots form groups, potentially impacting search performance due to longer probe sequences. Nonetheless, searching with linear probing remains a simple and widely used technique in hash tables for efficient key retrieval.

Hash Table Linear Probe Search Algorithm

Below is an example of searching in a hash table via linear probing in the context of C++:

```
// Structure for the item in the hash table
struct Item {
    int key;
    std::string data;
};

// Function to search for a key in the hash table using linear probing
const Item* HashSearch(const std::vector<Item>& hashTable, int key) {
    // Hash function determines initial bucket
    size_t bucket = Hash(key);
    size_t bucketsProbed = 0;
    size_t N = hashTable.size();

    while ((hashTable[bucket].status != EmptySinceStart) && (bucketsProbed < N)) {
        if ((hashTable[bucket].status != Empty) && (hashTable[bucket].key == key)) {
            return &hashTable[bucket];
        }

        // Increment bucket index
        bucket = (bucket + 1) % N;

        // Increment number of buckets probed
        ++bucketsProbed;
    }
```

```
25
26      return nullptr;  // Item not found
27  }
28
```

Removal

Removing with linear probing is a technique used in hash tables to delete a key-value pair. When removing an item, the hash function is applied to the key to determine the initial index in the array. If the key is not found at that index, linear probing is used to sequentially search the subsequent indices until the key is found or an empty slot is encountered. Once the key is located, the corresponding value is removed, and the slot is marked as empty. Linear probing ensures that the hash table remains compact and can efficiently handle deletions. However, it can cause clustering and lead to longer probe sequences, potentially impacting the performance of removal operations in hash tables.

Hash Table Linear Probe Removal Algorithm

Below is an example of removing in a hash table via linear probing in the context of C++:

```
1   // Structure for the item in the hash table
2   struct Item {
3       int key;
4       std::string data;
5       bool isEmptyAfterRemoval;
6   };
7
8   // Function to remove an item from the hash table using linear probing
9   void HashRemove(std::vector<Item>& hashTable, int key) {
10      // Hash function determines initial bucket
11      size_t bucket = Hash(key);
12      size_t bucketsProbed = 0;
13      size_t N = hashTable.size();
14
15      while ((hashTable[bucket].status != EmptySinceStart) && (bucketsProbed < N)) {
16          if ((hashTable[bucket].status != Empty) && (hashTable[bucket].key == key)) {
17              hashTable[bucket].isEmptyAfterRemoval = true;
18              return;
19          }
20
21          // Increment bucket index
22          bucket = (bucket + 1) % N;
23
24          // Increment number of buckets probed
25          ++bucketsProbed;
26      }
27  }
28
```

Inserting

Inserting with linear probing is a technique used in hash tables to add a new key-value pair. When inserting, the hash function is applied to the key to determine the initial index in the array. If the index is already occupied, linear probing is used to sequentially search for the next available slot by incrementing the index. The key-value pair is inserted at the first unoccupied location found. Linear probing allows for a direct mapping of keys to array indices and avoids the need for additional data structures. However, it can cause clustering and potentially result in longer probe sequences, which may affect the performance of insertion operations in hash tables. Nonetheless, linear probing remains a simple and widely used strategy for adding elements to hash tables.

Hash Table Linear Probing Insert Algorithm

Below is an example of inserting in a hash table via linear probing in the context of C++:

```
1   // Structure for the item in the hash table
2   struct Item {
3       int key;
4       std::string data;
5   };
6
7   // Function to insert an item into the hash table using linear probing
8   bool HashInsert(std::vector<Item>& hashTable, const Item& item) {
9       // Hash function determines initial bucket
10      size_t bucket = Hash(item.key);
11      size_t bucketsProbed = 0;
12      size_t N = hashTable.size();
13
14      while (bucketsProbed < N) {
15          // Insert item in the next empty bucket
```

```
            if (hashTable[bucket].status == Empty) {
                hashTable[bucket] = item;
                return true;
            }

            // Increment bucket index
            bucket = (bucket + 1) % N;

            // Increment number of buckets probed
            ++bucketsProbed;
        }

        return false; // Unable to insert item
    }

```

Section 14.4 - Quadratic Probing

Overview

Quadratic probing is a collision resolution technique used in hash tables. It addresses collisions by incrementing the index using a quadratic function instead of a linear function. When a collision occurs, quadratic probing checks a series of indices using a quadratic formula, such as '$(\text{hash} + c1 * i + c2 * i^2)\%N$', where 'hash' is the original hash value, 'i' is the probe sequence, 'c1' and 'c2' are constants, and 'N' is the size of the hash table. This approach helps to distribute keys more evenly and mitigate clustering. Quadratic probing offers a balance between linear probing and other more complex collision resolution techniques, providing a compromise between simplicity and performance for handling collisions in hash tables.

Searching

Searching in the context of quadratic probing involves locating a key-value pair in a hash table that uses quadratic probing as its collision resolution technique. Similar to linear probing, the hash function is applied to the key to determine the initial index. If the key is not found at that index, quadratic probing is used to incrementally search through subsequent indices based on a quadratic formula, such as '$(\text{hash} + c1 * i + c2 * i^2)\%N$'. The search continues until the key is found or an empty slot is encountered. Quadratic probing helps to evenly distribute keys and mitigate clustering, enabling efficient search operations in hash tables. However, care must be taken to choose suitable constants 'c1' and 'c2' to avoid clustering patterns that can impact the performance of the search process.

Hash Table Quadratic Probe Search Algorithm

Below is an example of searching in a hash table via quadratic probing in the context of C++:

```
// Structure for the item in the hash table
struct Item {
    int key;
    std::string data;
};

// Function to search for a key in the hash table using quadratic probing
const Item* HashSearch(const std::vector<Item>& hashTable, int key) {
    size_t i = 0;
    size_t bucketsProbed = 0;
    size_t N = hashTable.size();

    // Hash function determines initial bucket
    size_t bucket = Hash(key) % N;

    while ((hashTable[bucket].status != EmptySinceStart) && (bucketsProbed < N)) {
        if ((hashTable[bucket].status == Occupied)
                && (hashTable[bucket].key == key)) {
            return &hashTable[bucket];
        }

        // Increment i and recompute bucket index using quadratic probing formula
        size_t c1 = 1; // Programmer-defined constant
        size_t c2 = 1; // Programmer-defined constant
        i = i + 1;
        bucket = (Hash(key) + c1 * i + c2 * i * i) % N;

```

```
            // Increment number of buckets probed
            bucketsProbed = bucketsProbed + 1;
        }

        return nullptr; // Key not found
    }

```

Removal

Removing in the context of quadratic probing involves deleting a key-value pair from a hash table that utilizes quadratic probing as its collision resolution technique. Similar to searching, the hash function is applied to the key to determine the initial index. If the key is not found at that index, quadratic probing is used to incrementally search through subsequent indices based on a quadratic formula, such as '$(\text{hash} + c1 * i + c2 * i^2)\%N$'. Once the key is located, the corresponding value is removed, and the slot is marked as empty. Quadratic probing ensures that the hash table remains compact and enables efficient deletion operations. However, it is crucial to use appropriate quadratic probing constants to avoid clustering patterns that can impact the performance of the removal process.

