Data Structures

Assignment 2

General Trees, BST, AVL Trees

Deadline: 11:55 pm on Sunday, March 3, 2024,

Lead TAs: Muhammad Emad Sarwar, Saad Momin Mehmood, Fayzan Ali Akhtar

Plagiarism Policy:

1. Students must not share the actual program code with other students.
2. Students must be prepared to explain any program code they submit.
3. Students cannot copy code from the Internet.
4. Students must indicate any assistance they received.
5. All submissions are subject to automated plagiarism detection.
6. Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Learning Outcomes

In this assignment, you will

- Practice creating general tree and their various functions
- Practice creating BST and AVL Trees and their various functions
- See how these data structures can be used in real-life scenarios while preparing for a megaevent

Pre-requisites

For this assignment, you will require

- An understanding of arrays, vectors, smart pointers, and trees
- Familiarity with C++ syntax
- And knowledge of object-oriented programming

Linux Environment

Your code must be compiled in the Linux environment using either the Windows Subsystem for Linux or a Linux Distribution via Dual Boot.

You are good to go if you are already running Linux or MacOS, but for those who are on Windows. You can use one of the two options below; **the safest one is to go with WSL as we suggest**, Dual Boot, only if you have an idea of what you are doing.

Option 1: WSL

Please follow these steps:

1. Press the Windows button and type Powershell
2. Run the Powershell as an **Administrator**.

3. Type the following command prompt in the PowerShell

```
wsl –install -d Ubuntu
```

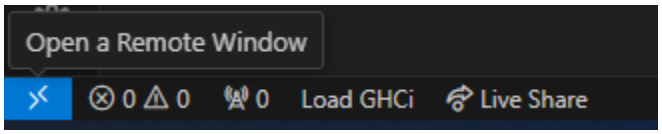
4. Once WSL is installed, please reboot your system.
5. Once your system has rebooted, press Windows + R to launch the run window.
6. Type cmd and press Enter
7. Once cmd has launched, type in WSL or bash to boot up the Linux VM installed through WSL and type the following command.

```
sudo apt-get update && sudo apt-get upgrade -y && sudo apt install build-essential
```

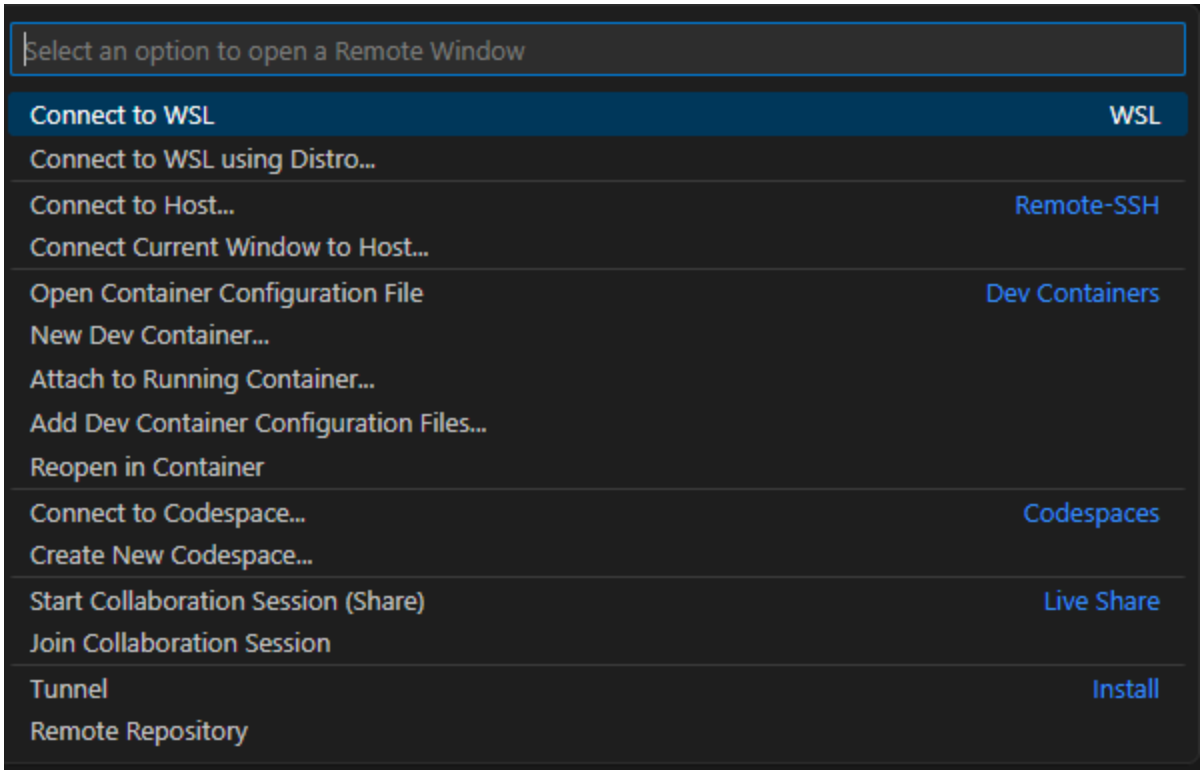
8. Inputting the command above will take some time and once it does, you'll have it installed.

