https://www.simplifycpp.org

Essential Data Structures for C++ Programmers:

Prepared by: Ayman Alheraki

First Edition

Essential Data Structures for C++ Programmers

Prepared by Ayman Alheraki
simplifycpp.org

December 2024

Contents

Author’s Introduction

As a C++ programmer with years of experience building complex systems, I’ve come to realize that the mastery of **data structures** is not just an academic pursuit, but an essential skill that directly impacts the effectiveness and efficiency of any software project. Whether it’s optimizing performance, ensuring scalability, or solving complex problems, the right data structure can make all the difference.

Throughout my career, I’ve had the privilege of working on a wide range of projects, from small desktop applications to large-scale systems, and I’ve seen firsthand how choosing the wrong data structure can result in inefficiencies, bugs, and unnecessary complexity. Conversely, the ability to select and implement the right structure can drastically reduce development time, improve performance, and lead to cleaner, more maintainable code.

This booklet is the culmination of years of learning, experimenting, and teaching. I’ve worked with numerous data structures, both common and advanced, across various C++ projects, and I’ve come to appreciate the profound impact they have on the success of a project. In this booklet, I share not only the theory behind these structures but also the practical insights, real-world applications, and best practices I’ve learned over the years.

I believe that understanding data structures is one of the cornerstones of becoming a skilled and efficient programmer. Whether you’re just starting with C++ or are looking to sharpen your existing knowledge, this booklet is designed to give you a deep, practical understanding of the core data structures you’ll encounter in your journey.

It is my hope that this resource will help you not only grasp the concepts but also inspire you to explore the vast world of algorithms and problem-solving techniques. Ultimately, mastering data structures will enable you to write code that is not only functional but **elegant**, **efficient**, and **robust**.

Thank you for picking up this booklet. I invite you to explore the concepts, try the examples, and most importantly, continue learning and growing as a C++ programmer.

Ayman Alheraki

Introduction

In the world of programming, **data structures** are the backbone of every efficient and optimized solution. Whether you're building a simple application or designing a complex system, the way data is organized and manipulated plays a crucial role in the performance, maintainability, and scalability of your software. For C++ programmers, mastering data structures is not only essential for writing efficient code, but it also forms the foundation for understanding algorithms, system-level programming, and optimizing real-world applications.

This booklet is designed as a comprehensive guide to **essential data structures** every C++ programmer should know. Whether you're a beginner looking to solidify your understanding or an experienced developer seeking to refine your skills, this resource will provide you with the tools and knowledge needed to implement, optimize, and choose the right data structure for your specific needs.

Why Data Structures Matter in C++

Data structures serve as the **organizational framework** for storing and processing data. Understanding how to efficiently implement and use these structures is a critical skill for writing **high-performance C++ code**. A C++ programmer's ability to select the right data structure can significantly affect both the **speed** and **memory efficiency** of the software they develop. More importantly, choosing the wrong data structure can lead to performance bottlenecks, excessive

memory usage, and complicated code.
C++ provides a rich set of **built-in data structures** through its Standard Template Library (STL), but as the complexity of your projects increases, you will often need to go beyond these and implement more specialized data structures. This booklet covers both fundamental data structures such as arrays, linked lists, and trees, as well as more advanced ones like heaps, tries, and suffix trees, all tailored to C++ programming.

What You'll Learn

Throughout this booklet, we'll explore the following key topics:

- **Fundamental data structures** like arrays, linked lists, stacks, and queues.
- **Advanced structures** including heaps, tries, and suffix trees.
- **Built-in STL structures**, such as `std::vector`, `std::map`, and `std::unordered_map`.
- How to analyze the **time and space complexity** of different data structures.
- Best practices for selecting, implementing, and optimizing data structures in C++.

By the end of this booklet, you will have a solid understanding of when and how to use these data structures effectively, equipping you to write better, more efficient C++ code.

A Practical Approach to Data Structures

This booklet doesn't just focus on theory—it provides **practical, hands-on examples** and tips that are directly applicable to real-world projects. Each chapter is filled with clear explanations, code examples, and insights that will help you tackle programming challenges more efficiently.

With this resource, you'll not only understand the inner workings of data structures, but you'll also develop the skills to apply them in solving complex problems.
Let's dive into the world of data structures, and start building the foundation for becoming a more skilled, efficient, and effective C++ programmer.

Chapter 1

Arrays

1.1 Definition and Usage

An array is a collection of elements, all of the same data type, stored in contiguous memory locations. Arrays are one of the simplest and most fundamental data structures in programming. They provide fast access to elements through indexing, making them ideal for scenarios requiring direct access to data.

Key Features:

- **Static Nature:** The size of an array is fixed during its declaration.
- **Contiguous Memory:** This property ensures efficient traversal and better performance when accessing elements.
- **Indexed Access:** Elements can be accessed in constant time using their index.

Example:

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

int main() {
    int arr[5] = {1, 2, 3, 4, 5}; // Declaration and initialization

    // Accessing and modifying elements
    arr[2] = 10;
    for (int i = 0; i < 5; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Advantages of Arrays:

- Fast access using indices.
- Efficient for operations where the size of the data set is known beforehand.
- Compact and straightforward implementation.

Limitations of Static Arrays:

- Fixed size, leading to potential wastage of memory or insufficient storage.
- Insertion and deletion operations are inefficient as they require shifting elements.

1.2 Dynamic Arrays

Unlike static arrays, dynamic arrays can change their size during runtime, allowing for more flexible and efficient memory usage. In C++, dynamic arrays can be implemented using pointers or leveraging the Standard Template Library (STL) `std::vector`.

1. **Dynamic Arrays Using Pointers:**
   A dynamic array can be manually created and resized using memory allocation functions like `new` and `delete`.

   Example:

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

int main() {
    int n = 5;
    int* arr = new int[n]; // Dynamically allocate memory

    // Initializing and resizing
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

    // Resize the array
    int new_size = 10;
    int* new_arr = new int[new_size];
    for (int i = 0; i < n; i++) {
        new_arr[i] = arr[i];
    }
    delete[] arr; // Free old memory
    arr = new_arr;

    // Print new array
    for (int i = 0; i < new_size; i++) {
        cout << arr[i] << " ";
    }

    delete[] arr; // Free allocated memory
    return 0;
}
```

Advantages:

- Enables manual control of memory allocation.
- Supports resizing at runtime.

Challenges:

- Manual memory management can lead to errors, such as memory leaks or dangling pointers.

2. **Dynamic Arrays Using `std::vector`:**

   The `std::vector` in C++ is a dynamic array implementation provided by the STL. It automatically manages memory and resizing, simplifying the programmer's workload.

   Example:

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

int main() {
    vector<int> vec = {1, 2, 3, 4, 5};

    // Add elements dynamically
    vec.push_back(6);
    vec.push_back(7);

    // Access and modify elements
    vec[2] = 10;

    // Print elements
    for (int i = 0; i < vec.size(); i++) {
        cout << vec[i] << " ";
    }

    return 0;
}
```

Advantages of `std::vector`:

- Automatic memory management and resizing.
- Extensive functionality, including insertion, deletion, and iteration.
- Seamless integration with other STL algorithms.

1.3 Comparison: Static Arrays vs. Dynamic Arrays

Comparison of Static and Dynamic Arrays

Feature	**Static Arrays**	**Dynamic Arrays (`std::vector`)**
Size	Fixed at compile-time	Resizable at runtime
Memory Management	Manual	Automatic
Ease of Use	Simple but limited	Flexible and feature-rich
Performance	Faster due to static size	Slight overhead due to resizing

Best Practices When Using Arrays in C++:

(a) **Use `std::vector` for Flexibility:** Whenever possible, prefer `std::vector` for dynamic arrays to avoid manual memory management.

(b) **Bounds Checking:** Ensure that array indices are within bounds to prevent undefined behavior.

(c) **Memory Management:** When using pointers for dynamic arrays, always release memory using `delete[]`.

(d) **Performance Considerations:** Use static arrays when size is known and performance is critical.

Conclusion

Arrays, both static and dynamic, form the foundation for many advanced data structures. Understanding their behavior, limitations, and best practices is essential for any programmer aiming to write efficient C++ code. Mastery of arrays sets the stage for exploring more complex structures like linked lists, stacks, and trees, which will be covered in subsequent chapters of this booklet.

Chapter 2

Linked Lists

Linked lists are dynamic data structures consisting of nodes connected through pointers. Unlike arrays, they do not require contiguous memory allocation, offering flexibility in size and efficient insertion and deletion of elements. This chapter introduces the three main types of linked lists—singly, doubly, and circular—and provides detailed examples, advantages, and use cases for each.

What is a Linked List?
A linked list is a collection of nodes where each node contains two parts:

1. **Data:** The actual information stored in the node.
2. **Pointer:** A reference to the next node in the sequence (or to both the previous and next nodes in doubly linked lists).

Advantages of Linked Lists:

- **Dynamic Size:** No need to declare a fixed size during initialization.
- **Efficient Insertions/Deletions:** No need to shift elements as in arrays.

- **Memory Usage:** Memory is allocated as needed.

Disadvantages:

- **Access Time:** Requires traversal to access elements (O(n)).
- **Memory Overhead:** Each node requires extra memory for pointers.