Hash Table Quadratic Probe Removal Algorithm

Below is an example of removal in a hash table via quadratic probing in the context of C++:

```
// Structure for the item in the hash table
struct Item {
    int key;
    std::string data;
    bool isEmptyAfterRemoval;
};

// Function to remove an item from the hash table using quadratic probing
bool HashRemove(std::vector<Item>& hashTable, int key) {
    size_t i = 0;
    size_t bucketsProbed = 0;
    size_t N = hashTable.size();

    // Hash function determines initial bucket
    size_t bucket = Hash(key) % N;

    while ((hashTable[bucket].status != EmptySinceStart) && (bucketsProbed < N)) {
        if ((hashTable[bucket].status == Occupied)
                && (hashTable[bucket].key == key)) {
            hashTable[bucket].isEmptyAfterRemoval = true;
            return true;
        }

        // Increment i and recompute bucket index using quadratic probing formula
        size_t c1 = 1; // Programmer-defined constant
        size_t c2 = 1; // Programmer-defined constant
        i = i + 1;
        bucket = (Hash(key) + c1 * i + c2 * i * i) % N;

        // Increment number of buckets probed
        bucketsProbed = bucketsProbed + 1;
    }

    return false; // Key not found
}

```

Inserting

Inserting in the context of quadratic probing involves adding a new key-value pair to a hash table that utilizes quadratic probing as its collision resolution technique. Similar to linear probing, the hash function is applied to the key to determine the initial index. If the index is already occupied, quadratic probing is used to sequentially search for the next available slot using a quadratic formula, such as '$(\text{hash} + c1 * i + c2 * i^2)\%N$', where 'hash' is the original hash value, 'i' is the probe sequence, 'c1' and 'c2' are constants, and 'N' is the size of the hash table. The key-value pair is inserted at the first unoccupied location found. Quadratic probing helps to evenly distribute keys and mitigate clustering, allowing efficient insertion operations in hash tables. However, it is essential to choose appropriate quadratic probing constants to avoid clustering patterns that can impact the performance of the insertion process.

Hash Table Quadratic Probe Insert Algorithm

Below is an example of inserting in a hash table via quadratic probing in the context of C++:

```
// Structure for the item in the hash table
```

```cpp
struct Item {
    int key;
    std::string data;
};

// Function to insert an item into the hash table using quadratic probing
bool HashInsert(std::vector<Item>& hashTable, const Item& item) {
    size_t i = 0;
    size_t bucketsProbed = 0;
    size_t N = hashTable.size();

    // Hash function determines initial bucket
    size_t bucket = Hash(item.key) % N;

    while (bucketsProbed < N) {
        // Insert item in the next empty bucket
        if (hashTable[bucket].status == Empty) {
            hashTable[bucket] = item;
            return true;
        }

        // Increment i and recompute bucket index using quadratic probing formula
        size_t c1 = 1; // Programmer-defined constant
        size_t c2 = 1; // Programmer-defined constant
        i = i + 1;
        bucket = (Hash(item.key) + c1 * i + c2 * i * i) % N;

        // Increment number of buckets probed
        bucketsProbed = bucketsProbed + 1;
    }

    return false; // Unable to insert item
}

```

Section 14.5 - Double Hashing

Overview

Double hashing is a collision resolution technique used in hash tables. It involves using two hash functions to determine the index for key-value pairs when collisions occur. When a collision occurs, the secondary hash function is applied to the key to calculate an offset value, which is then used to probe for the next available slot in the hash table. The double hashing technique helps distribute keys more evenly and reduces clustering compared to linear probing or quadratic probing. By employing two hash functions, it provides a greater range of possible indices and improves the chances of finding an empty slot. Double hashing offers an efficient way to resolve collisions and maintain good performance in hash tables.

Search

Searching with double hashing in hash tables involves utilizing two hash functions to handle collisions and locate a specific key-value pair. The primary hash function is applied to calculate the initial index, and if the key is not found at that index, the secondary hash function is used to determine the probe offset. The offset is added to the current index, and the search continues until the key is found or an empty slot is encountered. By employing two hash functions, double hashing reduces clustering and improves key distribution, leading to efficient and effective search operations in hash tables. The use of double hashing allows for a wider range of possible indices, increasing the chances of finding the desired key in a timely manner.

Removal

Removing with double hashing in hash tables involves using two hash functions to handle collisions and delete a specific key-value pair. The primary hash function is applied to calculate the initial index, and if the key is not found at that index, the secondary hash function is used to determine the probe offset. The offset is added to the current index, and the search continues until the key is found or an empty slot is encountered. Once the key is located, the corresponding value is removed, and the slot is marked as empty. By utilizing double hashing, the removal process is efficient and ensures the integrity of the hash table, effectively handling collisions and maintaining

good performance. The use of two hash functions allows for better key distribution and reduces clustering, providing an effective mechanism for removing key-value pairs in hash tables.

Inserting

Inserting with double hashing in hash tables involves using two hash functions to handle collisions and add a new key-value pair. The primary hash function is applied to calculate the initial index, and if the index is already occupied, the secondary hash function determines the probe offset. The offset is added to the current index, and the search continues until an empty slot is found. Once an empty slot is located, the new key-value pair is inserted into that slot, ensuring proper placement within the hash table. The use of double hashing allows for better key distribution and reduces clustering, providing an efficient and effective mechanism for inserting key-value pairs in hash tables. By employing two hash functions, the insertion process is optimized, leading to improved performance and maintaining the integrity of the hash table.

Section 14.6 - Hash Table Resizing

Overview

Hash table resizing is a dynamic operation that involves increasing or decreasing the size of a hash table to accommodate a changing number of elements. When the number of elements exceeds a certain threshold or falls below a certain load factor, resizing is triggered to ensure optimal performance. During resizing, a new hash table with a larger or smaller size is created, and all existing key-value pairs are rehashed and inserted into the new table. Resizing allows for better utilization of memory, improved efficiency, and reduced collision probabilities. It helps maintain a balanced and efficient hash table by adjusting its size based on the number of elements it needs to store, ensuring optimal performance and avoiding excessive collisions.

Resizing Requirements

Resizing a hash table is typically done when certain conditions are met, indicating the need for adjustment in order to maintain efficient performance. One common scenario is when the number of elements stored in the hash table exceeds a certain threshold, often referred to as the load factor. Resizing is triggered to increase the size of the hash table, allowing for a larger number of elements and reducing the likelihood of collisions. Conversely, if the number of elements decreases significantly, resizing may be performed to reduce the size of the hash table and optimize memory usage. The decision to resize a hash table depends on factors such as the desired load factor, the expected number of elements, and the performance requirements. Resizing a hash table ensures that it remains balanced and provides efficient access and storage of elements based on the current workload.