Connecting to WSL:
- You can either open a controlled environment of the subsystem in VSCode or simply integrate it within your terminal

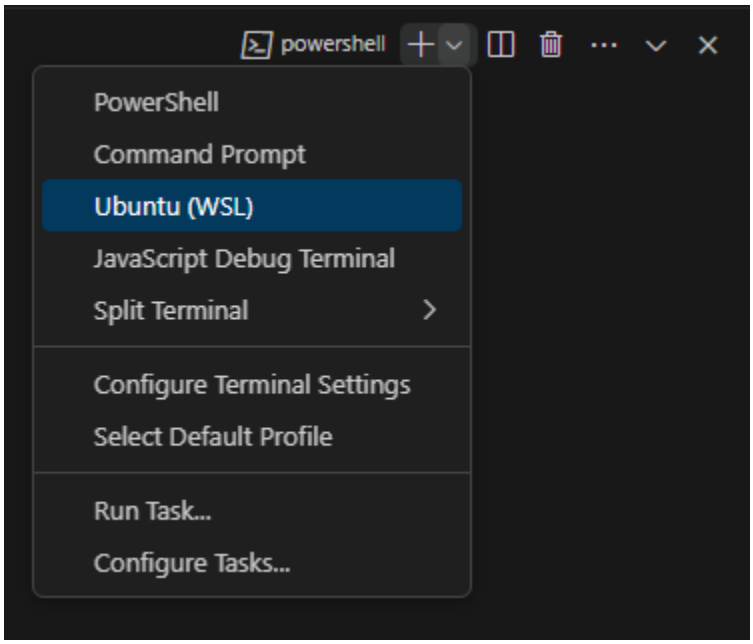
-Case 1: Controlled Environment



-> Open the remote window (little blue box) at the bottom left, after which you'll get the following pop-up



Case 2:- Integration within the terminal



At the top right side of the terminal, you'll see the option to open a WSL terminal within.

Option 2: Dual Boot

Please follow these steps :

1. Download Ubuntu from the official Ubuntu Website or any other distribution of your liking.
2. Download Rufus (a tool for creating a bootable USB)
3. Launch Rufus and select your Linux Image and burn it onto a USB
4. Restart your laptop and enter your BIOS
5. Select the bootable USB and the Ubuntu Image will launch
6. Click on “Install Ubuntu” and follow the steps
7. When prompted, select the “Install alongside Windows Boot Manager” and use the slider in the next step to allocate space for your installation (32 GB at least).
8. Simply follow the steps to complete the installation
9. Enter the following commands once the installation is completed

```
sudo apt-get update && sudo apt-get upgrade -y && sudo apt install build-essential
```

Start early as the assignment will take **time** and **effort**

Submission guidelines:

Zip the complete folder and use the following naming convention:

PA2_<rollnumber>.zip

For example, if your roll number is 25100045, then your zip file name should be:

PA2_25100045.zip

All submissions must be uploaded on LMS before the deadline.

You are allowed 5 "free" late days during the semester (that can be applied to one or more assignments; the final assignment will be due tentatively on the final day of classes, i.e., before the deadline week, and cannot be turned in late. The last day to make any late submission is also the final day of classes, even if you have free late days remaining).

If you submit your work late for any assignment once your 5 “free” late days are used, the following penalty will be applied:

- 10% for work submitted up to 24 hours late
- 20% for work submitted up to 2 days late
- 30% for work submitted up to 3 days late
- 100% for work submitted after 3 days (i.e., you cannot submit assignments more than 3 days late after you have used your 5 free late days.

PART 1: GENERAL TREES marks = 20

In this part, you will be applying what you learned in class to implement different functionalities of a tree efficiently. The basic layout of a general tree is given to you in the Tree.h file.

The template node in Tree.h represents a node in the tree. The class Tree implements the general tree, which contains a pointer to the root and other function declarations.

NOTE:
A node in this tree can have **any** number of children.

Member functions:

Write implementation for the following methods as described here.

