Data Structures and Algorithms

Chapter 2: Intro to Data Structures and Algorithms

Introduction

A key component of creating effective software is understanding data structures and algorithms, which are fundamental concepts in computer science. a data structure specifies how data is stored and organized on a computer, whereas an algorithm outlines a step-by-step process for carrying out a task or solving an issue.

Data Structures

A data structure is a specialized format for organizing and storing data in a computer program. The purpose of using data structures is to make data management and organization more efficient, enabling quicker access and modification of data. Data structures provide a way to manage and store data in a logical and systematic manner, which is critical for solving complex problems and implementing algorithms.

More specifically, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

Common Data Structures:

Linear Data Structures

In linear data structures, elements are arranged in a sequential manner, and each element is connected to the next one in a linear sequence.

1. Array

An array is a collection of elements of the **same data type** stored in **contiguous memory locations.** Arrays provide constant-time access to elements and can be useful for storing lists of data of the same type.

Arrays in C++ have a **fixed size**, which is determined at compile-time. These are also called *static arrays*. Arrays can store elements of any data type, including built-in types (int, float, etc.) and user-defined types (structs, classes).

They are stored in contiguous memory. This means that the elements of an array are stored one after another in memory, without any gaps between them. The memory address of the first element of the array serves as the base address, and each subsequent element is located at a memory address that is offset from the base address by the size of the data type.

2. Linked List

A linked list is a linear data structure where each element is a separate object. Each element (usually called a **node**) contains a reference to its next node and sometimes to its previous node (depending on the type/implementation). Linked lists can be useful for dynamic data structures and inserting or deleting elements.

a. Singly Linked list

A singly linked list is a linear data structure made up of nodes that are connected together via pointers. Each node contains two parts:

Data - The actual data being stored in the node. This could be an integer, string, object, etc.

Pointer - A reference or address to the next node in the list.

The nodes are linked together sequentially where each node points to the next node in the list. The first node is called the **head** while the last node points to null, None, or nullptr (depending on the language) indicating the end of the list.

Some key characteristics of singly linked lists:

- Sequential access - To access or traverse the list, you must start from the head and follow the pointers one-by-one to the end.
- Dynamic size - Linked lists have no fixed size, they can grow or shrink in size by adding/removing nodes.
- Non-contiguous memory - The nodes are stored in any random memory location and linked by address pointers.
- Efficient insertion/deletion - Adding or removing nodes just requires updating the pointer of the previous node.
- No random access - To access a specific node, you have to sequentially traverse from the head node.

There are many types of linked lists. The common ones are:

- Singly Linked List - Each node contains a data field and a reference to the next node in the sequence.
- Double Linked List - Similar to a singly linked list, but each node also contains a reference to the previous node.
- Circular Linked List - This can be either singly or doubly linked, but the last node's next pointer points back to the first node (or in the case of a doubly circular list, both the first and last nodes point to each other).

Example implementation of SLL:

```cpp
#include <iostream>

// A singly linked list consists of nodes where each node contains data and
// a pointer to the next node in the sequence.

class SinglyLinkedList {
private:
    // Definition of a node in the linked list
    struct Node {
        int data;      // The value stored in the node
        Node* next;    // Pointer to the next node in the list

        // Constructor to initialize a node with a given value
        Node(int value) : data{ value }, next{ nullptr } {}
    };

    Node* head; // Pointer to the first node in the list

public:
    // Constructor to initialize the linked list
    SinglyLinkedList() : head{ nullptr } {}

    // Destructor to clean up memory when the list is destroyed
    ~SinglyLinkedList() {
        clear(); // Free all nodes to prevent memory leaks
    }

    // Function to insert a new node at the beginning of the list
    void insertAtBeginning(int value) {
        // Create a new node with the provided value
        Node* newNode{ new Node(value) };

        // Link the new node to the current head of the list
        newNode->next = head;

        // Update the head to point to the new node
        head = newNode;
    }

    // Function to insert a new node at the end of the list
    void insertAtEnd(int value) {
        // Create a new node with the provided value
        Node* newNode{ new Node(value) };

        // Check if the list is empty
        if (!head) {
            // If empty, the new node becomes the head
            head = newNode;
            return;
        }

        // Traverse the list to find the last node
        Node* current{ head };
```

```
        while (current->next) { // Continue until current->next is nullptr
            current = current->next; // Move to the next node
        }

        // Link the last node to the new node
        current->next = newNode;
    }

    // Function to delete the first node that contains the specified value
    void deleteNode(int value) {
        if (!head) return; // If the list is empty, nothing to delete

        // If the node to delete is the head node
        if (head->data == value) {
            Node* temp = head;       // Temporary pointer to hold the current h
ead
            head = head->next;       // Move head to the next node
            delete temp;             // Free memory of the old head
            return;
        }

        Node* current{ head }; // Pointer to traverse the list
        Node* previous{ nullptr }; // Pointer to keep track of the previous nod
e

        // Traverse the list to find the node with the specified value
        while (current && current->data != value) {
            previous = current;             // Update previous to current
            current = current->next;        // Move to the next node
        }

        // If the node was found in the list
        if (current) {
            previous->next = current->next; // Bypass the current node
            delete current;                  // Free memory of the deleted node
        }
        // If the node was not found, do nothing
    }

    // Function to search for a node with the specified value
    bool searchNode(int value) const {
        Node* current{ head }; // Start from the head of the list

        // Traverse the list node by node
        while (current) {
            if (current->data == value)
                return true; // Value found
            current = current->next; // Move to the next node
        }
        return false; // Value not found in the list
    }

    // Function to clear the entire linked list and free memory
    void clear() {
        Node* current{ head }; // Start from the head
```

```
        // Traverse and delete each node
        while (current) {
            Node* nextNode = current->next; // Keep track of the next node
            delete current;                 // Delete the current node
            current = nextNode;             // Move to the next node
        }

        head = nullptr; // Reset head to indicate the list is now empty
    }

    // Function to display the contents of the linked list
    void display() const {
        Node* current{ head }; // Start from the head

        // Traverse the list and print each node's data
        while (current) {
            std::cout << current->data << " -> ";
            current = current->next; // Move to the next node
        }
        std::cout << "nullptr" << '\n'; // Indicate the end of the list
    }
};
```