Hash Table Resizing Example

Below is an example of resizing a hash table in the context of C++:

```cpp
// Function to resize the hash table
std::vector<int> HashResize(const std::vector<int>& hashTable, int currentSize) {
    int newSize = nextPrime(currentSize * 2);

    std::vector<int> newArray(newSize, EmptySinceStart);

    int bucket = 0;
    while (bucket < currentSize) {
        if (hashTable[bucket] != Empty) {
            int key = hashTable[bucket];
            HashInsert(newArray, key); // Assuming HashInsert function exists
        }
        bucket = bucket + 1;
    }

    return newArray;
}

```

Section 14.7 - Common Hash Functions

Overview

Common hash functions are algorithms used to convert input data of arbitrary size into a fixed-size output value, typically used for indexing and accessing data in hash tables. Some commonly used hash functions include the division method, where the key is divided by the table size and the remainder is used as the index; the multiplication method, where the key is multiplied by a constant fraction and the fractional part is used as the index; and the mid-square method, where the key is squared and a portion of the middle bits is extracted as the index. Other popular hash functions include cryptographic hash functions like MD5 and SHA-1, which are designed for security applications. Each hash function has its advantages and considerations, such as efficiency, distribution of keys, and collision resistance, and the choice of hash function depends on the specific requirements of the application.

Modulo Hash Function

The modulo hash function, also known as the division method, is a simple and commonly used hash function. It involves taking the remainder of the key divided by the size of the hash table to determine the index for storage or retrieval. The modulo hash function is easy to implement and provides a uniform distribution of keys when the table size is a prime number. However, it may result in clustering and poor performance if the table size is not well-chosen, leading to frequent collisions. To mitigate this, the table size is often selected as a prime number to improve the distribution of keys and reduce collisions. The modulo hash function is widely used in many applications due to its simplicity and acceptable performance for various scenarios.

Modulo Hash Function Example

Below is an example of the modulo hash function in the context of C++:

```cpp
// Function to calculate the remainder using the modulo hash function
int HashRemainder(int key, int N) {
    return key % N;
}

```

Mid-Square Hash Function

The mid-square hash function is a technique used to convert a given key into a hash value by squaring the key and extracting a portion of the middle bits from the resulting square. The middle bits are then used as the hash index. This method aims to achieve a more uniform distribution of keys in the hash table. However, the effectiveness of the mid-square hash function heavily depends on the choice of the number of bits extracted from the square and the properties of the input keys. If the number of bits extracted is not carefully selected or if the input keys exhibit certain patterns, the mid-square hash function can result in clustering and an increased likelihood of collisions. Therefore, careful consideration should be given to the selection of the number of bits to extract and the nature of the keys being hashed.

Mid-Square Hash Function Example

Below is an example of the mid-square has function in the context of C++:

```cpp
// Function to calculate the hash using the mid-square hash function
int HashMidSquare(int key, int R, int N) {
    int squaredKey = key * key;

    int lowBitsToRemove = (32 - R) / 2;
    int extractedBits = squaredKey >> lowBitsToRemove;
    extractedBits = extractedBits & (0xFFFFFFFF >> (32 - R));

    return extractedBits % N;
}

```

Multiplicative String Hash Function

The multiplicative string hash function is a technique used to convert a string into a hash value by multiplying the numeric representation of each character in the string by a constant factor. The constant factor is typically a fractional value between 0 and 1, chosen carefully to ensure good distribution and reduce the likelihood of collisions.

The multiplication is performed on each character in the string, and the resulting values are combined to generate the final hash value. The multiplicative string hash function aims to achieve a balanced distribution of keys and can be efficient for string-based hashing. However, it requires careful selection of the constant factor to avoid clustering and ensure a uniform distribution of hash values.

Multiplicative String Hash Function Example

Below is an example of the multiplicative string hash function in the context of C++:

```cpp
// Function to calculate the hash using the multiplicative string hash function
int HashMultiplicative(const std::string& key, int N,
                       int InitialValue, int HashMultiplier) {
    int stringHash = InitialValue;

    for (char strChar : key) {
        stringHash = (stringHash * HashMultiplier) + static_cast<int>(strChar);
    }

    return stringHash % N;
}

```

Section 14.8 - Direct Hashing

Direct hashing, also known as perfect hashing, is a hash function technique that eliminates collisions entirely by ensuring a one-to-one mapping between keys and their corresponding hash values. It achieves this by precomputing a hash function specific to the set of keys being hashed. In direct hashing, an initial hash function is used to determine a primary hash value, and then a secondary hash function is applied to resolve any potential collisions. The secondary hash function maps each key to a unique index in a small auxiliary table, which stores the key-value pairs. Direct hashing provides constant-time access to the stored values, as there are no collisions to handle. However, it requires extra memory to store the auxiliary table and entails additional computation during the construction phase to create the perfect hash function. Direct hashing is suitable for applications where collision-free access to a fixed set of keys is essential.

Section 14.9 - Cryptography

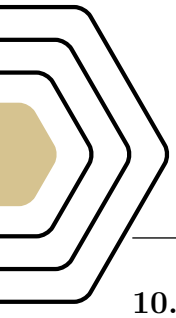
Cryptography

Cryptographic hashing is a method used to transform input data into a fixed-size, unique, and irreversible hash value. It is commonly employed in various security applications, such as password storage, data integrity verification, and digital signatures. Cryptographic hash functions have specific properties, including pre-image resistance, second pre-image resistance, and collision resistance, which ensure that it is computationally infeasible to reconstruct the original data or find two different inputs that produce the same hash value. These hash functions generate a unique output for each input, making them suitable for verifying data integrity, detecting tampering, and securely storing passwords without revealing the actual passwords themselves. Examples of widely used cryptographic hash functions include MD5, SHA-1, SHA-256, and SHA-3.

Hashing Functions For Data

Hashing functions for data are algorithms that convert data of any size or type into a fixed-size hash value. These functions are designed to efficiently map data to a unique representation, enabling quick data retrieval, identification, and comparison. Hashing functions generate a hash value based on the content of the data, ensuring that even a small change in the input data produces a significantly different output. This property makes hashing functions ideal for tasks such as data indexing, data integrity verification, duplicate detection, and password storage. Well-designed hashing functions exhibit desirable properties, including uniform distribution, minimal collisions, and resistance to pre-image attacks, ensuring the integrity and security of data in various applications and systems.

Compression



Compression

10.0.1 Activities

The following are the activities that are planned for Week 10 of this course.

- Complete your Assessments of peer Project Proposals
- Read zyBooks Chapter 15 and do activities (due next Monday)
- Watch lecture videos
- Homework on Huffman (due next Tuesday)

10.0.2 Lectures

Here are the lectures that can be found for this week:

- Encoding & Decoding (Codecs)
- Lossless Text Compression
- Huffman Trees
- Huffman Encode & Decode

The lecture notes for this week are:

- Codec Lecture Notes

10.0.3 Programming Assignment

The programming assignment for Week 10 is:

- Programming Assignment 8 - Huffman

10.0.4 Chapter Summary

The chapter of this week is **Chapter 15: Compression**.

