

IKT203: Final Portfolio Assignment

(Algorithms & Data Structures)



Welcome to the Capstone!

This 3-week portfolio is the final capstone for IKT203 and counts for **50%** of your **total course grade**.

The goal isn't to start from scratch. The goal is to show how you can apply everything you've learned. You will be combining skills you've already practiced in your 5 submissions (like Linked Lists, Stacks, Queues, and Sorting) with new, more advanced topics (like Trees, Heaps, and Graphs).

This is a group assignment (1-4 members). You have three weeks of dedicated time without lectures to complete it. Good luck!



Grading, Points, and Group Rules

To make grading fair and transparent, we are using a point-based system. Please read these rules carefully.

1. The Point System

The portfolio is made of 4 categories. In each category, you can choose one of two options:

- A **Standard Option** (worth up to **200 points**)
- An **Advanced Option** (worth up to **300 points**)

This means the "Standard Path" (all 4 standard options) is worth a maximum of **800 points**. A group that takes on advanced challenges can earn **1000 points or more**.

2. Group Size & Grade Thresholds

To ensure the workload is fair for all group sizes, we use **two different point tables** for the final grade.

Grade	Groups of 1-2 (Baseline: 800 pts)	Groups of 3-4 (Baseline: 1000 pts)
A (89-100%)	712+ points	890+ points
B (77-88%)	616 - 711 points	770 - 889 points
C (65-76%)	520 - 615 points	650 - 769 points

D (53-64%)	424 - 519 points	530 - 649 points
E (41-52%)	328 - 423 points	410 - 529 points
F (0-40%)	0 - 327 points	0 - 409 points

What this means:

- **Groups of 1-2** can achieve an 'A' by perfectly completing the "Standard Path" (800 points > 712).
- **Groups of 3-4** *cannot* achieve an 'A' with the "Standard Path" (800 points < 890). They **must** complete advanced options to earn enough points for an 'A'. A perfect "Standard Path" (800 points) will earn a 'B'.

3. The Assignment "Menu"

Here are the points available for each option:

Category	Standard Option (200 pts)	Advanced Option (300 pts)	Extra Credit
Cat 1: Lists, Stacks, Queues	Console Text Editor	Console Music Player	
Cat 2: Sorting & Search	Cruise Ship Manifest	Combined Corporate Directory	
Cat 3: Trees	Employee Directory (BST/AVL)	Interpreted Calculator	+50 pts (RBT)
Cat 4: Graphs & Dijkstra's	Data Center Network Monitor	Inter-city Logistics Router	

Project Setup & Submission Requirements

Please read these technical requirements very carefully. **Failure to follow these rules may result in your submission being marked as incomplete or failed (Not Passed).**

1. Project Starter Kit (Mandatory)

All students **must** use the provided GitHub repository as a starting point. This repository contains the helper files (like `ReadGraph.h/.cpp`) and the required folder structure.

- **Link:** <https://github.com/atsond12/IKT203>
- You must copy or download this archive to start your project.
- You are **not** required to use GitHub or Git for your own version control.

2. Folder & CMake Structure (Mandatory)

You must organize your portfolio into a separate folder for each category. Each category's folder **must** contain its own `CMakeLists.txt` file so that it can be built as a standalone project.

Your final folder structure should look like this:

Portfolio/

```
    └── Assignment-01/
        ├── CMakeLists.txt
        ├── main.cpp
        └── ... (your .h and .cpp files for Cat 1)

    └── Assignment-02/
        ├── CMakeLists.txt
        ├── main.cpp
        └── ... (your .h and .cpp files for Cat 2)

    └── Assignment-03/
        ├── CMakeLists.txt
        ├── main.cpp
        └── ... (your .h and .cpp files for Cat 3)

    └── Assignment-04/
        ├── CMakeLists.txt
        ├── main.cpp
        └── ... (your .h and .cpp files for Cat 4)
```

This structure is essential for us and the external examiners to build and test your work.