`Tree(shared_ptr<node<T,S>> root)` Simple default constructor.
`shared_ptr<node<T,S>> findKey(T key)` Finds the node with the given key and returns a pointer to that node. NULL is returned if the key doesn't exist.
`shared_ptr<node<T,S>> findKeyHelper(shared_ptr<node<T,S>> currNode, T key)` Helper function to be used in findKey function.
`bool insertChild(shared_ptr<node<T,S>> newNode, T key)` • Inserts the given node as the child of the given key. Returns true if the insertion is successful and false if the key doesn't exist. Insertion should also fail if another node with the same key as the new node already exists. • If the node at the given key already has children, the new node should be added to the end of the children vector.
`vector<shared_ptr<node<T,S>>> getAllChildren(T key)` Returns all the children of the node with the given key. Should return an empty vector in case the node has no child or key doesn't exist.
`int findHeight()` Returns the height of the tree. **NOTE**: A tree with only root has height 0.
`int findHeightHelper(shared_ptr<node<T,S>> currNode)` Helper function to be used in the findHeight function.
`void deleteTree()` Deletes the entire tree.
`bool deleteLeaf(T key)` Delete node with a given key if and only if it is a leaf node. It doesn't delete the root node, even if it is the only node in the tree. Returns true on success, false on failure.

```
shared_ptr<node<T,S>> deleteLeafHelper(shared_ptr<node<T,S>> currNode, T
key)
```

Helper function to delete the leaf node.

NO CREATION OF ADDITIONAL HELPER FUNCTION IS ALLOWED.

PART 2: DOMAIN NAME SYSTEM marks = 25

The internet is in its nascent stages, rapidly evolving into a global network. With an increasing number of domains being registered, there's a pressing need to organize and manage these domains efficiently. You have been hired to design a hierarchical system for the Domain Name System (DNS), a critical component of the Internet's infrastructure.

Your task is to develop a DNS[1] hierarchy using the general tree structure implemented in Part 1 of your assignment. The DNS hierarchy will structure and categorize domains and subdomains to reflect their relationships and dependencies.

Structure of the DNS Hierarchy:

1. **Root Node:**

   At the top of the hierarchy is the **ROOT** node, representing the starting point of the DNS (already done for you).

2. **TLD Nodes:**

   The next level contains nodes for Top Level Domains (TLDs) like **.com**, **.org**, **.net**, etc. These nodes are children of the **ROOT** node.

3. **Domain Nodes:**

   Under each **TLD** node, there will be **domain** nodes. For example, under **.com**, there could be **google**, **facebook**, etc.

4. **Subdomain Nodes:**

   Each **domain** node can have multiple **subdomain** nodes. For instance, under **google**, there could be **maps**, **mail**, etc.

5. **Further Levels:**

   If applicable, **subdomains** can have their **child** nodes, representing deeper levels of the URL structure, such as **google.com/drive/create**, where **create** is a child of **drive**.