Section 15.1 - Compression

Compression

Compression in computer science refers to the process of reducing the size of data files or streams without significant loss of information. It aims to optimize storage space and transmission bandwidth by eliminating redundant or irrelevant data. Compression algorithms leverage various techniques such as removing data duplication, representing data in a more efficient manner, or encoding patterns to achieve higher compression ratios. By effectively compressing data, computer systems can store and transmit information more efficiently, resulting in reduced storage requirements, faster transmission speeds, and improved overall performance.

Dictionary Compression

Dictionary-based compression is a data compression technique that involves building and utilizing a dictionary to achieve efficient compression. The dictionary is essentially a collection of frequently occurring patterns or sequences in the input data. During compression, the algorithm replaces these patterns with shorter codes or references, thus reducing the overall size of the data. This approach is particularly effective when there are repetitive patterns or long sequences of repeated data in the input. Dictionary-based compression algorithms, such as LZ77 and LZ78, have been widely used in various applications, including file compression formats like ZIP and gzip, as well as network protocols to optimize data transmission.

Lossless Compression

Lossless compression refers to a data compression technique that allows for the complete reconstruction of the original data from the compressed version without any loss of information. Unlike lossy compression, which sacrifices some data quality for higher compression ratios, lossless compression algorithms aim to reduce the size of data files or streams while preserving every single bit of the original content. These algorithms leverage various techniques, such as dictionary-based methods, entropy encoding, and run-length encoding, to eliminate redundancy and compress the data efficiently. Lossless compression is crucial in applications where data integrity is paramount, such as archiving, data storage, and transmission of critical information, as it ensures that the original data can be fully recovered without any loss or corruption.

Lossy Compression

Lossy compression is a data compression technique that sacrifices some amount of data quality in order to achieve higher compression ratios. Unlike lossless compression, lossy compression permanently discards certain information deemed less perceptually significant or redundant. This approach is commonly used in multimedia applications, such as image, audio, and video compression, where small losses in quality may be acceptable to achieve significant reductions in file sizes. Lossy compression algorithms employ various methods, such as quantization, transformation, and perceptual coding, to discard or approximate less critical data while preserving perceptual fidelity. While lossy compression may result in a slight degradation of the reconstructed data compared to the original, it offers considerable benefits in terms of reduced storage requirements and efficient transmission of multimedia content.

JPEG Compression

JPEG (Joint Photographic Experts Group) compression is a widely used lossy compression technique specifically designed for compressing digital images. It is based on the principles of human visual perception and aims to minimize the file size of images while maintaining an acceptable level of visual quality. JPEG compression achieves this by applying a combination of techniques, including color subsampling, discrete cosine transform (DCT), quantization, and Huffman coding. The color subsampling reduces the resolution of the color information, while the DCT converts image data into a frequency domain representation. Quantization discards less visually important details, and Huffman coding efficiently encodes the transformed data. The resulting compressed JPEG image can significantly reduce file sizes while preserving a visually satisfactory representation, making it suitable for efficient storage and transmission of digital images in various applications.

Video Compression

Video compression is a process of reducing the size of digital video files while maintaining an acceptable level of visual quality. It involves applying various compression techniques to exploit redundancies and eliminate unnecessary information in the video data. Video compression algorithms typically utilize both temporal and spatial redundancies present in consecutive frames and within each frame, respectively. These algorithms may include methods such as motion estimation, motion compensation, transform coding (e.g., discrete cosine transform), quantization, and entropy coding. By efficiently compressing video data, video compression enables efficient storage, transmission, and streaming of videos over limited bandwidth networks while minimizing storage requirements without significant perceptible loss in quality. Popular video compression standards include MPEG (Moving Picture Experts Group) formats like MPEG-2, MPEG-4, and H.264/AVC.

Section 15.2 - Data Compression

Overview

Data compression is the process of reducing the size of data files or streams to optimize storage space and transmission bandwidth. It involves various techniques that eliminate redundancy, exploit patterns, and encode data in a more efficient manner. There are two main types of data compression: lossless and lossy compression. Lossless compression aims to reduce file size without any loss of information, while lossy compression sacrifices some data quality to achieve higher compression ratios. Compression algorithms can be applied to various types of data, including text, images, audio, and video, offering benefits such as reduced storage requirements, faster transmission speeds, and improved overall efficiency of data handling in computer systems and communication networks.

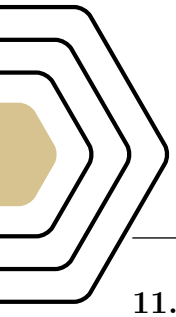
Humman Coding

Huffman coding is a widely used algorithm for lossless data compression that assigns variable-length codes to symbols based on their frequency of occurrence in the input data. It exploits the principle of assigning shorter codes to more frequently occurring symbols, resulting in efficient compression. The algorithm constructs a binary tree, known as the Huffman tree, by iteratively combining the least frequent symbols until all symbols are incorporated into the tree. The resulting codes are then derived from the paths traversed in the tree, with shorter codes assigned to symbols closer to the root. Huffman coding is widely utilized in various compression formats, such as ZIP, JPEG, and MP3, to achieve effective compression while preserving the original data without any loss.

Decompressing Huffman Coded Data

Decompressing Huffman-coded data involves the reverse process of the Huffman coding algorithm. It takes the compressed data, along with the Huffman tree or codebook used for encoding, and reconstructs the original uncompressed data. Starting from the compressed bitstream, the algorithm traverses the Huffman tree, bit by bit, decoding each code into its corresponding symbol. By following the path in the tree based on the encountered bits, the algorithm recreates the original sequence of symbols. This process continues until the entire compressed data has been decoded and the uncompressed data is fully reconstructed. Decompressing Huffman-coded data allows for the recovery of the original information, enabling the restoration of the data to its original form after compression.

Graphs



Graphs

11.0.1 Activities

The following are the activities that are planned for Week 11 of this course.

- Exam #2 (Friday Aug 4th, 10 am - 10 pm)
- Read zyBooks Chapter 16 and work on book activities
- Watch lecture videos
- Start on Graphs Homework

11.0.2 Lectures

Here are the lectures that can be found for this week:

- Intro To Graphs
- Implementing Graphs
- Graph Metadata
- Star Wars Graph
- Depth First Search
- Breadth First Search
- Directed Acyclic Graphs
- Spanning Trees & Edge Types

The lecture notes for this week are:

- Graphs Lecture Notes

11.0.3 Programming Assignment

The programming assignment for Week 11 is:

- Programming Assignment 9 - Graphs

11.0.4 Exam

The exam for this week is:

- Exam 2 Notes
- Exam 2

11.0.5 Chapter Summary

The chapter of this week is **Chapter 16: Graphs**.

Section 16.1 - Graphs: Introduction

Overview

In computer science, graphs are fundamental data structures used to model and analyze various relationships and connections between objects. A graph consists of a set of vertices (or nodes) and a collection of edges that represent the connections between those vertices. Graphs are extensively employed in various domains, such as network analysis, social networks, routing algorithms, optimization problems, and data visualization. They provide a powerful framework for solving complex problems, enabling efficient representation and manipulation of interconnected data, and facilitating the exploration and analysis of relationships within a system or dataset. Graph algorithms and techniques, ranging from basic traversal and path finding to more advanced algorithms like spanning trees and shortest paths, play a crucial role in solving a wide range of computational problems in computer science.