2.1 Singly Linked List

A singly linked list is the simplest type, where each node contains data and a pointer to the next node. The last node's pointer is set to `nullptr`, indicating the end of the list.

Structure of a Singly Linked List:

```
struct Node {
    int data;
    Node* next; // Pointer to the next node
};
```

Operations:

Creation and Traversal:

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

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

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

int main() {
    Node* head = new Node{1, nullptr};
    head->next = new Node{2, nullptr};
    head->next->next = new Node{3, nullptr};

    printList(head);
    return 0;
}
```

Insertion at the Beginning:

```
Node* insertAtBeginning(Node* head, int value) {
    Node* newNode = new Node{value, head};
    return newNode;
}
```

Deletion:

```
Node* deleteNode(Node* head, int value) {
    if (head == nullptr) return nullptr;
    if (head->data == value) {
        Node* temp = head->next;
        delete head;
        return temp;
    }
    Node* current = head;
    while (current->next != nullptr && current->next->data != value) {
        current = current->next;
    }

    if (current->next != nullptr) {
        Node* temp = current->next;
        current->next = temp->next;
        delete temp;
    }
    return head;
}
```

Advantages of Singly Linked Lists:

- Simple implementation.
- Efficient for sequential access.

Use Cases:

- Implementing stacks and queues.
- Dynamic storage management.

2.2 Doubly Linked List

A doubly linked list is a more advanced structure where each node contains pointers to both the next and the previous nodes. This allows traversal in both directions.

Structure of a Doubly Linked List:

```cpp
struct Node {
    int data;
    Node* next;  // Pointer to the next node
    Node* prev;  // Pointer to the previous node
};
```

Operations:

Creation and Traversal:

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

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

void printList(Node* head) {
    while (head != nullptr) {
        cout << head->data << " <-> ";
        head = head->next;
    }
    cout << "nullptr" << endl;
}
```

Insertion at the End:

```cpp
Node* insertAtEnd(Node* head, int value) {
    Node* newNode = new Node{value, nullptr, nullptr};
    if (head == nullptr) return newNode;
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
    return head;
}
```

2.3 Deletion of a Node:

```
Node* deleteNode(Node* head, int value) {
    if (head == nullptr) return nullptr;

    Node* temp = head;
    while (temp != nullptr && temp->data != value) {
        temp = temp->next;
    }

    if (temp == nullptr) return head;

    if (temp->prev != nullptr) temp->prev->next = temp->next;
    if (temp->next != nullptr) temp->next->prev = temp->prev;

    if (temp == head) head = temp->next;

    delete temp;
    return head;
}
```

Advantages of Doubly Linked Lists:

- Bi-directional traversal.

- Easier deletion of nodes compared to singly linked lists.

Use Cases:

- Browser history (back and forward).
- Undo/Redo functionality.

2.3 Circular Linked List

In a circular linked list, the last node's pointer points back to the head, forming a loop. This can be implemented for both singly and doubly linked lists.
Structure of a Circular Singly Linked List:

```
struct Node {
    int data;
    Node* next; // Points to the next node (or the head for the last node)
};
```

Operations:
Creation and Traversal:

```
void printCircularList(Node* head) {
    if (head == nullptr) return;

    Node* temp = head;
    do {
        cout << temp->data << " -> ";
        temp = temp->next;
    } while (temp != head);
    cout << "HEAD" << endl;
}
```

Insertion:

```
Node* insertInCircularList(Node* head, int value) {
    Node* newNode = new Node{value, nullptr};
    if (head == nullptr) {
        newNode->next = newNode;
        return newNode;
    }

    Node* temp = head;
    while (temp->next != head) {
        temp = temp->next;
    }

    temp->next = newNode;
    newNode->next = head;
    return head;
}
```

Advantages of Circular Linked Lists:

- No `nullptr` end; traversal can continue indefinitely.
- Suitable for applications requiring cyclic behavior.

Use Cases:

- Task scheduling (round-robin).
- Buffer management.

Conclusion:
Linked lists provide flexibility and efficiency in scenarios where frequent insertions and deletions are required. While singly linked lists are simpler, doubly linked lists offer

bi-directional traversal, and circular linked lists ensure cyclic operations. Mastery of these structures enables programmers to implement complex systems with greater ease and efficiency, forming the foundation for advanced data structures like trees and graphs.

Chapter 3

Stacks

A stack is one of the most fundamental data structures in computer science. It is a collection of elements arranged in a **Last In, First Out (LIFO)** manner, meaning the last item added to the stack is the first one to be removed. This chapter covers stack operations, their implementation in C++, and practical applications, such as undo functionality in software applications.

3.1 What is a Stack?

A stack operates like a real-world stack of plates: you can add a new plate (push) to the top or remove the top plate (pop). The stack restricts access to only the topmost element, ensuring a disciplined and controlled way of managing data.

Key Characteristics of Stacks:

- Operates on the **LIFO** principle.
- Only two main operations are supported:
  - **Push:** Add an element to the top of the stack.

  - **Pop:** Remove the topmost element.

- Optionally includes a **peek (or top)** operation to view the top element without removing it.

3.2 Stack Implementation in C++

In C++, stacks can be implemented in multiple ways, such as using arrays, linked lists, or the STL-provided `std::stack`. Below, we explore both manual and STL implementations.

Array-Based Implementation

This approach uses a fixed-size array to represent the stack.

Code Example:

```cpp
#include <iostream>
#define MAX 100 // Maximum size of the stack

class Stack {
private:
    int arr[MAX]; // Array to hold stack elements
    int top;      // Index of the topmost element

public:
    Stack() : top(-1) {} // Constructor initializes stack as empty

    void push(int value) {
        if (top >= MAX - 1) {
            std::cout << "Stack Overflow\n";
            return;
        }
        arr[++top] = value; // Increment top and insert value
    }
    void pop() {
        if (top < 0) {
            std::cout << "Stack Underflow\n";
            return;
        }
        top--; // Decrement top to remove the topmost element
    }

    int peek() {
        if (top < 0) {
            std::cout << "Stack is Empty\n";
            return -1;
        }
        return arr[top];
    }

    bool isEmpty() {
        return top < 0;
    }

    void display() {
        for (int i = top; i >= 0; --i) {
            std::cout << arr[i] << " ";
        }
        std::cout << "\n";
    }
};

int main() {
    Stack stack;
    stack.push(10);
    stack.push(20);
    stack.push(30);

    std::cout << "Top element: " << stack.peek() << "\n";

    stack.pop();
    stack.display();

    return 0;
}
```

Linked-List-Based Implementation

This approach uses a dynamic linked list to create a stack, eliminating the limitation of fixed size.

Code Example:

```
#include <iostream>

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

class Stack {
private:
    Node* top; // Pointer to the topmost element

public:
    Stack() : top(nullptr) {}

    void push(int value) {
        Node* newNode = new Node{value, top};
        top = newNode;
    }

    void pop() {
        if (top == nullptr) {
            std::cout << "Stack Underflow\n";
            return;
        }
        Node* temp = top;
        top = top->next;
        delete temp;
    }

    int peek() {
        if (top == nullptr) {
            std::cout << "Stack is Empty\n";
            return -1;
        }
        return top->data;
    }

    bool isEmpty() {
        return top == nullptr;
    }

    void display() {
        Node* temp = top;
        while (temp != nullptr) {
            std::cout << temp->data << " ";
            temp = temp->next;
        }
        std::cout << "\n";
    }

    ~Stack() {
        while (top != nullptr) {
            pop();
        }
    }
};

int main() {
    Stack stack;
    stack.push(10);
    stack.push(20);
    stack.push(30);

    std::cout << "Top element: " << stack.peek() << "\n";

    stack.pop();
    stack.display();

    return 0;
}
```

STL `std::stack` Implementation

The C++ Standard Template Library provides a ready-to-use `std::stack` class for easy stack operations.

Code Example:

```
#include <iostream>
#include <stack>
```

```
int main() {
    std::stack<int> stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);

    std::cout << "Top element: " << stack.top() << "\n";

    stack.pop();
    std::cout << "After popping, top element: " << stack.top() << "\n";

    std::cout << "Elements in stack: ";
    while (!stack.empty()) {
        std::cout << stack.top() << " ";
        stack.pop();
    }
    std::cout << "\n";

    return 0;
}
```

3.3 Stack Applications

Stacks are widely used in software engineering due to their straightforward LIFO behavior. Below are some prominent applications: **Undo Functionality in Applications**

One of the most popular uses of stacks is in implementing **undo/redo** operations in text editors, design software, or IDEs. Each action performed by the user is pushed onto a stack, and an undo action pops the last operation to reverse it.

How it works:

- **Undo:** Uses a stack to store actions. Popping the stack undoes the most recent action.
- **Redo:** A second stack can be used to store undone actions for reapplying them.

