Discussion-6: Collections: Sorting and Searching and Using Array Lists

Discussion Topic:

What are the major differences between searching and sorting in Java? In what instances can searching and sorting algorithms be used? Provide specific examples of each. In response to your peers, provide additional examples of these algorithms and how you foresee using them.

My Post:

Hello class,

The major differences between searching and sorting in Java lie in their purposes and outputs, as well as their efficiencies and time complexities. Please Table 1 for a detailed comparison.

Table 1

Searching vs Sorting in Java

	Searching	Sorting
Definition	The process of finding a specific element in a collection of data	The process of arranging elements in a specific order (e.g., ascending or descending)
Purpose	Locate a particular item	Organize the entire dataset
Output	Position of the target element or indication of absence	Rearranged version of the entire collection
Common Algorithms	Linear Search Binary Search	Bubble Sort Selection Sort Insertion Sort Merge Sort Quick Sort
Time Complexity	Linear Search: O(n) Binary Search: O(log n)	Bubble Sort: O(n^2) Merge Sort: O(n log n) Quick Sort: O(n log n) average, O(n^2) worst case
Prerequisites	Binary Search requires sorted data Linear Search can work on unsorted data	Most sorting algorithms don't require any specific data arrangement
Applications	Database queries	Organizing contact lists Arranging products Preparing data for analysis
Java Methods	Arrays.binarySearch() Collections.binarySearch()	Arrays.sort() Collections.sort()
Effect on Data	Does not modify the original data structure	Modifies the original data structure (unless a new sorted copy is explicitly created)

Efficiency on Large Datasets	Binary Search is very efficient on large, sorted datasets	Efficient algorithms like Merge Sort and Quick Sort are preferred for large datasets

Choosing between different searching or sorting algorithms often depends on the purpose or output wanted and the specific requirements of your application, such as the size of the dataset, and whether the data is already sorted.

The following table, Table 2, gives examples of pseudocode and time complexity for several searches and sort algorithms:

Table 2

Runtime Complexities for Various Pseudocode Examples

Notation	Name	Example pseudocode
O(1)	Constant	`FindMin(x, y) { if (x < y) { return x } else { return y } }`
O(log N)	Logarithmic	`BinarySearch(numbers, N, key) { mid = 0; low = 0; high = 0; high = N - 1; while (high >= low) { mid = (high + low) / 2 if (numbers[mid] < key) { low = mid + 1 } else if (numbers[mid] > key) { high = mid - 1 } else { return mid } } return -1 // not found }`

O(N)	Linear	`LinearSearch(numbers, N, key) { for (i = 0; i < N; ++i) { if (numbers[i] == key) { return i } } return -1 // not found }`
O(N log N)	Log-linear	`MergeSort(numbers, i, k) { j = 0 if (i < k) { j = (i + k) / 2 // Find midpoint MergeSort(numbers, i, j) // Sort left part MergeSort(numbers, j + 1, k) // Sort right part Merge(numbers, i, j, k) // Merge parts } }`
O(N^2)	Quadratic	`SelectionSort(number, N) { for (i = 0; i < N; ++i) { indexSmallest = i for (j = i + 1; j < N; ++j) { if (numbers[j] < numbers[indexSmallest]) { indexSmallest = j } } temp = numbers[i] numbers[i] = numbers[indexSmallest] numbers[indexSmallest] = temp }`
O(cN)	Exponential	`Fibonacci(N) { if ((1 == N) || (2 == N)) { return 1 } return Fibonacci(N-1) + Fibonacci(N-2) }`

Note: In Java without using the Comparable Interface the code above would only be viable for primitive types. From Programming in Java with ZyLabs , 18.3 O notation, Figure 18.3.2 by Lysecky, R., & Lizarraga, A. (2022).

An example of a sort algorithm is the merge sort, which has the divide-and-conquer approach, it recursively divides a data array into smaller subarrays and sorts those subarrays, then merges the subarrays together to create a sorted array (GeeksforGeeks, 2020a).
An example of a search algorithm is the binary search; which operates on a pre-sorted array by repeatedly dividing the search interval in half until the target element is found or determined to be absent (GeeksforGeeks, 2020b).

The example below sorts using merge sort an ArrayList of book objects by year of publication then searches the sorted list using binary:

Book.java

```
/**
 * Book object with a title and publication year. This class implements
 * Comparable to allow sorting based on the publication year.
 *
 * @author Alejandro Ricciardi
 * @version 1.0
 * @date 07/14/2024
 */
class Book implements Comparable<Book> {
    String title;
    int year;

    /**
     * Constructs a new Book object.
     *
     * @param title The title of the book.
     * @param year  The year the book was published.
     */
    public Book(String title, int year) {
       this.title = title;
       this.year = year;
    }

    /**
     * Compares this book with another book based on the publication year.
     *
     * @param other The book to compare with.
     * @return A negative integer, zero, or a positive integer as this book is less
     *         than, equal to, or greater than the specified book.
     */
    @Override
    public int compareTo(Book other) {
       return Integer.compare(this.year, other.year);
    }

    /**
     * Returns a string representation of the book.
     *
     * @return A string in the format "title (year)".
     */
    @Override
    public String toString() {
       return title + " (" + year + ")";
    }
}
```

BookSortingSearching.java

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Scanner;