Adjacency & Paths

In graph theory, adjacency and paths are key concepts used to analyze the relationships and connectivity within a graph. Adjacency refers to the relationship between two vertices in a graph that are directly connected by an edge. It represents the presence or absence of a connection between vertices and forms the basis for many graph algorithms and operations. Paths, on the other hand, are sequences of edges that connect a series of vertices in a graph. They represent routes or connections between vertices and can be used to determine the existence of a path between two vertices, find the shortest path, or explore various paths within a graph. The study of adjacency and paths in graphs is crucial for understanding the connectivity and structure of data, and it enables the development of algorithms that solve complex problems in fields such as network routing, social network analysis, and optimization.

Section 16.2 - Applications of Graphs

Graphs find extensive applications in various fields due to their ability to model and analyze relationships and connections. In computer science, graphs are utilized for network analysis, social network analysis, routing algorithms, and data visualization. They are also widely used in logistics and transportation for route planning and optimization. In biology, graphs help represent gene interactions, protein networks, and evolutionary relationships. Graphs are applied in recommendation systems to analyze user preferences and make personalized suggestions. Furthermore, graphs are valuable in finance for risk analysis, fraud detection, and portfolio optimization. Overall, the applications of graphs span diverse domains, providing a versatile framework for solving complex problems and gaining insights from interconnected data structures.

Section 16.3 - Graph Representations: Adjacency Lists

Overview

Graphs can be represented in various ways, and one common representation is through adjacency lists. In this representation, each vertex in the graph is associated with a list that contains its adjacent vertices. The adjacency list efficiently captures the connectivity of the graph by storing only the necessary information. It allows for quick access to a vertex's neighbors and enables efficient traversal of the graph. Moreover, it requires less space compared to other representations, especially for sparse graphs. Adjacency lists are widely used in graph algorithms and applications where efficient neighbor access and memory optimization are essential, making them a popular choice for representing graphs in computer science.

Advantages of Adjacency Lists

Adjacency lists offer several advantages as a graph representation. Firstly, they are memory-efficient, especially for sparse graphs, as they only store the necessary information about the connections between vertices. This allows for significant space savings compared to other representations, such as adjacency matrices. Secondly, adjacency lists facilitate efficient neighbor access. Given a vertex, its adjacent vertices can be quickly retrieved by accessing the associated list. This enables efficient graph traversal and exploration of neighboring relationships. Additionally, adjacency lists can be easily modified, as adding or removing edges requires updating only the relevant lists. Overall, adjacency lists strike a balance between memory efficiency and efficient neighbor access, making them a popular choice in scenarios where memory optimization and efficient graph operations are crucial.

Section 16.4 - Graph Representations: Adjacency Matrices

Overview

Adjacency matrices are a commonly used representation for graphs. In this approach, a matrix is utilized to indicate the connections between vertices. Each row and column of the matrix correspond to a vertex, and the entry at the intersection of a row and column represents whether an edge exists between the respective vertices. Adjacency matrices provide a straightforward and intuitive representation, allowing for easy identification of edge presence and absence. They also enable efficient lookup of edge information. However, they can be memory-intensive, especially for large and sparse graphs, as the matrix size is proportional to the square of the number of vertices. Despite this drawback, adjacency matrices are preferred in situations where edge queries are frequent and memory constraints are not a significant concern.

Efficiency

Adjacency matrices offer several efficiency advantages as a graph representation. First and foremost, they provide constant-time lookup for determining edge existence between two vertices. This makes adjacency matrices highly efficient for querying edge information. Additionally, adjacency matrices allow for efficient computation of various graph properties, such as the degree of a vertex, the number of edges in the graph, and checking for self-loops. They also facilitate efficient matrix operations, enabling algebraic operations on graphs, such as matrix exponentiation for calculating paths and reachability. However, adjacency matrices can be memory-intensive, especially for large graphs or sparse graphs, where most entries in the matrix are zero. In such cases, the memory overhead can be significant, leading to inefficiencies. Therefore, the choice of using adjacency matrices depends on the specific requirements of the application, considering the trade-off between efficiency and memory usage.

Section 16.5 - Graphs: Breadth-First Search

Overview

Breadth-first search (BFS) is a fundamental graph traversal algorithm that systematically explores a graph by visiting all the vertices at the same level before moving on to the next level. Starting from a given source vertex, BFS explores all the vertices that are reachable in one hop before moving on to the vertices reachable in two hops, and so on. It utilizes a queue data structure to keep track of the vertices to be visited, ensuring that vertices closer to the source are visited before those farther away. BFS is widely used in various applications such as finding the shortest path, determining connectivity, and discovering all the vertices within a certain distance from a source vertex. Its simplicity, ability to find the shortest path in unweighted graphs, and guarantee of visiting all reachable vertices make it a valuable algorithm in graph theory and computer science.

Breadth-First Search Algorithm

The Breadth-First Search (BFS) algorithm is a fundamental graph traversal technique used to systematically explore a graph. Starting from a given source vertex, BFS explores all the vertices at the same level before moving on to the next level. It uses a queue data structure to manage the order in which vertices are visited. By maintaining a visited set, BFS ensures that each vertex is visited only once, preventing infinite loops in graphs with cycles. BFS is particularly useful for finding the shortest path between two vertices in an unweighted graph and for exploring all reachable vertices from a given source. Its time complexity is linear in the number of vertices and edges in the graph, making it an efficient algorithm for graph exploration and path finding.

BFS Algorithm Example

Below is an example of the breadth first search algorithm in the context of C++:

```cpp
// Breadth-First Search implementation
void BFS(int startVertex) {
    vector<bool> visited(numVertices, false); // Track visited vertices
    queue<int> bfsQueue; // Queue for BFS traversal

    visited[startVertex] = true;
    bfsQueue.push(startVertex);

    while (!bfsQueue.empty()) {
        int currVertex = bfsQueue.front();
        bfsQueue.pop();

        cout << currVertex << " ";

        for (int neighbor : adjList[currVertex]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                bfsQueue.push(neighbor);
            }
        }
    }
}

```

The Breadth-First Search (BFS) algorithm is implemented in the provided C++ code. It starts from a given vertex and explores its neighboring vertices at the same level before moving on to the next level. The algorithm uses a queue to manage the order of vertex visits and a visited array to keep track of the visited vertices. During the BFS traversal, each visited vertex is printed, ensuring that all reachable vertices are visited exactly once. The example code demonstrates the usage of the BFS algorithm on a graph with 6 vertices, showcasing the order in which the vertices are visited.