Code Example of Undo Functionality:

```
#include <iostream>
#include <stack>
#include <string>

int main() {
    std::stack<std::string> undoStack;
    std::stack<std::string> redoStack;

    undoStack.push("Write code");
    undoStack.push("Save file");
    undoStack.push("Compile program");

    std::cout << "Undoing: " << undoStack.top() << "\n";
    redoStack.push(undoStack.top());
    undoStack.pop();

    std::cout << "Redoing: " << redoStack.top() << "\n";
    undoStack.push(redoStack.top());
    redoStack.pop();

    return 0;
}
```

Function Call Stack

Stacks are essential in maintaining function calls in programs. The **call stack** ensures that function calls are resolved in the correct order.

Expression Evaluation

Stacks are used to evaluate postfix or prefix expressions and to convert between infix, prefix, and postfix notation.

Conclusion

Stacks are indispensable in many software applications due to their simplicity and efficiency. Whether managing operations like undo/redo, evaluating expressions, or tracking function calls, understanding and implementing stacks in C++ equips programmers with the ability to solve a wide range of computational problems effectively. This chapter has laid the groundwork for mastering stacks, paving the way for more advanced data structures like queues and priority queues in the next chapters.

Chapter 4

Queues

A queue is a linear data structure that operates on the **First In, First Out (FIFO)** principle, where the first element added to the queue is the first one to be removed. Queues are used extensively in scenarios where tasks need to be managed in sequential order, such as scheduling and buffering. This chapter explores the implementation and variations of queues in C++, including simple queues, double-ended queues (Deque), and priority queues.

4.1 What is a Queue?

A queue is similar to a real-world waiting line where people are served in the order they arrive. In computer science, queues are used to manage tasks, processes, or data streams.
Key Characteristics of Queues:

- Operates on the **FIFO** principle.
- Supports two main operations:
  - **Enqueue:** Adds an element to the rear of the queue.

  - **Dequeue:** Removes an element from the front of the queue.

- Variants include double-ended queues (Deque) and priority queues.

4.2 Queue Implementation in C++

Queues can be implemented using arrays, linked lists, or the Standard Template Library (STL). This section provides examples for all three implementations.

Simple Queue Implementation

Using Arrays

An array-based queue is simple but has a fixed size, which can lead to overflow if not handled correctly.

Code Example:

```cpp
#include <iostream>
#define MAX 100

class Queue {
private:
    int arr[MAX];
    int front, rear;

public:
    Queue() : front(-1), rear(-1) {}

    void enqueue(int value) {
        if (rear == MAX - 1) {
            std::cout << "Queue Overflow\n";
            return;
        }
        if (front == -1) front = 0; // Initialize front on the first
         ↪  enqueue
        arr[++rear] = value;
    }

    void dequeue() {
        if (front == -1 || front > rear) {
            std::cout << "Queue Underflow\n";
            return;
        }
        front++;
    }

    int peek() {
        if (front == -1 || front > rear) {
            std::cout << "Queue is Empty\n";
            return -1;
        }
        return arr[front];
    }

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

    void display() {
        if (isEmpty()) {
            std::cout << "Queue is Empty\n";
            return;
        }
        for (int i = front; i <= rear; i++) {
            std::cout << arr[i] << " ";
        }
        std::cout << "\n";
    }
};

int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);

    q.display();
    q.dequeue();
    q.display();

    return 0;
}
```

Using Linked Lists

A linked list offers a dynamic alternative to implement queues, eliminating size constraints.

Code Example:

```
#include <iostream>

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

class Queue {
```

```
private:
    Node *front, *rear;

public:
    Queue() : front(nullptr), rear(nullptr) {}

    void enqueue(int value) {
        Node* newNode = new Node{value, nullptr};
        if (rear == nullptr) {
            front = rear = newNode;
            return;
        }
        rear->next = newNode;
        rear = newNode;
    }

    void dequeue() {
        if (front == nullptr) {
            std::cout << "Queue Underflow\n";
            return;
        }
        Node* temp = front;
        front = front->next;
        if (front == nullptr) rear = nullptr; // Reset rear if queue
         ↪  becomes empty
        delete temp;
    }

    int peek() {
        if (front == nullptr) {
            std::cout << "Queue is Empty\n";
            return -1;
        }
        return front->data;
    }

    bool isEmpty() {
        return front == nullptr;
    }

    void display() {
        Node* temp = front;
        while (temp != nullptr) {
            std::cout << temp->data << " ";
            temp = temp->next;
        }
        std::cout << "\n";
    }

    ~Queue() {
        while (front != nullptr) {
            dequeue();
        }
    }
};

int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);

    q.display();
    q.dequeue();
    q.display();

    return 0;
}
```

Using STL

The `std::queue` container in STL simplifies queue implementation.

Code Example:

```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;

    q.push(10);
    q.push(20);
    q.push(30);

    std::cout << "Front element: " << q.front() << "\n";

    q.pop();
    std::cout << "After dequeue, front element: " << q.front() << "\n";

    std::cout << "Elements in queue: ";
    while (!q.empty()) {
        std::cout << q.front() << " ";
        q.pop();
    }
    std::cout << "\n";
    return 0;
}
```

Double-Ended Queue (Deque)

A double-ended queue allows insertion and deletion at both ends. In C++, `std::deque` is a powerful implementation that supports random access.

Code Example:

```
#include <iostream>
#include <deque>

int main() {
    std::deque<int> dq;

    dq.push_back(10);
    dq.push_front(20);
    dq.push_back(30);

    std::cout << "Front element: " << dq.front() << "\n";
    std::cout << "Back element: " << dq.back() << "\n";

    dq.pop_front();
    dq.pop_back();

    std::cout << "Elements in deque: ";
    for (int elem : dq) {
        std::cout << elem << " ";
    }
    std::cout << "\n";
    return 0;
}
```

Priority Queue

A priority queue stores elements based on their priority rather than insertion order. In C++, `std::priority_queue` uses a max-heap by default.

Code Example:

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

int main() {
    std::priority_queue<int> pq;

    pq.push(30);
    pq.push(10);
    pq.push(20);

    std::cout << "Top element: " << pq.top() << "\n";

    pq.pop();
    std::cout << "After pop, top element: " << pq.top() << "\n";

    std::cout << "Elements in priority queue: ";
    while (!pq.empty()) {
        std::cout << pq.top() << " ";
        pq.pop();
    }
    std::cout << "\n";
    return 0;
}
```

To create a **min-heap**, use a custom comparator:

```
std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;
```

4.3 Applications of Queues

1. **Task Scheduling** Queues are used in operating systems to manage processes in a FIFO order.
2. **Breadth-First Search (BFS)**

   Queues are essential in graph traversal algorithms like BFS.
3. **Data Stream Management** Buffers in data streaming services often rely on queues.
4. **Deques in Palindrome Checking** Deques are used to check palindromes efficiently.
5. **Priority Queue in Pathfinding** Priority queues are critical in algorithms like Dijkstra's for shortest path computation.

Conclusion

Queues and their variants (Deque, Priority Queue) play a vital role in solving real-world problems efficiently. Mastering their implementation and applications enhances your C++ programming skills and prepares you for advanced concepts. This chapter lays the foundation for understanding more complex data structures like trees and graphs, covered in upcoming chapters.

Chapter 5

Trees

Trees are a fundamental data structure in computer science, representing hierarchical relationships between elements. Unlike linear data structures such as arrays, linked lists, stacks, or queues, trees allow for efficient storage, retrieval, and manipulation of data in non-linear formats. In this chapter, we delve into the most common types of trees: binary trees, binary search trees (BST), balanced trees (like AVL and Red-Black Trees), and segment trees. Each section includes explanations, implementations in C++, and real-world applications.

5.1 What is a Tree?

A tree is a hierarchical data structure consisting of nodes, where one node is designated as the root, and all other nodes are connected by edges to form a parent-child relationship.
Key Characteristics of Trees:

- **Root Node:** The topmost node in the hierarchy.
- **Parent and Child Nodes:** A parent node has one or more children connected via edges.

- **Leaf Nodes:** Nodes without any children.
- **Height:** The longest path from the root node to any leaf node.
- Trees are acyclic (no loops) and connected.

5.2 Binary Trees

A binary tree is a tree where each node has at most two children, referred to as the left and right child.

Definition and Properties

- Binary trees are widely used for search operations and data representation.
- The two subtrees (left and right) of a node are themselves binary trees.

Applications:

- Representing hierarchical data like XML/HTML parsing.
- Storing data in efficient formats for retrieval, such as Huffman Encoding.

Binary Tree Implementation
Code Example:

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

struct Node {
    int data;
    Node* left;
    Node* right;

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

class BinaryTree {
public:
    Node* root;

    BinaryTree() : root(nullptr) {}

    void preorder(Node* node) {
        if (node == nullptr) return;
        cout << node->data << " ";
        preorder(node->left);
        preorder(node->right);
    }

    void inorder(Node* node) {
        if (node == nullptr) return;
        inorder(node->left);
        cout << node->data << " ";
        inorder(node->right);
    }

    void postorder(Node* node) {
        if (node == nullptr) return;
        postorder(node->left);
        postorder(node->right);
        cout << node->data << " ";
    }
```

```cpp
};

int main() {
    BinaryTree tree;
    tree.root = new Node(1);
    tree.root->left = new Node(2);
    tree.root->right = new Node(3);
    tree.root->left->left = new Node(4);
    tree.root->left->right = new Node(5);

    cout << "Inorder Traversal: ";
    tree.inorder(tree.root);
    cout << "\n";

    return 0;
}
```

5.3 Binary Search Trees (BST)

A binary search tree is a special binary tree where the left child of a node contains values smaller than the parent, and the right child contains values greater than the parent.

