Data Structures and Algorithms

Project 1

Data Structures Application (To-Do / Kanban System)

Weight: 25 % | **Language:** C# | **Focus:** Implementation and application of fundamental data structures

Purpose

This project develops a practical understanding of data representation, manipulation, and persistence by implementing a small-scale productivity app (To-Do / Kanban system) **without using built-in collection types**.
Students will practice building **arrays, linked lists, hash maps, and trees** from scratch and use them to manage and persist structured data.

Project Specification

Core Features

1. **To-Do List (Base Functionality)**
    - Add, update, delete, and toggle tasks (completed task / incomplete task).
    - Filter or list tasks by **priority**, **status**, or **creation date**.
    - Persist data locally (JSON / XML).
2. **Kanban View (Enhancement)**
    - Tasks grouped into columns such as *To Do / In Progress / Done*.
    - Move tasks across stages to simulate real workflow.
3. **Task Assignment (Challenge)**
    - Assign tasks to team members.
    - Enforce per-user modification rights.
4. **Task Dependencies (Challenge)**
    - Allow tasks to depend on others; a task unlocks only when prerequisites are completed.
    - Useful to model workflow constraints.

Technical Requirements

- **Important:** You are **NOT** allowed to use any existing collection classes (e.g., those from System.Collections.Generic) or any built-in library functions for finding, searching, or sorting. You must build your own collection classes to store and process tasks. **No built-in generic collections (List<T>, Dictionary<T>, Linq, etc.).**
- Follow **Clean Architecture**:
    - **Model** → defines entities
    - **Repository** → handles persistence
    - **Service** → contains business logic
    - **View / Program** → user interface

- Expand the **abstract data type** for tasks, including representations for **task priority, task status, task assignments to individuals, and task dependencies** on other tasks.
- Define **interfaces** for storing and manipulating tasks, individuals, etc., and provide multiple implementations of these interfaces using the following data structures:
    1. **Array**
    2. **Linked List**
    3. **Hash Map**
    4. **Binary Search Tree**
       (A sample generic list interface is provided below, and can be expanded / extended.)

```
interface IMyCollection<T> {
    void Add(T item);
    void Remove(T item);
    T FindBy<K>(K key, Func<T, K, bool> comparer);
    IMyCollection<T> Filter(Func<T, bool> predicate);
    void Sort(Comparison<T> comparison);
    int Count { get; }
    bool Dirty {get; set;}
    R Reduce<R>(Func<R, T, R> accumulator);
   // OR
    R Reduce<R>(R initial, Func<R, T, R> accumulator);
    IMyIterator<T> GetIterator(); // Custom Iterator - Since we
are not using System.Collections.Generic
    IEnumerator<T> GetEnumerator() // Extra foreach lookup.
}
interface IMyIterator<T> {
```

```
    bool HasNext();   // Checks if there is another element
    T Next();         // Returns the next element
    void Reset();     // Resets the iterator to the beginning
}
```

- Provide a **user interface** that allows users to interact with the to-do list and perform all of the listed functionalities.
- Write test cases to validate the basic functional requirements.
- Implement **persistence** by saving and loading tasks from a file or database, or by making the generic data structures **serializable**.
- Bonus: Enhance the UI to be **more interactive and comprehensive** by presenting the to-do list as a **Kanban board** and supporting **task dependency visualization as a graph**.

Deliverables

- Source code in C# (GitHub / GitLab).
- Short README explaining architecture and chosen data structures.
- Console or minimal GUI interface demonstrating all required features.
- Working Demo and Viva

Suggested Learning Path

1. Start with a minimal console To-Do app using arrays.
2. Replace built-in lists with custom linked lists.
3. Introduce hashing for faster lookup.
4. Extend to tree-based sorting or Kanban visualization.

Project Walkthrough: To begin with start with toy example without any constraints.

Create an interactive console application that allows a user to manage very basic tasks. The user can:

- Add tasks
- Remove tasks
- Update task status (toggle completion)
- List tasks

Additionally, the application can persist data by saving the list of tasks to a JSON file and loading them back when the app starts. We provide a basic implementation walkthrough of a To-Do List application. After completing this walkthrough, a student will be able to expand the project based on the listed requirements.

Architecture: We will follow a Clean Architecture approach by separating the application into distinct layers:

- **Model:** Contains the definition of a task.
- **Repository:** Handles persistence (saving/loading tasks as JSON).
- **Service:** Contains business logic for managing tasks.
- **View:** Presents the user interface and interacts with the service.
- **Program:** Sets up dependency injection and runs the application.