3. Stack

A stack is a simple data structure that follows the **Last In, First Out** (LIFO) principle. It means that the last element added to the stack is the first one to be removed. Stack can be implemented in many ways, including:

- Array-based stack
- Linked list-based stack
- Circular stack

4. Queue

A queue is a linear data structure that represents a collection of elements where elements can be inserted at one end (rear) and removed from the other end (front). This means that the elements are added in a **first-in, first-out (FIFO**) order.

Queues can be implemented in many ways, including:

- Array-based queue: The size of the array is fixed, and when the queue is full, new elements are not added. Instead, the oldest element is removed from the front of the array to make room for the new element.
- Circular Queue
- Linked list-based queue: The size of the queue is not fixed, and new elements can be added to the end of the list as needed. When an element is removed from the front of the list, the remaining elements are shifted to fill the gap.

- Priority queue: This is a specialized queue that allows elements to be added and removed based on their priority. Elements with higher priorities are added to the front of the queue, and elements with lower priorities are added to the back of the queue. When an element is removed from the front of the queue, it is replaced by the next element with the highest priority.

5. **ArrayList**

   ArrayList is a data structure that is used to store a collection of elements. It is a dynamic array, which means that it can grow or shrink in size as elements are added or removed. This makes it a good choice for situations where the size of the collection is not known ahead of time, or where the size of the collection may change frequently. It is a versatile data structure that can be used to store a variety of different types of data, including integers, strings, and objects.

6. **ArrayBuffer**

   A ring buffer, also known as a circular buffer, is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end.

   This structure lends itself easily to buffering data streams. It's called a "ring buffer" because when the pointer that marks the end of the data gets to the end of the buffer, it loops back around to the beginning.

   One of the key benefits of using a ring buffer is that it is very efficient in terms of memory usage. Because it is implemented as a circular array, it only requires a fixed amount of memory to store the items in the queue, regardless of the size of the queue. It's useful in scenarios where the buffer might be read from and written to at different speeds. If the buffer becomes full, instead of blocking until there's room, it will usually overwrite old data.

Non-Linear Data Structures

1. **Trees**
2. **Graphs**
3. **Maps**
4. **Hash Tables**
5. **LRU cache**

Algorithms

Algorithms are sequences of steps used to carry out tasks or solve certain problems. Their presence is found everywhere in domains such as engineering, mathematics, and computer science. Time complexity—the length of time an algorithm takes to execute—and space complexity—the amount of memory it uses—are two common metrics used to assess an algorithm's efficiency.

Sorting Algorithms

Sorting is a fundamental concept in data structure and algorithms that involves arranging a collection of items in a particular order. The goal of sorting is to find the most efficient way to arrange the items in a collection so that they can be easily searched, accessed, or processed.

Common Sorting algorithms

1. Bubble sort

Bubble sort is a simple sorting algorithm that works by **repeatedly iterating through the collection of items, comparing adjacent items, and swapping them if they are in the wrong order.**

Example implementation:

```
void bubbleSort(int arr[], int size) {
    for (int i{ 0 }; i < size - 1; ++i) {
        for (int j{ 0 }; j < size - 1 - i; ++j) {
            // compare adjacent elements
            if (arr[j] > arr[j + 1]) {
                // swap the elements
                int temp{ arr[j] };
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }

        }
    }
}
```

The code above works, but the bubble sort will keep executing even if the array is already sorted, which is inefficient.

We can optimize the code by **ending the loop when the array is already sorted**. We can do this by adding a bool flag to tell us if swaps happened inside the loop. We can optimize it further by using the swap() function from the C++ standard library which is more efficient than the swap in our previous code which uses a 3rd temporary variable.

2. Insertion sort
3. Merge sort
4. Quicksort
5. Heapsort
6. Radix sort

Searching Algorithms

Searching in DSA refers to the process of finding a specific element(s) or value(s) in a given data structure. It is an essential operation in many algorithms and plays an important role in problem-solving.

1. **Linear search**

   This is the simplest and most basic search algorithm. It traverses the data structure in a **linear order, comparing each element to the target value until a match is found or the end of the data structure is reached.**

   Example implementation:

```
bool linearSearch(int arr[], int size, int val) {
     for (int i{ 0 }; i < size; ++i) {
         if (arr[i] == val) {
            return true;
         }
     }
     return false;
}
```

2. **Binary search**

   The binary search algorithm works by **repeatedly dividing in half the portion of the list that could contain the target value, until you narrow down the possible values to just one.**

   This is done by comparing the middle element of the list to the target value.

   If the middle element matches the target value, the search is successful, and the index of the middle element is returned.

   If the middle element is greater than the target value, the search continues on the left half of the list.

   If the middle element is less than the target value, the search continues on the right half of the list.

Example implementation:

```
bool binarySearch(int arr[], int size, int val) {
     int start{ 0 };
     int end{ size - 1 };

     while (start <= end) {
            int mid{ (start + end) / 2 };
         if (arr[mid] == val) {
               return true;
         } else if (arr[mid] > val) {
               end = mid - 1;
         } else {
               start = mid + 1;
         }
     }
     return false;
}
```

3. Depth-first search
4. Breadth-first search
5. Dijkstra's algorithm
6. Hashing Search