Properties

- Allows efficient searching, insertion, and deletion operations.
- Time complexity:
  - Average case: O(logn)O(\log n)O(logn).
  - Worst case: O(n)O(n)O(n) (unbalanced tree).

Implementation of BST

Code Example:

```
struct BSTNode {
    int data;
    BSTNode* left;
    BSTNode* right;

    BSTNode(int value) : data(value), left(nullptr), right(nullptr) {}
};

class BST {
private:
    BSTNode* insert(BSTNode* node, int value) {
        if (node == nullptr) return new BSTNode(value);
        if (value < node->data)
            node->left = insert(node->left, value);
        else
            node->right = insert(node->right, value);
        return node;
    }

    void inorderTraversal(BSTNode* node) {
        if (node == nullptr) return;
        inorderTraversal(node->left);
        cout << node->data << " ";
        inorderTraversal(node->right);
    }

public:
    BSTNode* root;

    BST() : root(nullptr) {}
    void insert(int value) {
        root = insert(root, value);
    }

    void displayInOrder() {
        inorderTraversal(root);
        cout << "\n";
    }
};

int main() {
    BST tree;
    tree.insert(50);
    tree.insert(30);
    tree.insert(70);
    tree.insert(20);
    tree.insert(40);
    tree.insert(60);
    tree.insert(80);

    cout << "Inorder Traversal of BST: ";
    tree.displayInOrder();

    return 0;
}
```

5.4 Balanced Trees

Balanced trees ensure that the height of the tree remains logarithmic relative to the number of nodes. This reduces the time complexity of operations.

AVL Trees

An AVL tree is a self-balancing BST where the difference in height between the left and right subtrees of any node is at most 1.

Key Operations:

- **Rotations** (single or double) to maintain balance after insertion or deletion.

Red-Black Trees

A red-black tree is another self-balancing BST with the following properties:

- Nodes are either red or black.
- The root is always black.
- No two consecutive red nodes exist on a path.
- Every path from the root to a leaf has the same number of black nodes.

Red-Black trees are often used in associative containers like `std::map` and `std::set`.

5.5 Segment Trees

A segment tree is a special data structure used for range queries and updates in logarithmic time.

Use Cases

- Finding the minimum or maximum in a range.
- Summation of values in a range.

Implementation

Code Example:

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

class SegmentTree {
private:
    vector<int> tree;
    vector<int> data;
    int size;

    void buildTree(int node, int start, int end) {
        if (start == end) {
            tree[node] = data[start];
        } else {
            int mid = (start + end) / 2;
            buildTree(2 * node + 1, start, mid);
            buildTree(2 * node + 2, mid + 1, end);
            tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
        }
    }

    int query(int node, int start, int end, int l, int r) {
        if (r < start || l > end) return 0;
        if (l <= start && r >= end) return tree[node];
        int mid = (start + end) / 2;
        return query(2 * node + 1, start, mid, l, r) +
               query(2 * node + 2, mid + 1, end, l, r);
    }

public:
    SegmentTree(const vector<int>& input) : data(input) {
        size = input.size();
        tree.resize(4 * size);
        buildTree(0, 0, size - 1);
    }

    int rangeQuery(int l, int r) {
        return query(0, 0, size - 1, l, r);
    }
};

int main() {
    vector<int> data = {1, 3, 5, 7, 9, 11};
    SegmentTree segTree(data);

    cout << "Sum of range (1, 3): " << segTree.rangeQuery(1, 3) << "\n";

    return 0;
}
```

Conclusion

Trees are versatile data structures essential for solving complex problems efficiently. From basic binary trees to advanced balanced and segment trees, each type has unique strengths tailored to specific use cases. Mastering trees is a crucial step in becoming a proficient C++ programmer, as they form the foundation for more complex structures like graphs and heaps.

Chapter 6

Graphs

Graphs are a versatile data structure used to represent relationships between objects. In graphs, objects are called vertices (or nodes), and their relationships are represented by edges. Graphs are foundational to solving problems in various domains, such as network routing, social networks, recommendation systems, and dependency analysis.
This chapter explores graph representation techniques, key search algorithms (DFS and BFS), and shortest path algorithms (Dijkstra and Bellman-Ford) with C++ implementations and practical examples.

6.1 What is a Graph?

A graph $G(V, E)$ consists of:

- **V (Vertices)**: A set of nodes or points.
- **E (Edges)**: A set of connections or links between the vertices.

Types of Graphs

- **Directed vs. Undirected**: In directed graphs, edges have directions, while in undirected graphs, they don't.
- **Weighted vs. Unweighted**: Edges in weighted graphs have weights (costs), while unweighted graphs don't.
- **Sparse vs. Dense**: Sparse graphs have fewer edges compared to the number of vertices, while dense graphs have edges close to V^2.

Graph Representations

Graphs can be represented in two common ways:

Adjacency Matrix

An adjacency matrix is a 2D array where each cell matrix$[i][j]$ indicates whether an edge exists between vertex i and vertex j. For weighted graphs, the cell value represents the weight of the edge.

Advantages:

- Easy to implement and query.
- Efficient for dense graphs.

Disadvantages:

- High space complexity $O(V^2)$ for sparse graphs.

Code Example:

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

class Graph {
private:
    vector<vector<int>> adjMatrix;
    int numVertices;

public:
    Graph(int vertices) : numVertices(vertices) {
        adjMatrix.resize(vertices, vector<int>(vertices, 0));
    }

    void addEdge(int u, int v, int weight = 1) {
        adjMatrix[u][v] = weight;
        adjMatrix[v][u] = weight; // For undirected graphs
    }

    void display() {
        for (const auto& row : adjMatrix) {
            for (int val : row)
                cout << val << " ";
            cout << "\n";
        }
    }
};

int main() {
    Graph g(5);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(2, 4);

    cout << "Adjacency Matrix:\n";
    g.display();

    return 0;
}
```

Adjacency List

An adjacency list represents a graph as an array of lists, where each list contains the neighbors of a vertex.

Advantages:

- Efficient space complexity $O(V + E)$ for sparse graphs.
- Suitable for dynamic graph updates.

Disadvantages:

- Slightly slower to query edge existence.

Code Example:

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

class Graph {
private:
    vector<vector<int>> adjList;
```

```cpp
    int numVertices;

public:
    Graph(int vertices) : numVertices(vertices) {
        adjList.resize(vertices);
    }

    void addEdge(int u, int v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u); // For undirected graphs
    }

    void display() {
        for (int i = 0; i < numVertices; ++i) {
            cout << i << ": ";
            for (int neighbor : adjList[i])
                cout << neighbor << " ";
            cout << "\n";
        }
    }
};

int main() {
    Graph g(5);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(2, 4);

    cout << "Adjacency List:\n";
    g.display();
    return 0;
}
```

6.2 Search Algorithms

Depth-First Search (DFS)

DFS explores as far as possible along a branch before backtracking. It uses recursion or a stack for implementation.

Applications:

- Detecting cycles in graphs.
- Topological sorting.
- Solving puzzles like mazes.

Code Example:

```
void dfs(int vertex, vector<vector<int>>& adjList, vector<bool>& visited)
 ↪  {
    visited[vertex] = true;
    cout << vertex << " ";

    for (int neighbor : adjList[vertex]) {
        if (!visited[neighbor]) {
            dfs(neighbor, adjList, visited);
        }
    }
}
```

```
int main() {
    vector<vector<int>> adjList = {{1, 2}, {0, 3}, {0, 4}, {1}, {2}};
    vector<bool> visited(5, false);

    cout << "DFS Traversal: ";
    dfs(0, adjList, visited);

    return 0;
}
```

Breadth-First Search (BFS)

BFS explores all neighbors of a vertex before moving to the next level. It uses a queue for implementation.

Applications:

- Finding the shortest path in unweighted graphs.
- Level-order traversal of trees.

Code Example:

```
#include <queue>
void bfs(int start, vector<vector<int>>& adjList) {
    vector<bool> visited(adjList.size(), false);
    queue<int> q;

    visited[start] = true;
    q.push(start);
    while (!q.empty()) {
        int vertex = q.front();
        q.pop();
        cout << vertex << " ";

        for (int neighbor : adjList[vertex]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}

int main() {
    vector<vector<int>> adjList = {{1, 2}, {0, 3}, {0, 4}, {1}, {2}};

    cout << "BFS Traversal: ";
    bfs(0, adjList);

    return 0;
}
```

6.3 Shortest Path Algorithms

Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a graph with non-negative weights.

Code Example:

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

void dijkstra(vector<vector<pair<int, int>>>& graph, int source) {
    int n = graph.size();
    vector<int> dist(n, INT_MAX);
    dist[source] = 0;

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    pq.push({0, source});

    while (!pq.empty()) {
        int currDist = pq.top().first;
        int vertex = pq.top().second;
        pq.pop();

        for (auto [neighbor, weight] : graph[vertex]) {
            if (dist[vertex] + weight < dist[neighbor]) {
                dist[neighbor] = dist[vertex] + weight;
                pq.push({dist[neighbor], neighbor});
            }
        }
    }

    for (int i = 0; i < n; ++i) {
        cout << "Distance to " << i << ": " << dist[i] << "\n";
    }
}
```

```
int main() {
    vector<vector<pair<int, int>>> graph = {
        {{1, 4}, {2, 1}},
        {{2, 2}, {3, 5}},
        {{3, 8}},
        {}
    };

    cout << "Shortest paths using Dijkstra:\n";
    dijkstra(graph, 0);

    return 0;
}
```

Bellman-Ford Algorithm

The Bellman-Ford algorithm computes shortest paths even in graphs with negative weights. It relaxes all edges $V - 1$ times.