Section 16.6 - Graphs: Depth-First Search

Overview

Depth-First Search (DFS) is a popular graph traversal algorithm that systematically explores a graph by traversing as far as possible along each branch before backtracking. Starting from a given source vertex, DFS visits an adjacent unvisited vertex and continues recursively until it reaches a vertex with no unvisited neighbors. It then backtracks to the previous vertex and explores any remaining unvisited neighbors. This process continues until all vertices have been visited. DFS is commonly used for applications like finding connected components, detecting cycles, and exploring paths in graphs. Its simplicity and ability to deeply explore a graph make it a valuable algorithm in graph theory and computer science.

Depth-First Search Algorithm

The Depth-First Search (DFS) algorithm is a fundamental graph traversal technique that systematically explores a graph by traversing as deeply as possible before backtracking. Starting from a given source vertex, DFS visits an adjacent unvisited vertex and continues recursively until it reaches a vertex with no unvisited neighbors. It then backtracks to the previous vertex and explores any remaining unvisited neighbors. This process continues until all vertices have been visited. DFS is widely used for applications such as finding connected components, detecting

cycles, and exploring paths in graphs. Its recursive nature and ability to deeply explore a graph make it a powerful algorithm in graph theory and computer science.

Depth-First Search Algorithm Example

Below is an example of the depth first search algorithm in the context of C++:

```cpp
// Recursive Depth-First Search implementation
void RecursiveDFS(int currentV, unordered_set<int>& visitedSet) {
    if (visitedSet.find(currentV) == visitedSet.end()) {
        visitedSet.insert(currentV);
        cout << "Visit " << currentV << endl;

        for (int adjV : adjList[currentV]) {
            RecursiveDFS(adjV, visitedSet);
        }
    }
}

```

The provided C++ code implements the Depth-First Search (DFS) algorithm for graph traversal. The 'RecursiveDFS' function recursively explores the graph starting from a given vertex. It maintains a set of visited vertices to avoid revisiting already visited vertices. During the traversal, each vertex is "visited" by printing its value. The algorithm recursively explores all adjacent unvisited vertices until no further unvisited neighbors are found, at which point it backtracks to the previous vertex. The code demonstrates the usage of the DFS algorithm on a graph with 6 vertices, showcasing the order in which the vertices are visited.

Section 16.7 - Directed Graphs

Overview

Directed graphs, also known as digraphs, are a type of graph in which the edges have a specific direction. In a directed graph, each edge is associated with an ordered pair of vertices, indicating a one-way connection from one vertex (the source) to another vertex (the target). This directional nature of edges allows directed graphs to model various real-world scenarios such as transportation networks, social media relationships, information flow, and dependencies between tasks. Directed graphs provide a powerful framework for analyzing asymmetric relationships and capturing the flow of information or influence in a system. They are widely used in fields like computer science, network analysis, social network analysis, and operations research for solving problems that require modeling and analyzing directional relationships and dependencies.

Paths & Cycles

In directed graphs, paths and cycles play significant roles in understanding and analyzing the structure and dynamics of the graph. A path in a directed graph is a sequence of vertices connected by directed edges, where each vertex is the target of the previous edge and the source of the subsequent edge. Paths can represent various relationships or sequences of actions in a directed graph. A cycle, on the other hand, is a closed path that starts and ends at the same vertex, allowing for repeated visits to vertices. Cycles can signify recurring patterns or feedback loops within a directed graph. Analyzing paths and cycles in directed graphs helps in understanding connectivity, reachability, and the flow of information or processes within the graph. It enables the identification of optimal routes, detection of dependencies or bottlenecks, and analysis of system behavior in various domains such as network routing, task scheduling, and circuit design.

Section 16.8 - Weighted Graphs

Overview

Weighted graphs are a type of graph where each edge is assigned a numerical value or weight. These weights represent the cost, distance, or any other quantitative measure associated with traversing the edge. Weighted graphs are used to model real-world scenarios where the relationships between vertices have different magnitudes or significance. They are commonly employed in applications such as transportation networks, route planning, resource allocation, and optimization problems. Weighted graphs enable the analysis of the shortest paths between vertices based on the accumulated weights, allowing for efficient route optimization and decision-making. Algorithms like Dijkstra's algorithm and the Bellman-Ford algorithm are specifically designed to handle weighted graphs and determine the optimal paths based on the assigned weights. The introduction of weights in graphs adds a new dimension to their representation, enabling a more accurate and detailed modeling of various systems and phenomena.

Path Length

In weighted graphs, path length refers to the sum of the weights of the edges along a given path. It represents the total cost, distance, or any other quantitative measure required to traverse the path from the source vertex to the destination vertex. Path length is a crucial metric in analyzing weighted graphs as it allows for evaluating the efficiency, optimality, and comparison of different paths. Determining the shortest path, which has the minimum path length, is a common problem in weighted graph analysis. Various algorithms, such as Dijkstra's algorithm and the A* algorithm, are designed to calculate the shortest path by considering the weights of the edges. Path length in weighted graphs is essential in applications such as route planning, network optimization, resource allocation, and scheduling, as it helps in making informed decisions and finding the most efficient paths based on the assigned weights.

Negative Edge Weigth Cycle

Negative edge weight cycles are cycles in weighted graphs where the sum of the weights along the cycle is negative. These cycles introduce unique challenges and considerations in graph analysis. Unlike positive cycles that can simply be ignored for finding shortest paths, negative cycles can lead to ambiguous or infinite path lengths. They can disrupt traditional algorithms like Dijkstra's algorithm or the Bellman-Ford algorithm, which assume non-negative weights, and cause them to produce incorrect or undefined results. Negative edge weight cycles can arise in scenarios such as financial transactions with debts, time-travel scenarios, or scenarios involving resource allocation. Detecting and handling negative cycles in graph analysis often involves specialized algorithms like the Bellman-Ford algorithm with cycle detection or the Floyd-Warshall algorithm. Understanding and managing negative edge weight cycles is crucial for accurate analysis and decision-making in situations where negative weights are present in the graph.

Section 16.9 - Algorithm: Dijkstra's Shortest Path

Dijkstra's algorithm is a popular and efficient algorithm for finding the shortest path between a source vertex and all other vertices in a weighted graph. It operates by maintaining a set of visited vertices and a set of tentative distances from the source vertex to each vertex in the graph. Initially, all tentative distances are set to infinity except for the source vertex, which is set to 0. The algorithm iteratively selects the vertex with the minimum tentative distance, known as the "greedy" step, and updates the distances of its neighboring vertices if a shorter path is found. By repeating this process until all vertices have been visited, Dijkstra's algorithm effectively computes the shortest path from the source vertex to every other vertex. The algorithm's time complexity depends on the implementation, typically ranging from $\mathcal{O}(V^2)$ to $\mathcal{O}((V+E)\log(V))$, where V represents the number of vertices and E denotes the number of edges in the graph. Dijkstra's algorithm is widely used in various applications such as route planning, network routing, and resource allocation.