---

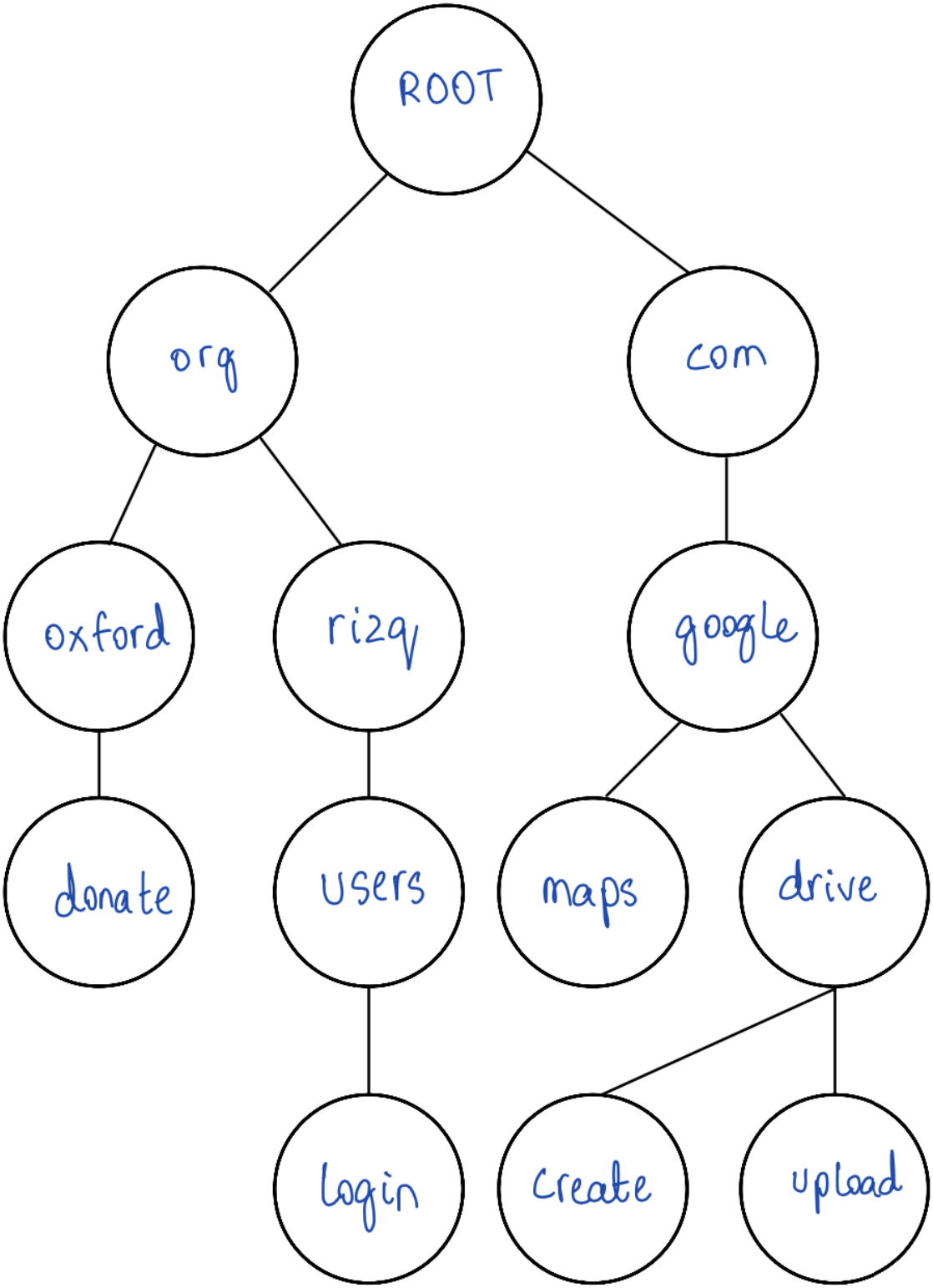
[1] It is a simplified version; actual DNS is more complex

Member functions:

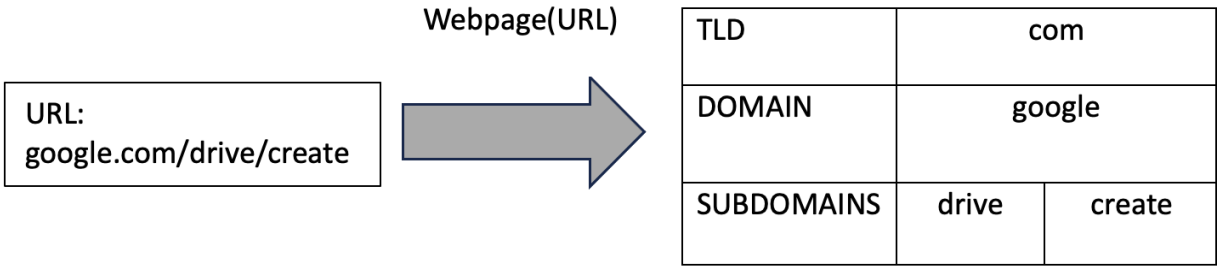
Write implementation for the following methods as described here.

`DNS()` Simple default constructor, which creates the **ROOT** node.
`void addWebpage(string url)` Adds a new webpage to the DNS hierarchy. (Hint: Start from TLD, then domain name, and then subdomain)
`int numRegisteredTLDs()` Returns the count of nodes at the second level.
`vector<shared_ptr<Webpage>> getDomainPages(string domain)` Retrieves all webpages under the specific domain.
`void getDomainPagesHelper(shared_ptr<node<string,string>> currDomain, string currentPath, shared_ptr<vector<string>> results)` Helper function to be used in `getDomainPages` function.
`vector<shared_ptr<Webpage>> getAllWebpages(string TLD)` Retrieves all webpages registered under the specific TLD.
`shared_ptr<Tree<string, string>> getDomainTree()` Returns the pointer to the DNS tree.
`string findTLD(string domain)` Finds the TLD for the given domain. Return an empty string if no such domain had been registered.
`void generateOutput(vector<shared_ptr<Webpage>> pages)` Writes the content of the vector to the text file. **Used for testing purposes**

Here's a small demo of how the methods are supposed to work:



We have created a `struct Webpage` for you which parses the url into **domain**, **subdomains**, and **TLD**. An example:

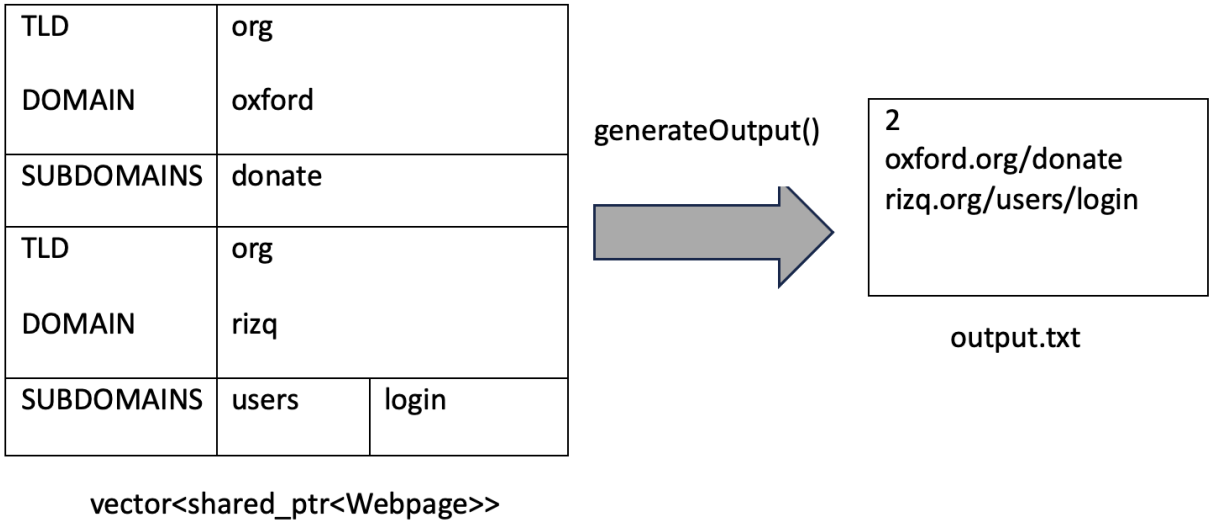


The tree on the previous page was created by the following function calls:

```
addWebpage("oxford.org/donate")
addWebpage("google.com/maps")
addWebpage("google.com/drive/create")
addWebpage("rizq.org/users/login")
addWebpage("google.com/drive/upload")
```

The function call below returns a vector:

1. `getAllWebpages("org")`



2. `getDomainPages("google")`

TLD DOMAIN	com google	
SUBDOMAINS	maps	
TLD DOMAIN	com google	
SUBDOMAINS	drive	create
TLD DOMAIN	com google	
SUBDOMAINS	drive	upload

vector<shared_ptr<Webpage>>

generateOutput()

3
google.com/maps
google.com/drive/create
google.com/drive/upload

output.txt

The rest of the methods are self-explanatory:

1. `findTLD("rizq")` returns "org"
2. `numRegisteredTLDs()` returns 2

NOTE:

- **No** creation of additional helper functions is allowed.
- **Only** methods defined in Part 1 should be used for the implementation.
- **Visualizing** the tree structure on paper with sample domains is recommended to facilitate understanding and implementation.

PART 3: BALANCED SEARCH TREES marks = 30

You have been hired as a recruitment manager at Amazon. Your job is to shortlist and interview clients who have cleared the initial automated resume screening. For the automated screening, Amazon uses an AVL tree data structure to store applicants' resumes. Each node represents a resume and associated data (e.g., name, contact information, skills, experience, education). The tree is ordered by the job seeker's years of work experience. Refer to avl.h, which provides all the required definitions. Write your implementation in avl.cpp using the provided boiler code.

For this part, the following simplified node class is provided to store the resume:

```
template <class T, class S>
class Node {
public:
     S fullName;
     S gender;
     T workExperience;
     shared_ptr<node> left;
     shared_ptr<node> right;
     Int height;

     node(S n, S g, T w) {
          this->fullName = n;
          this->gender = g;
          this->workExperience = w;
          left = NULL;
          right = NULL;
          height = 1;
     }
};
```

NOTE:

- You can assume that `fullName` is a string without spaces and does not contain any invalid characters.
- `gender` is represented as 'M' or 'F'.
- `workExperience` represents the work experience of the applicant in months.

Implement the AVL class to store the simplified resumes of all applicants.

Member functions:

Write implementation for the following methods as described here.