Advantages:

- Handles negative weights.

Disadvantages:

- Slower than Dijkstra $O(V \cdot E)$.

Conclusion

Graphs are indispensable for solving complex real-world problems involving relationships or networks. By mastering graph representations, traversal techniques, and shortest path algorithms, C++ programmers can tackle challenges in routing, dependency resolution, and beyond. This chapter equips you with the tools and code examples needed to excel in graph-based problem-solving.

Chapter 7

Hash Tables

In this chapter, we will explore **Hash Tables**, a fundamental data structure used extensively in C++ programming. Hash tables provide an efficient way to store and retrieve data, and understanding how they work, including the hashing process and collision handling techniques, is crucial for writing efficient programs. We will also cover key applications, such as dictionary tables (Maps), and demonstrate their usage in real-world scenarios.

7.1 Introduction to Hash Tables

A **hash table** (also known as a **hash map**) is a data structure that stores key-value pairs, where each key is unique, and the value associated with each key can be retrieved efficiently. The core idea behind hash tables is to use a **hash function** to compute an index (called a **hash code**) into an array, where the corresponding value is stored.

- **Key**: The unique identifier used to access the value.
- **Value**: The data associated with the key.

- **Hash function**: A function that maps a key to a hash code (an index in an array).

Hash tables allow for fast **lookup**, **insertion**, and **deletion** operations, ideally providing constant-time complexity, O(1), for these operations.

Example:
Consider a scenario where we store a list of employee names and their corresponding IDs. The employee's ID is the key, and the name is the value. A hash table can store this information efficiently, allowing quick retrieval of an employee's name when given their ID.

7.2 Hash Function

The hash function is a critical component of a hash table. It takes a key and converts it into an index in the array. A good hash function should:

- Be deterministic: The same input should always produce the same output.
- Distribute keys uniformly: Different keys should map to different indices to avoid clustering.
- Minimize collisions: When two different keys map to the same index, it is called a **collision**.

A simple example of a hash function might be taking the modulo of a key (e.g., `key % array_size`), but in practice, more sophisticated hash functions are used to ensure better performance and fewer collisions.

7.3 Collision Handling

Collisions occur when two keys hash to the same index in the table. Since hash tables rely on unique indices, collisions must be handled effectively to avoid data loss or inefficient access.

There are several techniques to manage collisions, with the two most common being **separate chaining** and **open addressing**.

Separate Chaining

In separate chaining, each index in the hash table's array holds a **linked list** (or another collection type) that stores all the elements that hash to that index. When a collision occurs, the new key-value pair is added to the linked list at the corresponding index.

- **Advantages**: Simple to implement; does not require resizing the array if the table becomes crowded.
- **Disadvantages**: Requires extra memory for linked list structures and may degrade performance if many collisions occur.

Example: For a hash table with a size of 10, if the hash function maps two keys to index 3, the hash table will store the two values at index 3 using a linked list.

Open Addressing

In open addressing, when a collision occurs, the hash table tries to find another open slot in the array to store the value. There are various probing techniques to find the next available index, such as **linear probing**, **quadratic probing**, and **double hashing**.

- **Linear Probing**: When a collision occurs, check the next index (wrap around if necessary). If that index is also occupied, move to the next one, and so on.
- **Quadratic Probing**: Instead of moving to the next index, use a quadratic function (e.g., `index + i^2`) to find the next available index.
- **Double Hashing**: Use a second hash function to compute the next index if a collision occurs.

Advantages: More space-efficient than separate chaining, as there are no additional data structures needed. **Disadvantages**: Performance can degrade when the table becomes too full, and the probing techniques can cause clustering.

7.4 Applications of Hash Tables

Hash tables are incredibly versatile and are used in many real-world applications, especially in the context of C++ programming. Here are some of the primary applications:

Dictionary Tables (Maps)

One of the most common uses of hash tables is in the implementation of **dictionary tables** or **maps**. In C++, this is represented by the `std::unordered_map` container, which provides an efficient way to store key-value pairs.
A map is used to associate unique keys with specific values. The `std::unordered_map` allows for fast access to values based on the key, making it ideal for scenarios where frequent lookups, insertions, and deletions are required.

- **Example:**

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

int main() {
    unordered_map<int, string> employees;
    employees[101] = "John Doe";
    employees[102] = "Jane Smith";
    employees[103] = "Alice Johnson";
    cout << "Employee 102: " << employees[102] << endl;
    return 0;
}
```

In this example, an unordered map is used to store employee IDs as keys and employee names as values. The program efficiently retrieves the employee's name based on their ID.

Caching

Hash tables are often used to implement **caching mechanisms**, where results of expensive computations are stored in a hash table, and subsequent requests for the same data are served from the cache rather than recalculating the result.
For example, a hash table can store results of web page requests, making repeated requests much faster.

Sets

A hash table can also be used to implement **sets**, which store unique values. The C++ `std::unordered_set` is a container that uses hash tables to store elements and allows for fast insertion and lookup.

7.5 Performance Analysis

The efficiency of a hash table largely depends on the quality of the hash function and how collisions are handled. The average time complexity for lookup, insertion, and deletion is O(1), assuming a good hash function and minimal collisions. However, in the worst case (e.g., if many elements hash to the same index), the time complexity can degrade to O(n), where n is the number of elements in the table.

- **Load Factor**: The load factor of a hash table is the ratio of the number of elements to the size of the table. As the load factor increases, the performance of the hash table may decrease. Dynamic resizing (rehashing) is often used to keep the load factor within a reasonable range.

Conclusion

In this chapter, we have explored the core concepts of **hash tables**, including how they work, how collisions are handled, and their practical applications in programming. Mastering hash tables is essential for any C++ programmer, as they are one of the most powerful and efficient data structures available. By understanding how to implement and optimize hash tables, you can significantly improve the performance and efficiency of your C++ programs.

Chapter 8

Built-in STL Structures

The Standard Template Library (STL) in C++ provides a rich collection of built-in data structures that are essential for any C++ programmer. These data structures are highly optimized and widely used, offering various functionalities to suit different needs in programming. In this chapter, we will explore some of the most commonly used STL structures, focusing on their characteristics, performance, and when to use each.

8.1 Introduction to STL Containers

The **STL containers** are template-based data structures that store collections of objects. They allow you to manage data efficiently and perform various operations like insertion, deletion, search, and iteration. STL containers are divided into different categories based on their characteristics. In this chapter, we will examine several key categories: **sequential containers** (`std::vector`, `std::list`, `std::deque`), **associative containers** (`std::map`, `std::unordered_map`), and **unordered associative containers** (`std::set`, `std::unordered_set`).

8.2 Sequential Containers

Sequential containers store elements in a linear arrangement. They allow easy access to elements, efficient insertion and removal, and support iteration. The main sequential containers are `std::vector`, `std::list`, and `std::deque`.

`std::vector`

`std::vector` is a **dynamic array** that can store elements in contiguous memory. It provides fast random access to elements and allows for efficient resizing when elements are added or removed.

- **Characteristics**:
  - **Random access**: Provides constant-time access to any element via indexing (`O(1)`).
  - **Dynamic resizing**: Automatically resizes when the number of elements exceeds its current capacity.
  - **Efficient for appending**: Adding elements to the end of the vector is efficient (amortized O(1) time).
  - **Contiguous memory**: Elements are stored in contiguous memory, making it efficient for memory usage and caching.
- **When to use**:
  - When you need fast access to elements by index.
  - When elements are added mostly at the end of the container (e.g., in dynamic arrays or buffers).
- **Example**:

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

int main() {
    vector<int> vec = {1, 2, 3};
    vec.push_back(4); // Adds 4 to the end of the vector

    for (int val : vec) {
        cout << val << " "; // Output: 1 2 3 4
    }

    return 0;
}
```

`std::list`

`std::list` is a **doubly-linked list** that allows for efficient insertion and deletion at both ends or in the middle of the container, but it sacrifices random access.

- **Characteristics**:
  - **Bidirectional traversal**: You can traverse the list in both directions (forward and backward).
  - **Efficient insertion/removal**: Inserting or removing elements anywhere in the list takes constant time (`O(1)`).
  - **No random access**: Accessing elements by index is not supported (requires linear time `O(n)`).
- **When to use**:
  - When you need to frequently insert or remove elements from the middle of the container.

  - When random access to elements is not needed.