3. Final Submission (Mandatory)

You must deliver your entire portfolio as a **single ZIP file**.

- **DO** include: All your `.cpp`, `.h`, and `CMakeLists.txt` files.
- **DO NOT** include: Any binaries, build folders, `bin/` or `obj/` folders, `.exe` files, or other compiled files.

4. Memory Management & "Dumb" Pointers (Mandatory)

This is a strict and fundamental rule for the entire portfolio.

- All data objects (e.g., `TSong`, `TEmployee`, `TNode`, `TVertex`) **must** be created on the heap and stored in your data structures as **raw pointers** (e.g., `DoublyLinkedList<TSong*>` or `BSTNode<TEmployee*>`).
- You are **not permitted** to use smart pointers (`std::unique_ptr`, `std::shared_ptr`).
- A major part of this assignment is to demonstrate correct manual memory management. You must identify which class or list is the "owner" of the data.
- The "owning" class's destructor **must** be implemented to iterate and `delete` all allocated memory to prevent any memory leaks. Failure to manage memory correctly will result in a significant point deduction.

Assignment categories

1 Category 1: Lists, Stacks, & Queues (Application)

This category tests your ability to *apply* the data structures you built in previous submissions to solve a practical problem.

A Friendly Reminder: You are strongly encouraged to re-use and adapt your code from **Submission 1 (Linked List)** and **Submission 3 (Stacks & Queues)**. The main challenge here is not re-implementing them, but integrating them to build a working system.

Memory & Pointers: Remember, all data must be stored as **raw pointers** (e.g., `TSong*`, `TEmployee*`). You are responsible for all `new` and `delete` operations.

You must choose **one** of the following two options:

Option 1 (Standard): Console Text Editor

- **Max Points:** 200
- **Objective:** Simulate a simple line-based text editor with undo/redo functionality and a print queue.

Core Requirements:

1. Text Document (Doubly Linked List):
 - Use a **Doubly Linked List** to store the lines of text in your document. Each node should hold a **string**.
 - This list allows you to add new lines, delete lines, and display the full document by traversing forward or backward.
2. Undo/Redo (Two Stacks):
 - Implement an **Undo Stack** and a **Redo Stack**. These stacks will store "action" objects (e.g., a struct **TAction** that remembers if the action was "INSERT" or "DELETE" and what text was involved).
 - **Undo:** When the user performs an "undo":
 - Pop the action from the **Undo Stack**.
 - Perform the *inverse* action (e.g., if the action was "INSERT", you will "DELETE" that line).
 - Push the original action onto the **Redo Stack**.
 - **Redo:** When the user performs a "redo":
 - Pop the action from the **Redo Stack**.
 - Re-perform the original action (e.g., "INSERT" the line again).
 - Push that action back onto the **Undo Stack**.
 - **New Action:** When the user performs a *new* action (like typing a new line), the **Redo Stack must be cleared**.
3. Print Jobs (Queue):
 - Implement a Print Queue.
 - Create a "Print" command that adds the entire document (e.g., as a single job) to the end of the queue.
 - Create a "Process Print Job" command that dequeues the next job and "prints" it to the console, showing the FIFO (First-In, First-Out) principle.

Option 2 (Advanced): Console Music Player

- **Max Points:** 300
- **Objective:** Simulate a music player with a song library, two different playback queues, and a listening history.

Core Requirements:

1. Song Library (Doubly Linked List):
 - Use a **Doubly Linked List** to store your "master library" of all `TSong` objects.
 - This list allows the user to navigate **forward and backward** through the entire library to view all available songs.
2. Playback Queues (Two Queues):
 - Implement a **Main Queue** and a **Wish Queue**.
 - The user must be able to add any song from the library to either queue.
3. Playback Logic (The Core Task):
 - Create a "Play Next Song" function. This function *must* follow this rule:
 - It **must** first check the **Wish Queue**. If the Wish Queue is not empty, it must dequeue and "play" (print to console) songs from it until it is empty.
 - **Only** when the Wish Queue is empty should it dequeue and "play" the next song from the **Main Queue**.
4. Playback History (Stack):
 - Implement a **History Stack**.
 - Every song that is "played" (from either queue) must be pushed onto the **History Stack**.
 - The user must be able to "View History" (showing the LIFO, Last-In, First-Out, order).
 - Implement a "Play Previous Song" function that **pops** the last-played song from the history stack and adds it to the *front* of the **Wish Queue** (so it plays immediately).