Setting Up the Console Project

1. Create a New Console Application: Open Visual Studio (or VS Code) and create a new C# Console App.

2. Define the Minimal Task Model: Create a new file for the model. This defines the data for each task.

```
class TaskItem {
    public int Id { get; set; }
    public required string Description { get; set; }
    public bool Completed { get; set; }
}
```

3. Repository Interface: Create an interface that declares methods for loading and saving tasks.

```
interface ITaskRepository {
    List<TaskItem> LoadTasks();
```

```
    void SaveTasks(List<TaskItem> tasks);
}
```

- Implement the Repository: with JSON Persistence so that tasks are stored in a JSON file.

```
class JsonTaskRepository : ITaskRepository {
  private readonly string _filePath;
  public JsonTaskRepository(string filePath) => _filePath =
filePath;

  public List<TaskItem> LoadTasks() {
    if (!File.Exists(_filePath)) {
      return new List<TaskItem>();
    }
    string json = File.ReadAllText(_filePath);
    var tasks = JsonSerializer.Deserialize<List<TaskItem>>(json);
    return tasks ?? new List<TaskItem>();
  }
  public void SaveTasks(List<TaskItem> tasks) {
    string json = JsonSerializer.Serialize(tasks, new
JsonSerializerOptions { WriteIndented = true });
    File.WriteAllText(_filePath, json);
  }
}
```

4. Business Logic for Task Management: Create an interface for the service that manages tasks.

```
interface ITaskService {
  IEnumerable<TaskItem> GetAllTasks();
  void AddTask(string description);
  void RemoveTask(int id);
  void ToggleTaskCompletion(int id);
}
```

- Implement the Service: The service loads tasks from the repository, applies business logic, and then saves changes.

```
class TaskService : ITaskService {
  private readonly ITaskRepository _repository;
  private readonly List<TaskItem> _tasks;
  public TaskService(ITaskRepository repository) {
    _repository = repository;
    _tasks = _repository.LoadTasks();
  }
  public IEnumerable<TaskItem> GetAllTasks() => _tasks;

  public void AddTask(string description) {
    int newId = _tasks.Count > 0 ? _tasks[_tasks.Count - 1].Id + 1 :
1;
    var newTask = new TaskItem { Id = newId, Description =
description, Completed = false };
    _tasks.Add(newTask);
    _repository.SaveTasks(_tasks);
  }
  public void RemoveTask(int id) {
    var task = _tasks.Find(t => t.Id == id);
    if (task != null) {
      _tasks.Remove(task);
      _repository.SaveTasks(_tasks);
    }
  }
  public void ToggleTaskCompletion(int id) {
    var task = _tasks.Find(t => t.Id == id);
    if (task != null) {
       task.Completed = !task.Completed;
       _repository.SaveTasks(_tasks);
     }
  }
}
```

5. Define the View Interface

```
interface ITaskView{
   void Run();
}
```

- Implement the Console View: The console view interacts with the user: displaying tasks and processing user input.

```
public class ConsoleTaskView : ITaskView {
  private readonly ITaskService _service;

  public ConsoleTaskView(ITaskService service) {
    _service = service;
  }

  void DisplayTasks(IEnumerable<TaskItem> tasks) {
    Console.Clear();
    Console.WriteLine("==== ToDo List ====");
    foreach (var task in tasks)
      Console.WriteLine($"{task}");
  }

  string Prompt(string prompt) {
    Console.Write(prompt);
    return Console.ReadLine();
  }

  public void Run() {
    while (true) {
      DisplayTasks(_service.GetAllTasks());
      Console.WriteLine("\nOptions:");
      Console.WriteLine("1. Add Task");
      Console.WriteLine("2. Remove Task");
      Console.WriteLine("3. Toggle Task State");
      Console.WriteLine("4. Exit");

      string option=Prompt("Select an option: ");
      switch (option) {
        case "1":
         string description = Prompt ("Enter task description: ");
          _service.AddTask(description);
          break;
        case "2":
        string removeIdStr = Prompt ("Enter task id to remove: ");
```

```
          if (int.TryParse(removeIdStr, out int removeId)) {
            _service.RemoveTask(removeId);
          }
          break;
        case "3":
        string toggleIdStr = Prompt ("Enter task id to toggle: ");
          if (int.TryParse(toggleIdStr, out int toggleId)) {
            _service.ToggleTaskCompletion(toggleId);
          }
          break;
        case "4":
          return;
        default:
          Console.WriteLine("Invalid option. Press any key to
continue...");
          Console.ReadKey();
          break;
      }
    }
  }
}
```

6. Program Entry Point: Wiring Up with Dependency Injection

Finally, set up the application by wiring together the repository, service, and view.

```
class Program {
   static void Main(string[] args){
    // Dependency injection: wiring up our components
    string filePath = "tasks.json";
    ITaskRepository repository = new JsonTaskRepository(filePath);
    ITaskService service = new TaskService(repository);
    ITaskView view = new ConsoleTaskView(service);

    // Run the view
    view.Run();
  }
}
```

7. Running the Application

Build and run your project.

The console will display the current tasks (initially empty) and the menu options.

Add tasks, **remove tasks**, or **toggle their completion**.

The tasks are automatically **saved to tasks.json** and loaded when the application restarts.

Summary

Minimalistic Project: An interactive To-Do List with add, remove, update, save, and load functionalities.

Clean Architecture: Separates the project into Model, Repository, Service, and View.

Dependency Injection: Ensures that each layer depends on abstractions (interfaces), making it flexible and testable.

JSON Persistence: Uses a JSON file (tasks.json) to persist tasks between sessions.

Sample Sprints

Sprint	**Feature**	**Data Structure**	**View Component**
1	Basic Task Management	Generic Array	Basic To-Do List View
2	Task Assignment	Generic Linked List	Enhancements for assignment display
3	Task Dependency	Generic BST	Kanban Board (with dependency highlights)
4	Task Dependency (advanced validations)	Generic Hash Map	Task Dependency Visualization (graph view)