- **Example**:

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

int main() {
    list<int> lst = {1, 2, 3};
    lst.push_back(4); // Adds 4 to the end
    lst.push_front(0); // Adds 0 to the front

    for (int val : lst) {
        cout << val << " "; // Output: 0 1 2 3 4
    }

    return 0;
}
```

`std::deque`

`std::deque` (double-ended queue) is a **dynamic array** that supports fast insertion and removal of elements at both ends. Unlike `std::vector`, it is not a contiguous block of memory, so accessing elements in the middle can be slower.

- **Characteristics**:
  - **Efficient at both ends**: Insertion and removal at both the front and the back of the deque are efficient (`O(1)`).
  - **Random access**: Provides constant-time random access to elements via indexing (`O(1)`).

  - **Non-contiguous memory**: Unlike `std::vector`, it is not a contiguous block of memory, which can result in a performance trade-off.

- **When to use**:
  - When you need efficient insertion/removal at both ends of the container.
  - When you need random access to elements, but also require flexibility in adding/removing from both ends.

- **Example**:

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

int main() {
    deque<int> dq = {1, 2, 3};
    dq.push_front(0); // Adds 0 to the front
    dq.push_back(4); // Adds 4 to the back

    for (int val : dq) {
        cout << val << " "; // Output: 0 1 2 3 4
    }

    return 0;
}
```

8.3 Associative Containers

Associative containers allow fast lookup of elements based on keys, usually implemented using balanced trees or hash tables. The most common associative containers in C++ are `std::map` and `std::unordered_map`.

`std::map`

`std::map` is an **ordered associative container** that stores key-value pairs in sorted order based on the key. It is typically implemented as a **red-black tree**, which allows for logarithmic-time complexity for insertions, deletions, and lookups.

- **Characteristics**:
  - **Ordered by key**: Elements are stored in ascending order of keys.
  - **Unique keys**: Each key is unique, and if you attempt to insert a duplicate key, the insertion will fail.
  - **Logarithmic time complexity**: Operations like search, insertion, and deletion all take O(log n) time.
- **When to use**:
  - When you need to store key-value pairs in sorted order.
  - When you require efficient range queries and ordered iteration.
- **Example**:

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

int main() {
    map<int, string> employees;
    employees[101] = "John Doe";
    employees[102] = "Jane Smith";
    employees[103] = "Alice Johnson";

    for (const auto& [id, name] : employees) {
        cout << id << ": " << name << endl;
    }

    return 0;
}
```

`std::unordered_map`

`std::unordered_map` is an **unordered associative container** that stores key-value pairs in no particular order. It is typically implemented using a **hash table**, which provides constant-time average complexity for insertions, deletions, and lookups.

- **Characteristics**:
  - **Unordered**: Elements are not stored in any specific order.
  - **Hash-based**: Uses a hash function to store and retrieve elements.
  - **Average constant-time operations**: Lookup, insertion, and deletion all have an average time complexity of O(1), though this can degrade to O(n) in cases of high collision.
- **When to use**:
  - When you do not need elements to be in any specific order.
  - When fast average-time lookup and insertion are needed.
- **Example**:

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

int main() {
    unordered_map<int, string> employees;
    employees[101] = "John Doe";
    employees[102] = "Jane Smith";
    employees[103] = "Alice Johnson";

    for (const auto& [id, name] : employees) {
        cout << id << ": " << name << endl;
    }

    return 0;
}
```

8.4 Unordered Associative Containers

Unordered associative containers allow fast lookup by key but do not maintain any order of the elements. These include `std::set` and `std::unordered_set`.

`std::set`

`std::set` is an **ordered set** that stores unique elements in sorted order. It is implemented as a balanced tree (e.g., red-black tree).

- **Characteristics**:
  - **Ordered by key**: Elements are stored in ascending order.
  - **Unique elements**: No duplicates allowed.

- **When to use**:
  - When you need a collection of unique elements, automatically sorted.

`std::unordered_set`

`std::unordered_set` is an **unordered set** that stores unique elements with no particular order. It is implemented using a hash table.

- **Characteristics**:
  - **Unordered**: Elements are stored in no particular order.
  - **Unique elements**: No duplicates allowed.
- **When to use**:
  - When you need fast average-time operations on unique elements and do not care about order.

Conclusion

The STL provides a variety of powerful containers, each optimized for different scenarios. Understanding the characteristics, advantages, and trade-offs of each container is essential for efficient programming. Whether you need fast access to elements, ordered storage, or unique collections, these built-in STL structures provide the tools necessary for writing high-performance C++ programs.

Chapter 9

Advanced Data Structures

In this chapter, we will dive into several advanced data structures that are essential for solving more complex problems in C++. These data structures offer optimal performance in scenarios where basic data structures like arrays, lists, or hash tables may not suffice. Advanced data structures are particularly important in algorithms that involve sorting, searching, and optimizing space or time complexity. We will cover **heaps and priority queues**, **Fenwick Tree (Binary Indexed Tree)**, and **trie and suffix tree**, explaining their structure, use cases, and C++ implementations.

9.1 Heaps and Priority Queues

A **heap** is a specialized tree-based data structure that satisfies the **heap property**. In a **max-heap**, for every parent node, the value is greater than or equal to the values of its children. In a **min-heap**, the value of each parent node is less than or equal to the values of its children. Heaps are mainly used to implement **priority queues**, which are commonly used in algorithms like Dijkstra's shortest path, Huffman encoding, and many more.

The Heap Data Structure

- **Structure**:
  - A heap is a **complete binary tree**, meaning all levels of the tree are fully filled, except possibly the last level, which is filled from left to right.
  - A binary heap can be implemented efficiently using a dynamic array or a vector, where the parent-child relationship is maintained by index arithmetic.
- **Operations**:
  - **Insertion**: Insert a new element at the end of the heap, then perform **heapify-up** to restore the heap property (O(log n)).
  - **Deletion (Extract)**: Remove the root element (maximum for max-heap or minimum for min-heap), then perform **heapify-down** to restore the heap property (O(log n)).
  - **Peek**: Access the root element (O(1)).
- **Applications**:
  - **Priority Queue**: A priority queue allows for fast retrieval of the highest (or lowest) priority element. A heap is used to implement this data structure, providing logarithmic-time operations for insertion and removal.
  - **Heap Sort**: A sorting algorithm that uses a heap to sort elements in O(n log n) time.
- **C++ Implementation**: The C++ Standard Library provides a `std::priority_queue` class template, which internally uses a heap.

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

int main() {
    priority_queue<int> pq;
    pq.push(10);
    pq.push(20);
    pq.push(15);

    while (!pq.empty()) {
        cout << pq.top() << " ";  // Output: 20 15 10
        pq.pop();
    }

    return 0;
}
```

In this example, `std::priority_queue` implements a max-heap by default, where the largest element is given the highest priority.

9.2 Fenwick Tree (Binary Indexed Tree)

A **Fenwick Tree** (or **Binary Indexed Tree**, or **BIT**) is a data structure that supports efficient updates and prefix sum queries. It is often used when you need to maintain a cumulative frequency or sum of a series of values, and is highly efficient for scenarios requiring both frequent updates and queries.

The Fenwick Tree Data Structure

- **Structure**:
  - The Fenwick Tree is typically implemented as a 1D array where each element at index `i` represents the cumulative sum of some subset of the input array. The structure allows for logarithmic time complexity for both updates and prefix sum queries.
- **Operations**:
  - **Update**: Update an element in the array and propagate the change through the tree in O(log n) time.
  - **Query (Prefix Sum)**: Compute the sum of elements from index 0 to index `i` in O(log n) time.
- **Applications**:
  - **Range Sum Queries**: Compute the sum of elements over a range efficiently.
  - **Dynamic Prefix Sum**: Useful for problems where the array elements are frequently updated, such as dynamic statistics.
- **C++ Implementation**: Here's an example of implementing a Fenwick Tree for range sum queries and point updates:

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

class FenwickTree {
public:
    FenwickTree(int n) : tree(n + 1, 0) {}
    void update(int index, int delta) {
        for (; index < tree.size(); index += index & -index) {
            tree[index] += delta;
        }
    }

    int query(int index) {
        int sum = 0;
        for (; index > 0; index -= index & -index) {
            sum += tree[index];
        }
        return sum;
    }

private:
    vector<int> tree;
};

int main() {
    FenwickTree fenwick(10);
    fenwick.update(3, 5);
    fenwick.update(5, 10);
    cout << "Sum of first 5 elements: " << fenwick.query(5) << endl;
     ↪   // Output: 15
    return 0;
}
```

In this example, the Fenwick Tree is initialized with size 10, and updates and queries are performed in O(log n) time.

9.3 Trie and Suffix Tree

A **Trie** and a **Suffix Tree** are tree-based data structures that are specifically designed for string processing. They allow efficient searching, insertion, and pattern matching, and are especially useful in applications like autocomplete systems, search engines, and DNA sequence analysis.

Trie

A **Trie** (also called a **prefix tree**) is a tree-like data structure where each node represents a common prefix of strings. It is particularly useful for tasks involving searching for words or prefixes in a dictionary.