2 Category 2: Sorting & Searching (Application)

This category tests your ability to apply and *adapt* the sorting and searching algorithms you built in **Submission 5**.

 **A Friendly Reminder:** You are expected to re-use your code for **Quick Sort**, **Merge Sort**, and **Binary Search**. The main challenge is adapting them to sort complex objects (structs/classes) and using them in a data-processing pipeline. You will also need to use the provided `readNamesFromFile` function.

 **Memory & Pointers:** Remember, all data must be stored as **raw pointers** (e.g., `TSong*`, `TEmployee*`). You are responsible for all `new` and `delete` operations.

You must choose **one** of the following two options:

Option 1 (Standard): Cruise Ship Manifest

- **Max Points:** 200
- **Objective:** Organize the passenger and crew manifests for a new cruise ship. You must load all personnel, sort them into "Guest" and "Employee" lists, and then group guests for cabin assignments.

Core Requirements:

1. Data Loading (using `readNamesFromFile`):
 - Define a `TPerson` struct. It must contain `firstName`, `lastName`, an `enum status` (e.g., `GUEST` or `EMPLOYEE`), and an `int cabinSize` (randomly 1, 2, 3, or 4).
 - In your `onNameRead` callback, create a `TPerson` object for each name.
 - Assign a `status` (e.g., first 1500 are `EMPLOYEE`, the rest are `GUEST`).
 - Add each person to one of **two separate linked lists**: a `guestList` or an `employeeList`.
2. Task 1 (Merge Sort): Alphabetical Manifests
 - Implement **Merge Sort** for linked lists (you can adapt your array-based one).
 - Sort *both* the `guestList` and the `employeeList` alphabetically by **1. Last Name, then 2. First Name**.
 - (You will need to overload `operator<` or add a callback for your `TPerson` structure compare).
3. Task 2 (Quick Sort): Cabin Grouping
 - **Convert** your *sorted* `guestList` (from Task 1) into a new `array` of data pointers, remember to hold correct data ownership.
 - Implement **Quick Sort** for arrays.
 - **Re-sort** this new `guestArray` using a *different* key: **1. cabinSize (ascending), then 2. Last Name**. This creates an array optimized for filling cabins.
4. Task 3 (Binary Search): Passenger Lookup
 - Implement **Binary Search** for arrays.
 - The purser needs to find guests by name, not cabin. You must perform this search on an **alphabetically-sorted array** of guests.
 - You can either:
 - A) Perform the binary search on the `guestArray` (from Task 2) *before* you re-sort it by cabin size.
 - B) Create a second, separate array from your sorted `guestList` (from Task 1) that remains alphabetical, just for searching.

Option 2 (Advanced): Combined Corporate Directory

- **Max Points:** 300
- **Objective:** Generate two different, sorted reports from a master employee database. This option is "Advanced" because it requires complex, multi-key sorting.