Dijkstra's Shortest Path Algorithm Example

Below is an example of Dijkstra's shortest path algorithm in the context of C++:

```
1  // Dijkstra's Shortest Path Algorithm
2  void DijkstraShortestPath(int startV) {
3      // Create an array of vertices
4      vector<Vertex> vertices(numVertices);
5
6      // Initialize vertices
7      for (int i = 0; i < numVertices; i++) {
8          vertices[i].id = i;
```

```
            vertices[i].distance = numeric_limits<int>::max();
            vertices[i].predV = -1;
        }

        // Start vertex has a distance of 0 from itself
        vertices[startV].distance = 0;

        // Priority queue to store unvisited vertices
        priority_queue<pair<int, int>, vector<pair<int, int>>,
            greater<pair<int, int>>> unvisitedQueue;

        // Enqueue startV
        unvisitedQueue.push(make_pair(vertices[startV].distance, startV));

        while (!unvisitedQueue.empty()) {
            // Dequeue vertex with minimum distance from startV
            int currentV = unvisitedQueue.top().second;
            unvisitedQueue.pop();

            for (auto neighbor : adjList[currentV]) {
                int adjV = neighbor.first;
                int edgeWeight = neighbor.second;
                int alternativePathDistance = vertices[currentV].distance + edgeWeight;

                // If shorter path from startV to adjV is found,
                // update adjV's distance and predecessor
                if (alternativePathDistance < vertices[adjV].distance) {
                    vertices[adjV].distance = alternativePathDistance;
                    vertices[adjV].predV = currentV;

                    // Enqueue updated vertex
                    unvisitedQueue.push(make_pair(vertices[adjV].distance, adjV));
                }
            }
        }

        // Print shortest path distances and predecessors
        cout << "Vertex\tDistance\tPredecessor" << endl;
        for (const auto& vertex : vertices) {
            cout << vertex.id << "\t" << vertex.distance << "\t\t"
                << vertex.predV << endl;
        }
    }

```

Dijkstra's Shortest Path Algorithm is implemented in the provided C++ code. It finds the shortest path between a given source vertex and all other vertices in a weighted graph. The algorithm initializes the distances of all vertices to infinity except for the source vertex, which is set to 0. It uses a priority queue to greedily select the vertex with the smallest tentative distance and updates the distances and predecessors of its neighboring vertices if a shorter path is found. This process continues until all vertices have been visited. Finally, the algorithm outputs the shortest path distances and predecessors for each vertex. The code demonstrates the application of Dijkstra's algorithm on a graph with 6 vertices, providing the shortest path information starting from a specified source vertex.

Section 16.10 - Algorithm: Bellman-Ford's Shortest Path

Bellman-Ford's Shortest Path Algorithm is a dynamic programming-based algorithm used to find the shortest path between a source vertex and all other vertices in a weighted graph. The algorithm works by iteratively relaxing the edges of the graph, reducing the estimated distance from the source vertex to each vertex. It initializes the distance of the source vertex to 0 and sets the distance of all other vertices to infinity. By repeatedly relaxing each edge in the graph, the algorithm guarantees to find the shortest paths in the presence of negative edge weights and detects negative cycles. After a maximum of V-1 iterations (V being the number of vertices), Bellman-Ford's algorithm produces the shortest path distances and predecessors for each vertex. This algorithm is widely used in various applications where negative edge weights and negative cycles need to be handled, such as network routing and resource allocation problems.

Bellman-Ford's Shortest Path Algorithm Example

Below is an example of Bellman-Ford's shortest path algorithm in the context of C++:

```
// Bellman-Ford's Shortest Path Algorithm
void BellmanFord(int startV) {
    // Initialize vertices
```

```
    for (int i = 0; i < numVertices; i++) {
        vertices[i].id = i;
        vertices[i].distance = numeric_limits<int>::max();
        vertices[i].predV = -1;
    }

    // Start vertex has a distance of 0 from itself
    vertices[startV].distance = 0;

    // Relax edges iteratively (V - 1) times
    for (int i = 1; i < numVertices; i++) {
        for (const auto& edge : edges) {
            int src = edge.src;
            int dest = edge.dest;
            int weight = edge.weight;
            int altPathDistance = vertices[src].distance + weight;

            // If shorter path from startV to dest is found,
            // update dest's distance and predecessor
            if (altPathDistance < vertices[dest].distance) {
                vertices[dest].distance = altPathDistance;
                vertices[dest].predV = src;
            }
        }
    }

    // Check for negative weight cycle
    for (const auto& edge : edges) {
        int src = edge.src;
        int dest = edge.dest;
        int weight = edge.weight;
        int altPathDistance = vertices[src].distance + weight;

        if (altPathDistance < vertices[dest].distance) {
            cout << "Negative weight cycle detected!" << endl;
            return;
        }
    }

    // Print shortest path distances and predecessors
    cout << "Vertex\tDistance\tPredecessor" << endl;
    for (const auto& vertex : vertices) {
        cout << vertex.id << "\t" << vertex.distance << "\t\t" << vertex.predV << endl;
    }
}

```

The provided C++ code implements the Bellman-Ford's Shortest Path Algorithm. This algorithm finds the shortest path between a source vertex and all other vertices in a weighted graph. It initializes the distances of all vertices to infinity except for the source vertex, which is set to 0. Then, it iteratively relaxes the edges V-1 times, updating the distances and predecessors whenever a shorter path is discovered. The algorithm also checks for the presence of negative weight cycles. Finally, it outputs the shortest path distances and predecessors for each vertex. The code demonstrates the application of the Bellman-Ford algorithm on a graph with 5 vertices and 7 edges, providing the shortest path information starting from a specified source vertex.

Section 16.11 - Topological Sort

Overview

Topological sort is an algorithm used to linearly order the vertices of a directed graph such that for every directed edge (u, v), vertex u comes before vertex v in the ordering. The topological sort is only applicable to directed acyclic graphs (DAGs) since cyclic graphs do not have a valid topological ordering. The algorithm starts by selecting a vertex with no incoming edges, called a source vertex, and adds it to the result. It then removes the source vertex and its outgoing edges from the graph. This process is repeated until all vertices have been processed. Topological sort is commonly used in tasks that require a specific order of operations or dependencies, such as scheduling tasks, resolving dependencies, and compiling software. It provides a consistent and deterministic way to organize the vertices of a directed graph based on their dependencies.

Topological Sort Algorithm

The Topological Sort algorithm is used to linearly order the vertices of a directed acyclic graph (DAG) based on their dependencies. It starts by selecting a vertex with no incoming edges, known as a source vertex, and adds it to the result. The algorithm then removes the source vertex and its outgoing edges from the graph. This process is repeated until all vertices have been processed. Topological sort provides a consistent and deterministic ordering of the vertices, ensuring that no vertex comes before its dependencies. It is commonly used in various applications, such as task scheduling, dependency resolution, and building project plans, where a specific order of operations or dependencies needs to be established.