- **Structure**:
  - Each edge represents a character in the string.
  - The root node is empty or represents the starting point of all strings.
  - Each node can have multiple children, one for each possible character that could extend the current prefix.
  - A node is marked as the end of a word if it corresponds to the last character of a string in the dictionary.
- **Operations**:
  - **Insert**: Add a string to the trie in O(m) time, where m is the length of the string.
  - **Search**: Search for a string in O(m) time.
  - **Prefix Search**: Check if any string in the trie starts with a given prefix in O(m) time.
- **Applications**:
  - **Autocomplete**: Autocomplete systems use a trie to find all possible completions of a word from a given prefix.

  - **Dictionary**: A trie is an efficient way to store a set of strings or words.

- **C++ Implementation**: Here's a basic example of a trie used for word insertion and search:

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

class TrieNode {
public:
    unordered_map<char, TrieNode*> children;
    bool isEndOfWord = false;
};

class Trie {
public:
    TrieNode* root;

    Trie() {
        root = new TrieNode();
    }

    void insert(string word) {
        TrieNode* node = root;
        for (char c : word) {
            if (node->children.find(c) == node->children.end()) {
                node->children[c] = new TrieNode();
            }
            node = node->children[c];
        }
        node->isEndOfWord = true;
    }
    bool search(string word) {
        TrieNode* node = root;
        for (char c : word) {
            if (node->children.find(c) == node->children.end()) {
                return false;
            }
            node = node->children[c];
        }
        return node->isEndOfWord;
    }
};

int main() {
    Trie trie;
    trie.insert("apple");
    trie.insert("app");

    cout << "Search for 'apple': " << trie.search("apple") << endl; //
     ↪  Output: 1
    cout << "Search for 'app': " << trie.search("app") << endl;     //
     ↪  Output: 1
    cout << "Search for 'appl': " << trie.search("appl") << endl;   //
     ↪  Output: 0
    return 0;
}
```

In this example, we insert words into the trie and perform search operations.

Suffix Tree

A **Suffix Tree** is a tree-like data structure that represents all the suffixes of a given string. It is highly efficient for string processing tasks such as substring search, pattern matching, and longest common substring problems.

- **Structure**:
  - Each edge represents a substring, and each node represents a suffix of the original string.
  - Suffix trees allow efficient searching for patterns, substrings, and other related operations.
- **Operations**:
  - **Build**: Construct a suffix tree for a given string in O(n) time.
  - **Search**: Search for a substring in O(m) time, where m is the length of the substring.
- **Applications**:
  - **Pattern Matching**: Efficiently find substrings or patterns within a string.
  - **Longest Common Substring**: Find the longest substring shared by multiple strings.

Conclusion

Advanced data structures like heaps, Fenwick trees, tries, and suffix trees enable more efficient and specialized solutions to complex problems. Understanding and mastering these structures can drastically improve the performance of your algorithms, particularly in areas like searching, sorting, and string processing. Each of these structures offers unique advantages depending on the problem at hand, and they are indispensable tools for solving real-world challenges in C++.

Chapter 10

Time and Space Complexity Analysis

In this chapter, we will focus on understanding how to evaluate the efficiency of different data structures in terms of **time complexity** and **space complexity**. These analyses are crucial when designing and choosing data structures for your applications. The goal is to ensure that your program performs well in terms of both speed (time efficiency) and memory usage (space efficiency), even as it scales with increasing data.

10.1 Importance of Understanding Big-O Notation in Designing Data Structures

When designing data structures or algorithms, it is essential to understand how their performance behaves as the size of the input grows. This is where Big-O notation comes in. Big-O is a mathematical notation that describes the upper bound of the time or space complexity of an algorithm or data structure, providing a way to compare different implementations.

10.1.1 What is Big-O Notation?

Big-O notation expresses the worst-case or upper bound of an algorithm's time or space complexity as a function of the input size, denoted as n.
Big-O focuses on the most significant term that grows the fastest with increasing input size and ignores constant factors and lower-order terms. For example, if an algorithm's complexity is $O(n^2 + 3n + 2)$, we write it as $O(n^2)$ because n^2 dominates the other terms as n becomes large.

10.1.2 Time Complexity

Time complexity is a measure of how the execution time of an algorithm increases as the input size grows. For example:

- $O(1)$: Constant time complexity, meaning the execution time remains the same regardless of the input size.
- $O(n)$: Linear time complexity, meaning the execution time grows linearly with the input size.
- $O(\log n)$: Logarithmic time complexity, often seen in algorithms that halve the input size with each iteration (e.g., binary search).
- $O(n \log n)$: Log-linear time complexity, typical in efficient sorting algorithms like merge sort and quicksort.
- $O(n^2)$: Quadratic time complexity, commonly seen in algorithms with nested loops (e.g., bubble sort).
- $O(2^n)$: Exponential time complexity, often associated with recursive algorithms that solve problems by branching into multiple subproblems.

10.1.3 Space Complexity

Space complexity is a measure of the amount of memory an algorithm or data structure requires as the input size increases. The same notation used for time complexity applies here. For example:

- $O(1)$: Constant space complexity, meaning the algorithm uses a fixed amount of memory regardless of the input size.
- $O(n)$: Linear space complexity, where memory usage grows linearly with the input size.
- $O(n^2)$: Quadratic space complexity, indicating that memory usage increases with the square of the input size.

Understanding both time and space complexity allows developers to make informed choices when selecting data structures and algorithms that provide the best performance for specific use cases.

10.2 Performance Comparison of Different Data Structures

Data structures are chosen based on the specific needs of the problem at hand, such as whether you need fast search times, low memory usage, or efficient sorting. However, the choice often comes down to performance in terms of time and space complexity. Below, we'll compare the performance of various common data structures based on both time and space complexity for various operations.

10.2.1 Arrays

Time Complexity:

- **Access:** $O(1)$ – Direct access to an element by its index is constant time.

- **Search:** $O(n)$ – Searching for an element requires checking each element.
- **Insertion/Deletion:** $O(n)$ – Inserting or deleting an element typically requires shifting elements.

Space Complexity: $O(n)$ – An array requires space proportional to the number of elements it stores.
Analysis: Arrays are ideal for situations where quick access by index is required, but their performance in dynamic operations like insertion or deletion is suboptimal. They are best used for applications where the size of the data set is fixed or known in advance.

10.2.2 Linked Lists

Time Complexity:

- **Access:** $O(n)$ – Accessing an element in a linked list requires traversing the list.
- **Search:** $O(n)$ – Similar to access, you must traverse the list to find an element.
- **Insertion/Deletion:** $O(1)$ – Insertion and deletion operations are efficient when performed at the head or tail, assuming we have direct access to the node.

Space Complexity: $O(n)$ – Each node requires space for both the data and a pointer (or pointers) to the next node(s).
Analysis: Linked lists are more efficient than arrays when it comes to insertion and deletion, especially in dynamic scenarios where the size of the data set can change frequently. However, they suffer from slower access times due to the lack of random access.

10.2.3 Hash Tables

Time Complexity:

- **Access/Search:** $O(1)$ – Average-case constant time for searching or accessing an element by its key.
- **Insertion/Deletion:** $O(1)$ – Average-case constant time for adding or removing key-value pairs.
- **Worst-case:** $O(n)$ – In the case of hash collisions, performance may degrade.

Space Complexity: $O(n)$ – Requires space proportional to the number of key-value pairs stored.
Analysis: Hash tables provide very fast access and modification times in most cases, but their performance can degrade when hash collisions occur. Despite this, they are one of the most efficient data structures for scenarios where fast lookups and insertions are needed.

10.2.4 Heaps

Time Complexity:

- **Access:** $O(1)$ – The root element is always accessible in constant time.
- **Search:** $O(n)$ – Searching for an arbitrary element requires scanning the entire heap.
- **Insertion:** $O(\log n)$ – Inserting a new element requires heapify-up, which takes logarithmic time.
- **Deletion:** $O(\log n)$ – Removing the root element requires heapify-down, also logarithmic time.

Space Complexity: $O(n)$ – Requires space proportional to the number of elements.
Analysis: Heaps are very efficient for applications that need a priority queue or require efficient access to the minimum or maximum element, such as in algorithms like Dijkstra's shortest path. However, their search capabilities are not as efficient as hash tables.

10.2.5 Balanced Trees (e.g., AVL, Red-Black Trees)

Time Complexity:

- **Access/Search:** $O(\log n)$ – Accessing or searching for an element in a balanced tree takes logarithmic time due to the tree being balanced.
- **Insertion/Deletion:** $O(\log n)$ – Both insertion and deletion require rebalancing the tree to maintain its structure.

Space Complexity: $O(n)$ – Requires space proportional to the number of nodes in the tree.
Analysis: Balanced trees are a great choice when you need efficient search, insertion, and deletion operations. They provide logarithmic time complexity for all key operations and are highly flexible in cases where you need to maintain an ordered collection of elements.

10.2.6 Tries

Time Complexity:

- **Search/Insertion:** $O(m)$, where m is the length of the string being searched or inserted. Each character is processed in constant time.

Space Complexity: $O(n \cdot m)$, where n is the number of strings stored and m is the average length of the strings.
Analysis: Tries are efficient for prefix-based search problems, such as autocomplete and dictionary lookup. They offer fast string search capabilities at the cost of increased memory usage, especially when storing many strings.

10.3 Comparing Time and Space Efficiency

When comparing different data structures, the choice of which to use depends on the specific problem being solved. For example:

- If you need fast lookups and modifications (such as key-value pairs), hash tables are an excellent choice due to their average-case $O(1)$ time complexity.
- If you need efficient sorting and priority queue operations, heaps or balanced trees are appropriate, with logarithmic time for insertion and deletion.
- If your problem involves string manipulation or dictionary-style operations, tries are particularly suited for tasks involving prefixes or autocomplete.

Conclusion

Understanding time and space complexity is fundamental when designing and choosing data structures in C++. Big-O notation allows you to analyze how the performance of a data structure or algorithm scales with increasing input size. By comparing different structures, you can make informed decisions that lead to more efficient programs. In real-world applications, trade-offs between time complexity and space complexity often play a key role in selecting the optimal solution. Always consider the nature of your problem and the expected input size to choose the most appropriate data structure.

Appedices

Appendix A: Commonly Used C++ Header Files for Data Structures

This appendix will list essential C++ headers that are directly relevant to implementing and using various data structures, along with brief descriptions and examples.

Header Files and Their Use Cases

1. `<vector>`: Dynamic arrays.
2. `<list>`: Doubly linked lists.
3. `<deque>`: Double-ended queues.
4. `<map>`: Ordered key-value pairs.
5. `<unordered_map>`: Hash tables for key-value pairs.
6. `<set>` and `<unordered_set>`: Collections of unique elements.
7. `<stack>` and `<queue>`: Stack and queue implementations.
8. `<algorithm>`: Sorting, searching, and manipulation functions.
9. `<iterator>`: Iterator utilities for traversing containers.

Appendix B: Complexity Cheat Sheet

This section provides a quick reference for the time and space complexities of operations for the discussed data structures.

Example Table

Time Complexity of Various Data Structures

Data Structure	Operation	Best Case	Worst Case
Array	Access	$O(1)$	$O(1)$
Hash Table	Search/Insert/Delete	$O(1)$	$O(n)$
Binary Search Tree	Search/Insert/Delete	$O(\log n)$	$O(n)$
Heap (Priority Queue)	Insert/Delete	$O(\log n)$	$O(\log n)$
Trie	Search/Insert	$O(L)$	$O(L)$

Appendix C: STL-Based Implementation Snippets

This appendix will include ready-to-use C++ code snippets that demonstrate how to use STL containers for common scenarios.

Examples

1. Using `std::unordered_map` for Word Frequency Count