`AVL(bool isAVL)` Simple default constructor to initialize the isAVL flag. If the isAVL flag is set, insertions and deletions will follow the AVL property. Otherwise, it will be a simple BST.
`void insertNode(shared_ptr<node<T,S>> N)` Inserts the given node into the tree such that the AVL (or BST) property holds.
`void deleteNode(T k)` • Deletes the node with the given key such that the AVL (or BST) property holds. • As a convention, while deleting a node with 2 children, the new root should be selected from the **left** subtree.
`shared_ptr<node<T,S>> searchNode(T k)` Returns the pointer to the node with the given key. NULL is returned if the key doesn't exist.
`shared_ptr<node<T, S>> getRoot()` Returns the pointer to the root node of the tree
`int height(shared_ptr<node<T,S>> p)` Returns the height of the tree. **NOTE**: A tree with only a root has a height 1.

TIPS:

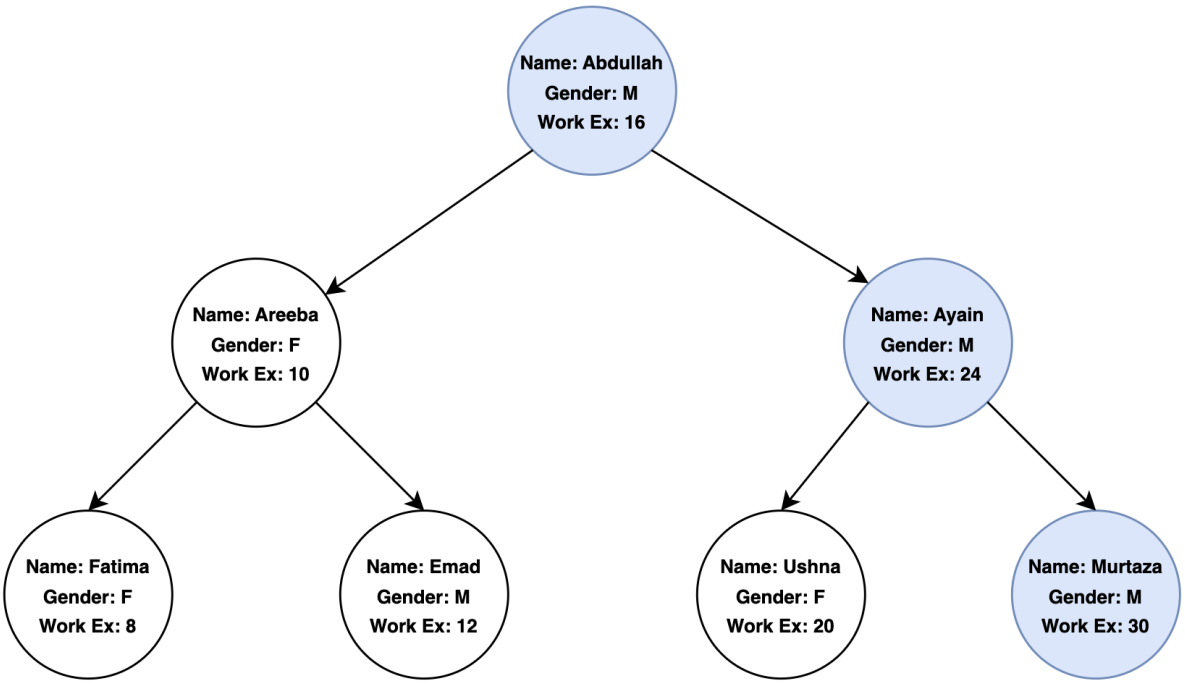
- Start by implementing all operations for a simple BST.
- Make **additional** helper functions for balancing the tree and the left and right rotations.
- Don't forget to update the height of nodes after insertion/deletion.

PART 4: ETHICAL IMPLICATIONS marks = 20

The algorithm you use to shortlist applicants is as follows: Given a tree of resumes, all applicants that fall on the right-most path from the root to the leaf are shortlisted. The applicant at the highest level in the tree is given the highest priority. If some applicant cannot appear for an interview, you pick the left sibling of that applicant from the same level. Note that the number of shortlisted applicants equals the height of the rightmost leaf. You will be implementing the AVL class in ethics.cpp, defined in ethics. h.

The highlighted nodes in the following diagram are the **shortlisted** applicants. The priority of the shortlisted candidates is as follows (a lower integer means a higher priority):

1. Murtaza
2. Ayain
3. Abdullah



Data analysts at Amazon apprise you of a surprising, yet unfortunate, insight. Based on the employee data collected from all Amazon offices, males tend to have a **higher** average work experience (when measured in a number of months) than females. You realize that your shortlisting algorithm is biased against females! This is because job seekers with lesser work experience will

generally occupy **lower-left** subtrees, thereby being less likely to be matched with job opportunities.