Topological Sort Algorithm Example

Below is an example of the topological sort algorithm in the context of C++:

```cpp
// Function to perform topological sort
vector<int> topologicalSort() {
    vector<int> resultList; // List of vertices in topological order
    vector<int> incomingCount(numVertices, 0); // Incoming edge count for each vertex
    list<int> noIncoming; // List of vertices with no incoming edges

    // Calculate incoming edge count for each vertex
    for (int i = 0; i < numVertices; i++) {
        for (int j : adjList[i]) {
            incomingCount[j]++;
        }
    }

    // Add vertices with no incoming edges to the noIncoming list
    for (int i = 0; i < numVertices; i++) {
        if (incomingCount[i] == 0) {
            noIncoming.push_back(i);
        }
    }

    // Perform topological sort
    while (!noIncoming.empty()) {
        int currentV = noIncoming.front();
        noIncoming.pop_front();
        resultList.push_back(currentV);

        for (int j : adjList[currentV]) {
            incomingCount[j]--;

            if (incomingCount[j] == 0) {
                noIncoming.push_back(j);
            }
        }
    }

    return resultList;
}

```

The provided C++ code implements the topological sort algorithm. Topological sort is used to linearly order the vertices of a directed acyclic graph (DAG) based on their dependencies. The algorithm starts by identifying the vertices with no incoming edges, adds them to the result list, and removes their outgoing edges. This process is repeated until all vertices have been processed. Topological sort provides an ordering of the vertices that satisfies the dependencies within the graph. The code demonstrates the application of the topological sort algorithm on a graph, displaying the order in which the vertices should be processed to maintain the graph's dependency structure.

Section 16.12 - Minimum Spanning Tree

Overview

A minimum spanning tree (MST) is a subset of edges in a connected, weighted graph that connects all vertices while minimizing the total weight or cost of the tree. The goal of finding an MST is to identify the set of edges that form a tree with the smallest possible sum of weights. MSTs are useful in various applications such as network design, transportation planning, and communication networks. Popular algorithms for finding the MST include

Prim's algorithm and Kruskal's algorithm. By constructing an MST, we can ensure efficient and cost-effective connections between vertices, optimizing the use of resources and minimizing overall costs in a given graph.

Kruskal MST Algorithm

Kruskal's algorithm is a widely used algorithm for finding a minimum spanning tree (MST) in a connected, weighted graph. The algorithm works by initially treating each vertex as a separate component and sorting the edges of the graph in non-decreasing order of their weights. It then iterates through the sorted edges, adding them to the MST if they do not create a cycle. This process continues until all vertices are connected, resulting in the construction of a minimum spanning tree. Kruskal's algorithm is efficient and guarantees the creation of an MST, making it a popular choice for various applications such as network design, cable layout planning, and clustering.

Kruskal MST Algorithm Example

Below is an example of Kruskal's minimum spanning tree algorithm in the context of C++:

```cpp
// Kruskal's Minimum Spanning Tree Algorithm
vector<Edge> kruskalsMinimumSpanningTree() {
    sort(edgeList.begin(), edgeList.end(), [](const Edge& a, const Edge& b) {
        return a.weight < b.weight;
    });

    vector<int> parent(numVertices, -1); // Array to track parent of each vertex
    vector<Edge> resultList; // List of edges in the minimum spanning tree

    for (int i = 0; i < numVertices; i++) {
        parent[i] = i;
    }

    int edgesAdded = 0;
    int index = 0;

    while (edgesAdded < numVertices - 1 && index < edgeList.size()) {
        Edge nextEdge = edgeList[index];
        int set1 = findParent(parent, nextEdge.vertex1);
        int set2 = findParent(parent, nextEdge.vertex2);

        if (set1 != set2) {
            resultList.push_back(nextEdge);
            unionSets(parent, set1, set2);
            edgesAdded++;
        }

        index++;
    }

    return resultList;
}

```

The provided C++ code implements Kruskal's algorithm for finding a minimum spanning tree (MST) in a weighted graph. The algorithm starts by sorting the edges of the graph in non-decreasing order of their weights. It then iterates through the sorted edges and adds them to the MST if they do not create a cycle. This is achieved by maintaining a collection of vertex sets and checking if the vertices of the current edge belong to different sets. The algorithm performs union operations on the sets and keeps track of the added edges. It terminates when either all vertices are connected or there are no more edges to consider. Finally, it returns the set of edges forming the MST. The code demonstrates the application of Kruskal's algorithm on a graph, displaying the edges of the minimum spanning tree along with their weights.

Section 16.13 - All Pairs Shortest Path

Overview

All pairs shortest path is an algorithmic problem that aims to find the shortest path between all pairs of vertices in a directed or undirected weighted graph. The problem is to determine the minimum path length between any two vertices in the graph, considering all possible pairs. Various algorithms can solve the all pairs shortest path problem, such as Floyd-Warshall algorithm and Johnson's algorithm. These algorithms use dynamic programming or graph traversal techniques to calculate the shortest path distances between all pairs of vertices efficiently. All pairs shortest

path is applicable in diverse domains like transportation networks, routing protocols, and optimization problems, enabling efficient route planning and network analysis by identifying the shortest paths between any given pair of vertices in a graph.

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is a dynamic programming algorithm used to find the shortest path between all pairs of vertices in a directed or undirected weighted graph. It constructs a matrix of shortest path distances by iteratively considering intermediate vertices as potential waypoints in the paths. The algorithm compares the direct distance between vertices with the distances via intermediate vertices and updates the matrix accordingly. By performing a series of updates for each vertex as an intermediate point, Floyd-Warshall algorithm efficiently computes the shortest path distances between all pairs of vertices. It is particularly useful for dense graphs or graphs with negative edge weights. The resulting matrix provides valuable information about the shortest paths, allowing for efficient route planning, network analysis, and solving optimization problems in various domains.

Floyd-Warshall Algorithm Example

Below is an example of the Floyd-Warshall algorithm in the context of C++:

```cpp
// Floyd-Warshall All Pairs Shortest Path Algorithm
vector<vector<int>> floydWarshallAllPairsShortestPath() {
    for (int i = 0; i < numVertices; i++) {
        distMatrix[i][i] = 0;
    }

    for (int k = 0; k < numVertices; k++) {
        for (int toIndex = 0; toIndex < numVertices; toIndex++) {
            for (int fromIndex = 0; fromIndex < numVertices; fromIndex++) {
                int currentLength = distMatrix[fromIndex][toIndex];
                int possibleLength = distMatrix[fromIndex][k] + distMatrix[k][toIndex];
                if (possibleLength < currentLength) {
                    distMatrix[fromIndex][toIndex] = possibleLength;
                }
            }
        }
    }

    return distMatrix;
}

```

The provided C++ code implements the Floyd-Warshall algorithm for finding the shortest path between all pairs of vertices in a directed or undirected weighted graph. The algorithm utilizes a distance matrix to store the shortest path distances between vertices. It initializes the matrix by setting all values to infinity, except for distances from each vertex to itself which are set to 0. It then iterates through intermediate vertices and updates the matrix by comparing the current distance with the possible distance via the intermediate vertex. If a shorter path is found, the matrix is updated accordingly. By performing a series of updates, the algorithm efficiently computes the shortest path distances between all pairs of vertices. The resulting distance matrix provides valuable information about the shortest paths, allowing for efficient route planning, network analysis, and solving optimization problems in various domains.