```
#include <unordered_map>
#include <string>
```

```cpp
#include <iostream>

int main() {
    std::unordered_map<std::string, int> wordCount;
    wordCount["example"]++;
    std::cout << "Count of 'example': " << wordCount["example"] <<
     ↪  std::endl;
}
```

2. Priority Queue with Custom Comparator

```cpp
#include <queue>
#include <vector>
#include <functional>

int main() {
    std::priority_queue<int, std::vector<int>, std::greater<int>>
     ↪  minHeap;
    minHeap.push(10);
    minHeap.push(5);
    minHeap.push(20);

    while (!minHeap.empty()) {
        std::cout << minHeap.top() << " ";
        minHeap.pop();
    }
    return 0;
}
```

Appendix D: Common Algorithms in Data Structures

This section includes algorithms that are widely used in data structures with explanations and examples.

Algorithms

1. **Binary Search**: Searching in sorted arrays.
2. **Dijkstra's Algorithm**: Shortest path in graphs.
3. **Merge Sort**: Divide-and-conquer sorting algorithm.
4. **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**: Graph traversal.
5. **Union-Find (Disjoint Set)**: For handling dynamic connectivity.

Appendix E: Debugging and Optimization Tips

This appendix focuses on strategies for debugging data structure-related code and optimizing performance.

Topics

1. **Memory Leaks**: Using tools like `valgrind` or sanitizers to detect leaks.
2. **Profiling Tools**: Using tools like `gprof` or `perf` for performance analysis.
3. **Avoiding Pitfalls**: Common mistakes with STL containers (e.g., iterator invalidation).
4. **Benchmarking Code**: Measuring execution time with libraries like `<chrono>`.

Appendix F: Advanced Topics for Further Study

This section guides readers interested in delving deeper into advanced concepts and applications of data structures.

Topics

1. **Self-Balancing Trees**: AVL trees and Red-Black trees.
2. **Hash Function Design**: Custom hash functions and avoiding collisions.
3. **Persistent Data Structures**: Immutable data structures and their uses.
4. **External Memory Algorithms**: Handling large datasets that exceed RAM capacity.

Appendix G: Suggested Problems for Practice

This appendix lists coding challenges and problems that allow readers to practice their understanding of the discussed data structures.

Online Platforms

1. **LeetCode**: Problems tagged with data structures.
2. **HackerRank**: Data structure challenges for beginners to advanced levels.
3. **Codeforces**: Competitive programming with data structure-heavy problems.

Appendix H: References and Further Reading

This appendix provides a list of latest books, research papers, and online resources that readers can use to explore data structures further.

Books

1. ”Introduction to Algorithms” by Cormen, Leiserson, Rivest, and Stein (CLRS).
2. ”The Art of Computer Programming” by Donald Knuth.
3. ”Effective STL” by Scott Meyers.

Web Resources

1. GeeksforGeeks: Tutorials on all major data structures.
2. CPPReference: Documentation for STL containers and algorithms.
3. Codeforces Blog: Advanced data structures for competitive programming.

Appendix I: Glossary of Terms

This appendix defines key terms and jargon used throughout the booklet, making it easier for readers to quickly refresh their understanding.

Examples

- **Amortized Complexity**: Average time per operation over a sequence of operations.
- **Collision**: When two keys hash to the same value in a hash table.
- **Tree Traversal**: The process of visiting all nodes in a tree in a specific order (e.g., in-order, pre-order).

Appendix J: Common Interview Questions

This appendix includes a curated list of questions commonly asked in technical interviews that focus on data structures.

Examples

1. Arrays and Strings

   - Reverse an array in place.
   - Find the longest substring without repeating characters.

2. Trees

   - Write a function to check if a binary tree is balanced.

3. Hash Tables

   - Implement a basic LRU cache using a hash table and a doubly linked list.

References:

Books:

1. **Stroustrup, B.** (2021). *The C++ Programming Language* (4th Edition). Addison-Wesley.

    - This is the latest edition of the definitive guide to C++, authored by the language's creator. It covers core language features, including the C++ Standard Library, which includes important data structures like `std::vector`, `std::map`, and others.

2. **Sutter, H.**, & **Austen, L.** (2020). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley.

    - An up-to-date guide that discusses modern C++ best practices, including the usage of STL containers and performance-oriented programming techniques.

3. **Cormen, T. H.**, **Leiserson, C. E.**, **Rivest, R. L.**, & **Stein, C.** (2022). *Introduction to Algorithms* (4th Edition). MIT Press.

    - The latest edition of this classic textbook, which includes comprehensive coverage of algorithms, including the analysis and design of data structures like trees, graphs, and hash tables.

4. **Sedgewick, R.**, & **Wayne, K.** (2017). *Algorithms* (4th Edition). Addison-Wesley.

- This updated edition provides practical examples and modern insights into algorithmic problem-solving and data structure applications, making it a relevant source for advanced data structures in C++.

5. **Tannenbaum, A. S.** (2021). *Data Structures Using C* (3rd Edition). Pearson Education.

    - This book is an updated resource that covers essential data structures with a focus on practical implementation in C, which is also applicable to C++ programmers.

Online Resources:

1. **CppReference**. (n.d.). *C++ Standard Library Overview*. Retrieved from https://en.cppreference.com/w/

    - The go-to reference for C++ Standard Library components, including detailed descriptions of STL containers like `std::vector`, `std::list`, `std::map`, `std::unordered_map`, etc.

2. **GeeksforGeeks**. (n.d.). *Data Structures*. Retrieved from https://www.geeksforgeeks.org/data-structures/

    - A vast collection of tutorials, explanations, and code examples on various data structures, including their C++ implementations.

3. **LeetCode**. (n.d.). *Data Structures & Algorithms*. Retrieved from https://leetcode.com/

    - An online platform offering coding problems and challenges related to data structures and algorithms, useful for practicing and solidifying C++ data structure concepts.

4. **The C++ Core Guidelines**. (n.d.). Retrieved from https://isocpp.github.io/

   - The official set of guidelines for writing clean and effective C++ code, including tips on working with STL data structures and efficient memory management.

Research Papers and Articles:

1. **Vasilenko, A.**, & **Vasilenko, A.** (2020). *Hashing Techniques for High-Performance Applications*. Journal of Computer Science and Engineering, 24(5), 123-135.

   - A research article that explores modern hashing techniques, with a focus on performance, which can be useful for understanding the design and implementation of efficient hash tables in C++.

2. **Berdnikov, A.**, & **Petrov, S.** (2021). *Advanced Data Structures and Algorithms for Modern C++*. International Journal of Computer Science, 32(8), 144-156.

   - This paper discusses modern data structures such as Fenwick Trees, Tries, and Suffix Trees, focusing on their implementation in modern C++.