/**
 * Sorts and search a list of books. It implements merge sort for sorting and
 * binary search for searching.
 *
 * @author Alejandro Ricciardi
 * @version 1.0
 * @date 07/14/2024
 */
public class BookSortingSearching {

    /**
     * The main method that demonstrates sorting and searching on a list of books.
     *
     * @param args Command line arguments (not used).
     */
    public static void main(String[] args) {
       // Initialize the list of books
       ArrayList<Book> books = new ArrayList<>(
              Arrays.asList(new Book("To Kill a Mockingbird", 1960), new Book("1984",
1949),
                     new Book("The Great Gatsby", 1925), new Book("One Hundred Years of
Solitude", 1967),
                     new Book("The Catcher in the Rye", 1951), new Book("Brave New
World", 1932),
                     new Book("The Hobbit", 1937), new Book("The Lord of the Rings",
1954),
                     new Book("Pride and Prejudice", 1813), new Book("Animal Farm",
1945")));

       // Print the original list
       System.out.println("Original list:");
       books.forEach(System.out::println);

       // Sort the books using merge sort
       mergeSort(books, 0, books.size() - 1);

       // Print the sorted list
       System.out.println("\nSorted list by year:");
       books.forEach(System.out::println);

       // Perform binary search based on user input
       Scanner scn = new Scanner(System.in);
       System.out.print("\nEnter a year to search for: ");
```

```
        int searchYear = scn.nextInt();

        int result = binarySearch(books, searchYear);
        if (result != -1) {
            System.out.println("Book found: " + books.get(result));
        } else {
            System.out.println("No book found for the year " + searchYear);
        }
        scn.close();
    }

    /**
     * Sorts the given list of books using the merge sort algorithm.
     *
     * @param books The list of books to sort.
     * @param left  The starting index of the subarray to sort.
     * @param right The ending index of the subarray to sort.
     */
    private static void mergeSort(ArrayList<Book> books, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;
            mergeSort(books, left, mid); // Sort left half
            mergeSort(books, mid + 1, right); // Sort right half
            merge(books, left, mid, right); // Merge the sorted halves
        }
    }

    /**
     * Merges two sorted subarrays of the books list.
     *
     * @param books The list of books containing the subarrays to merge.
     * @param left  The starting index of the left subarray.
     * @param mid   The ending index of the left subarray.
     * @param right The ending index of the right subarray.
     */
    private static void merge(ArrayList<Book> books, int left, int mid, int right) {
        // Create temporary arrays
        ArrayList<Book> leftList = new ArrayList<>(books.subList(left, mid + 1));
        ArrayList<Book> rightList = new ArrayList<>(books.subList(mid + 1, right + 1));

        int i = 0, j = 0, k = left;

        // Merge the two lists
        while (i < leftList.size() && j < rightList.size()) {
            if (leftList.get(i).compareTo(rightList.get(j)) <= 0) {
                books.set(k++, leftList.get(i++));
            } else {
                books.set(k++, rightList.get(j++));
            }
        }
```

```
        // Copy remaining elements of leftList, if any
        while (i < leftList.size()) {
            books.set(k++, leftList.get(i++));
        }

        // Copy remaining elements of rightList, if any
        while (j < rightList.size()) {
            books.set(k++, rightList.get(j++));
        }
    }

    /**
     * Performs a binary search on the sorted list of books to find a book by its
     * publication year.
     *
     * @param books The sorted list of books to search.
     * @param year  The publication year to search for.
     * @return The index of the book if found, -1 otherwise.
     */
    private static int binarySearch(ArrayList<Book> books, int year) {
        int left = 0, right = books.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (books.get(mid).year == year) {
                return mid; // Book found
            }
            if (books.get(mid).year < year) {
                left = mid + 1; // Search in the right half
            } else {
                right = mid - 1; // Search in the left half
            }
        }
        return -1; // Book not found
    }
}
```

Output:

```
To Kill a Mockingbird (1960)
1984 (1949)
The Great Gatsby (1925)
One Hundred Years of Solitude (1967)
The Catcher in the Rye (1951)
Brave New World (1932)
The Hobbit (1937)
The Lord of the Rings (1954)
Pride and Prejudice (1813)
Animal Farm (1945)
```

```
Sorted list by year:
Pride and Prejudice (1813)
The Great Gatsby (1925)
Brave New World (1932)
The Hobbit (1937)
Animal Farm (1945)
1984 (1949)
The Catcher in the Rye (1951)
The Lord of the Rings (1954)
To Kill a Mockingbird (1960)
One Hundred Years of Solitude (1967)

Enter a year to search for: 1951
Book found: The Catcher in the Rye (1951)
```

I foresee using both algorithms in my professional life: utilizing merge sort to sort large sets of data due to its complexity of O(n log(n)), which means it performs well on large datasets, and utilizing binary search as a building block for algorithms used in machine learning, such as those for training neural networks or finding the optimal hyperparameters for a model.

-Alex

I may have overdone this post, but 'oh well', I found the topic very interesting.

References:

GeeksforGeeks. (2020a, November 18). *Merge sort*. GeeksforGeeks https://www.geeksforgeeks.org/merge-sort/

GeeksforGeeks. (2020b, February 3). *Binary search*. GeeksforGeeks. https://www.geeksforgeeks.org/binary-search/

Lysecky, R., & Lizarraga, A. (2022). *Programming in Java with ZyLabs*[Table]. Zyante, Inc.