Contents

Chapter 1

Linked Lists

A linked list is a linear data structure where each element is a separate object, called a node. Each node holds its own data and the address of the next node, thus forming a chain-like structure.

A simple node in a linked list can be represented in C++ as follows:

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

In this structure, 'data' is used to store the data and 'next' is a pointer that holds the address of the next Node in the list.

Here is a simple example of creating and linking nodes in a linked list:

```
// Create nodes
Node* head = new Node();
Node* second = new Node();
Node* third = new Node();
```

```
// Assign data
head->data = 1;
second->data = 2;
third->data = 3;

// Link nodes
head->next = second;
second->next = third;
third->next = nullptr;  // The last node points to null
```

In this example, we first create three nodes using the ‘new‘ keyword, which dynamically allocates memory. We then assign data to the nodes and link them using the ’next’ pointer.

Chapter 2

Trees

Trees are one of the most versatile and widely used data structures in computer science. A tree is a hierarchical data structure consisting of nodes, where each node has a value and a set of references to its child nodes. The node at the top of the hierarchy is called the root, and nodes with the same parent are called siblings.

The power of trees comes from their ability to represent complex relationships between objects, while providing efficient operations for accessing and modifying those objects. Trees can be used to represent hierarchical relationships, to organize data for quick search and insertion, and to manage sorted lists of data, among other uses.

In this chapter, we will delve into the details of the tree data structure. We will start with the definition and properties of trees, including the key concepts of roots, nodes, children, siblings, leaves, and levels. We will then introduce binary trees, a specific type of tree where each node has at most two children, which are referred to as the left child and the right child.

We will explore the various ways to traverse a tree, including depth-first and breadth-first traversals, and discuss the applications and efficiencies of these methods. We will then cover binary search trees, a variant of binary trees that allows for fast lookup, addition, and removal of items.

Then, we'll take a look at balanced search trees, such as AVL trees and red-black trees, which automatically keep their height small to guarantee logarithmic time complexity in the worst case for search, insert, and delete operations.

Finally, we will explore more advanced topics such as B-trees, tries, and suffix trees, which have applications in databases, file systems, and string algorithms.

By the end of this chapter, you will have a deep understanding of the tree data structure, its variants, and their uses. Armed with this knowledge, you'll be able to choose the right tree structure for your data and implement it effectively in your software.

CHAPTER 3

Searching Trees

Chapter 4
Hash Tables

A hash table, also known as a hash map, is a data structure that implements an associative array abstract data type, a structure that can map keys to values. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

4.1 Structure of a Hash Table

A hash table is composed of an array (the 'table') and a hash function. The array has a predetermined size, and each location (or 'bucket') in the array can hold an item (or several items if collisions occur, as will be discussed later). The hash function is a function that takes a key as input and returns an integer, which is then used as an index into the array.

4.2 Inserting and Retrieving Data

When inserting a key-value pair into the hash table, the hash function is applied to the key to compute the index for the array. The corresponding value is then stored at that index.

When retrieving the value associated with a key, the hash function is applied to the key to compute the array index, and the value is retrieved from that index.

4.3 Handling Collisions

A collision occurs when two different keys hash to the same index. There are several methods for handling collisions:

- **Chaining (or Separate Chaining)**: In this method, each array element contains a linked list of all the key-value pairs that hash to the same index. When a collision occurs, a new key-value pair is added to the end of the list.
- **Open Addressing (or Linear Probing)**: In this method, if a collision occurs, we move to the next available slot in the array and store the key-value pair there. When looking up a key, we keep checking slots until we find the key or reach an empty slot.

4.4 Time Complexity

In an ideal scenario where hash collisions do not occur, hash tables achieve constant time complexity $O(1)$ for search, insert, and delete operations. However, due to hash collisions, the worst-case time complexity can become linear $O(n)$, where n is the number of keys inserted into the table.

Using good hash functions and collision resolution strategies can minimize this issue and allow us to take advantage of the hash table's efficient average-case performance.

Chapter 5

Sorting Algorithms

Sorting is a fundamental problem in computer science that has been extensively studied for many years. Sorting is the process of arranging items in ascending or descending order, based on a certain property. In the realm of algorithms, sorting generally refers to the process of rearranging an array of elements according to a specific order. This order could be numerical (ascending or descending) or lexicographical, depending on the nature of the elements.

Sorting algorithms form the backbone of many computer science and software engineering tasks. They are used in a myriad of applications including, but not limited to, data analysis, machine learning, graphics, computational geometry, and optimization algorithms. Thus, understanding these algorithms, their performance characteristics, and their suitability for specific tasks is crucial for anyone venturing into these fields.

This chapter will introduce several sorting algorithms, ranging from elementary methods like bubble sort and insertion sort to more advanced algorithms such as quicksort, mergesort, and heapsort. We will study these algorithms in terms of their time and space complexity, stability, and adaptability, among other characteristics. By the end of this chapter, you should have a solid understanding of how different sorting algorithms work and how to choose the appropriate algorithm for a specific context.

The knowledge of sorting algorithms not only helps in writing efficient code but also strengthens your problem-solving ability and analytical thinking, which are essential skills for succeeding in any technical interview. Let’s dive into this fascinating world of sorting algorithms.

Appendix A

Answers to Exercises

Index