TASK 1:

Your next task is to explore how this bias varies if you change your shortlisting algorithm. Your current approach has been stated above. You propose two alternatives:

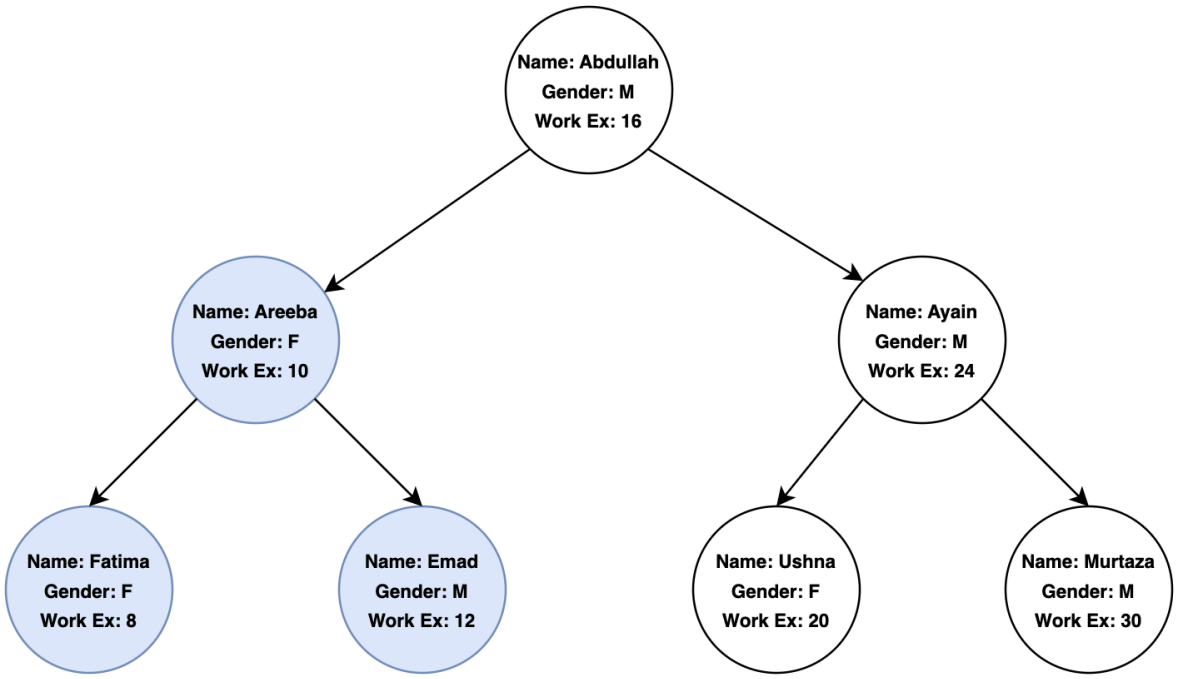
Algorithm 1:

Shortlist applicants according to the in-order traversal of the tree

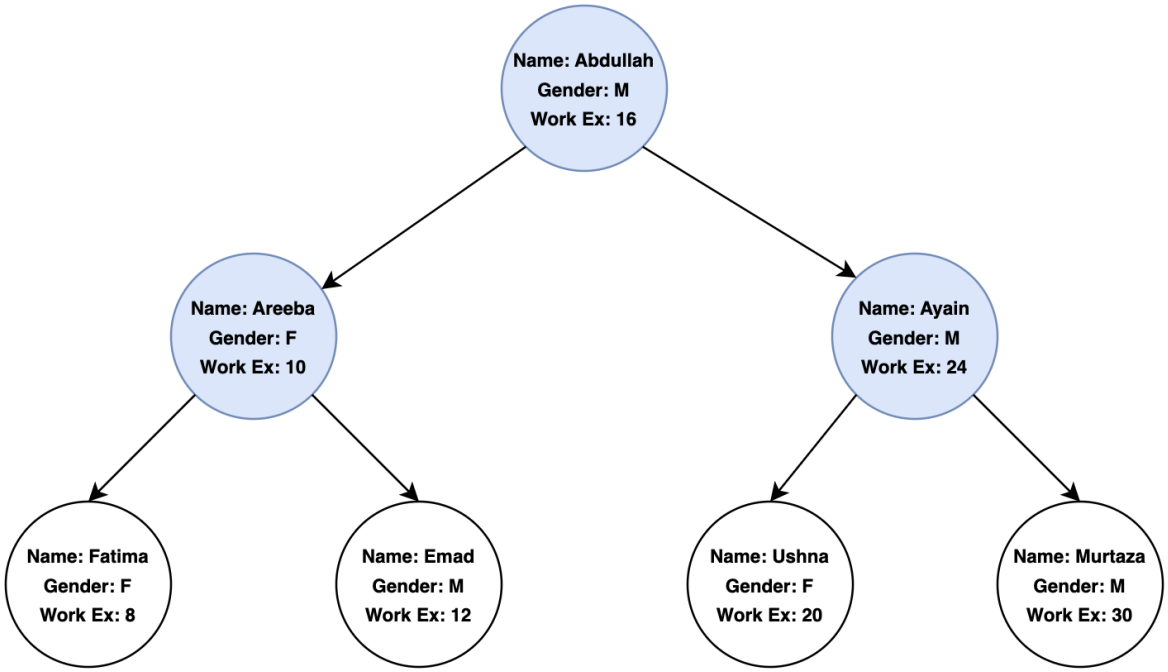
Algorithm 2:

Shortlist applicants according to the level-order traversal of the tree

The shortlisted candidates (here, 3) from the alternate algorithms are illustrated below:



Algorithm 1



Algorithm 2

You decide to write code to compare the two strategies. Implement the following functions:

`int number_to_shortlist(shared_ptr<node<T,S>> root)` Returns the total number of applicants you can shortlist.
`vector<T> right_most(shared_ptr<node<T,S>> root)` Returns an array of keys on the **right-most** path from the root to the leaf.
`vector<T> in_order(shared_ptr<node<T,S>> root)` Returns an array of keys traversed **in order** from the root. Note that you can only shortlist a limited number of applicants; therefore, you need to return the same amount of keys as the number of applicants that can be shortlisted.
`vector<T> level_order(shared_ptr<node<T,S>> root)` Returns a list of keys traversed in **level order** from the root. Note that you can only shortlist a limited number of applicants; therefore, you need to return the same amount of keys as the number of applicants that can be shortlisted.

TASK 2:

You decide to raise your concerns about this bias to Amazon's CEO. However, he isn't convinced by your argument. To help solidify your argument, you decide to prove the existence of the bias (in the original algorithm).

Implement the following function to reveal the bias in the automated resume screening system:

```
vector<T> bias(shared_ptr<node<T,S>> root)
```

- For every node in the tree, compute the ratio of **Females** to **Males** in the **left** subtree of that node.
- The function should return a vector that contains the ratios computed for all nodes, where the order of these ratios is the same as the level order traversal of the tree. (Simply put, the vector of ratios should be ordered the same way as the level-order traversal of the tree).

Additionally, include the answer to the following question as a comment (at the top of the function):

Describe what these ratios mean. Are they significant to prove that there is bias in the tree due to differences in average work experience?

PART 5: GRAPHICAL ANALYSIS marks = 5

This section will focus on the graphical analysis of the performance data collected from the previous tests. The objective is to visually interpret and compare the efficiency of BST and AVL Trees implemented in part 3.

Objectives:

- Generate graphs to compare the insertion performance (in terms of time efficiency) of both BSTs and AVL Trees..
- Visualize how the performance scales with an increasing number of operations.

We've provided you with relevant scripts in the test suite to generate graphs for you in the src/Part 5 directory. You just need to run them and then use the graphical data to conclude the relative performance of the BST and AVL Trees.

Deliverables:

1. Generated graphs
2. A summary detailing the insights gained from the graphical analysis is at this link.

Running and Testing Code:

The comprehensive testing suite developed for the assignment can be easily accessed and executed through a user-friendly interface. The following steps will guide you through running and testing various parts of the assignment:

Step 1: Start the Program

To begin the testing process, execute the Test program.

```
g++ Test.cpp
./a.out
```

Step 2: Interface Navigation

Upon starting the program, you will be presented with a menu interface listing all available test parts and their corresponding indices and allocated marks.

Step 3: Selecting a Test

To run a specific test, enter the index number corresponding to that test when prompted.

Step 4: Test Execution

After selecting a test, the program will automatically compile and execute the respective C++ file for that test. Some tests may take some time; your patience is greatly appreciated.

Step 5: Reviewing Results

Observe the output in the console for the results of the test. This may include success messages, error logs, performance metrics, or graphical output, depending on the specific test.

Step 6: Repeat or Exit

After a test is completed, the menu interface will reappear, allowing you to run additional tests or exit the program.

MARKS DISTRIBUTION

Part	Marks
Part 1: General Tree	**20**
Part 2: Domain Name System	**25**
Part 3: Balanced Search Trees	**30**
Part 4: Ethical Implications	**20**
Part 5: Graphical Analysis	**5**