Core Requirements:

1. Data Loading (using `readNamesFromFile`):
 - Define an `TEmployee` structure. It must contain `firstName`, `lastName`, and a randomly assigned `enum EDepartment` (e.g., "IT", "Finance", "HR").
 - In your `onNameRead` callback, create an `TEmployee` object for each name.
 - Add each new employee to **two** separate data structures:
 - A **master linked list**.
 - A **master array (data pointers)**.
2. Task 1 (Merge Sort): Master Roll Call
 - Implement **Merge Sort** for your linked list.
 - Sort the **master linked list** alphabetically by **1. Last Name**, then **2. First Name**.
 - (This will require you to overload `operator<` or pass a comparison function).
3. Task 2 (Quick Sort): Organizational Chart (Multi-key Sort)
 - Implement **Quick Sort** for your array.
 - Sort the **master array** *in-place* using a different, more complex multi-key:
 - **1. department (alphabetical)**
 - **2. lastName (alphabetical)**
 - **3. firstName (alphabetical)**
 - (This will likely require a more advanced comparison function/lambda for your sort).
4. Task 3 (Binary Search): Employee Lookup
 - **Convert** the alphabetically-sorted **linked list** (from Task 1) into a **new, separate array**.
 - Implement **Binary Search** for arrays.
 - Prompt the user to enter a last name and first name.
 - Perform the binary search on this **new, alphabetically-sorted array** to find an employee.

3 Category 3: Trees (BST, AVL & RBT)

This category is a major implementation challenge. Unlike the previous categories, you will be building these complex data structures from scratch.

 **A Note on this Task:** This is **not** an application task. You are implementing the core logic for Binary Search Trees and self-balancing AVL trees. You may, however, use the `readNamesFromFile` function to get data.

 **Memory & Pointers:** Remember, all data must be stored as **raw pointers**. You are responsible for all `new` and `delete` operations.

You must choose **one** of the following two options:

Option 1 (Standard): Employee Directory (BST vs. AVL)

- **Max Points:** 200
- **Objective:** Implement and compare a standard Binary Search Tree (BST) with a self-balancing AVL Tree for managing an employee directory.

Core Requirements:

1. **Data:** Use the first 200 names from your `readNamesFromFile` function.
2. **TEmployee Struct:** Create a `TEmployee` struct. In addition to `firstName` and `lastName`, add a unique, randomly generated integer `employeeID`. This **ID will be the key** for your trees.
3. **BST Implementation:** Implement a standard **Binary Search Tree (BST)** class. It must include `insert(key, data)`, `delete(key)`, and `search(key)` functions.
4. **AVL Tree Implementation:** Implement a self-balancing `AVLTree` class. This requires:
 - `insert(key, data)` and `search(key)` functions.
 - `getHeight` and `getBalanceFactor` helper functions.
 - The four rotation functions: **Left-Left (LL)**, **Right-Right (RR)**, **Left-Right (LR)**, and **Right-Left (RL)**.
 - Your `insert` function must correctly call these rotations to rebalance the tree.
 - **Note:** A functional `delete` for the AVL tree is **NOT** required for the 200 points. See Extra Credit.
5. **Traversals:** Implement all four common traversals (Inorder, Preorder, Postorder, Level-order) for both trees.

Option 2 (Advanced): Interpreted Calculator

- **Max Points:** 300

- **Objective:** Build a calculator that parses an infix expression string (like `(10 + x) * 3`) and evaluates it. This option is "Advanced" because it requires **two** different tree structures.

Core Requirements:

1. The Symbol Table (BST or AVL):
 - Implement a **BST** or **AVL Tree** (with `insert` and `search` only) to store variables.
 - The **key** will be the variable `name` (a `string`) and the **value** will be its `double` value.
 - Your program must allow users to define variables (e.t., `set x = 10`). This calls your tree's `insert("x", 10.0)` function.
2. The Parser & Expression Tree:
 - You must parse an infix string from the user (e.g., `calc (x + 5) * 3^2 - (14 / 4) * y`).
 - Your parser must handle: numbers, operators (`+`, `-`, `*`, `/`, `^`), parentheses `()`, and functions (`ln`, `log`, `exp`).
 - When the parser sees a **variable** (like `x` or `y`), it must call your Symbol Table's `search("x")` function to get its value.
 - The parser must build a **Binary Expression Tree**. Internal nodes will be operators/functions, and leaf nodes will be numbers (or the *values* of variables).
3. Evaluation:
 - Implement a **postorder traversal** of your Expression Tree. This traversal will naturally evaluate the expression and return the final `double` result.

Extra Credit (Category 3) ★

- **Max Points:** Up to +100
- **Tasks (Can be combined):**
 - **AVL Delete (+50 pts):** Implement a fully functional and balancing `delete` operation for your `AVLTree`.
 - **Red-Black Tree (+50 pts):** Implement a functional `RedBlackTree` (with `insert` and `search`). This can be used in either Option 1 or 2.

4 Category 4: Graphs & Dijkstra's Algorithm

This is the final implementation challenge. You will build a complete, weighted graph structure from a data file and then implement Dijkstra's famous shortest path algorithm.

 **A Note on this Task:** This category requires you to implement:

1. An **Adjacency List** structure to represent a weighted graph.
2. A **Min-Heap (Priority Queue)**, which is essential for an efficient Dijkstra's implementation.
3. Callback functions to parse the provided data files.

 **Memory & Pointers:** Remember, all data must be stored as **raw pointers**. You are responsible for all **new** and **delete** operations.

Provided Files: You will be given a helper utility (`ReadGraph.h` and `ReadGraph.cpp`) and two data files: `network_graph.txt` and `city_graph.txt`.

You must choose **one** of the following two options:

Option 1 (Standard): Data Center Network Monitor

- **Max Points:** 200
- **Objective:** Find the lowest-latency path for data to travel between two specific servers in a network.
- **Data File:** `network_graph.txt` (This is an **undirected** graph).

Core Requirements:

1. Graph Implementation (Data Parsing):
 - Use the provided `readGraphFromFile` function to parse `network_graph.txt`.
 - You must implement the two callback functions (`FNodeRead` and `FEdgeRead`).
 - Your callbacks must correctly populate your **Adjacency List** (or other graph structure) in memory. Remember, because this graph is *undirected*, an edge from "A to B" also means you must add an edge from "B to A".
2. Min-Heap (Priority Queue):
 - Implement a **Min-Heap** data structure. It must store "vertices to visit" and be prioritized by the *lowest known latency* from the source.
3. Dijkstra's Algorithm:
 - Implement Dijkstra's algorithm. Your implementation **must use your Min-Heap** to efficiently select the next vertex to process.
4. Application:
 - The user must be able to select a "**Source Server**" and a "**Destination Server**" from the nodes you loaded.
 - Your program must find and display the *single shortest path* and the *total latency* between those two nodes.

- **Example Output:** Lowest latency path: WebServer -> Router1 -> Database (Total: 7ms)

Option 2 (Advanced): Inter-city Logistics Router

- **Max Points:** 300
- **Objective:** Simulate a logistics GPS, finding the *cheapest cost* from one city to **all other reachable cities**.
- **Data File:** `city_graph.txt` (This is a **directed** graph).

Core Requirements:

1. Graph Implementation (Data Parsing):
 - Use the provided `readGraphFromFile` function to parse `city_graph.txt`.
 - You must implement the two callback functions (`FNodeRead` and `FEdgeRead`).
 - Your callbacks must correctly populate your **Adjacency List**. (Because this graph is *directed*, you only add the edge in the direction specified in the file).
2. Min-Heap (Priority Queue):
 - Implement a **Min-Heap** data structure. It must store "vertices to visit" and be prioritized by the *lowest known cost* from the source.
3. Dijkstra's Algorithm:
 - Implement Dijkstra's algorithm. Your implementation **must use your Min-Heap**.
4. Application:
 - This option is "Advanced" because you must find all paths from the source, not just one.
 - The user must be able to select a "**Start City** (**Source**)".
 - Your program must run Dijkstra's to find the shortest path from that source to **every other reachable city**.
 - It must then print a full "Routing Table" showing the cost to each destination.
 - **Example Output:**
 - Shortest cost to Bergen: 450
 - Shortest cost to Trondheim: 820
 - Shortest cost to Stavanger: 310