Group 17: (Kushal Panthi, Chinnabbaichowdary kancharla,Quan Nguyen, Nathan Paul, Seth Perry, Vikaas* (absent)
4/30/2024
Documentation for the Linked list and BST.

# FAKER

Fake Student Data Generation Documentation

This documentation describes a Python script that utilizes the Faker library to generate fake student data and write it to a text file.

**1. Purpose:**

The purpose of the script is to create a large dataset of fake student records with unique identifiers, names, dates of birth, and addresses. This data can be used for testing, simulation, or educational purposes.

**2. Dependencies:**

- Faker: The Faker library is used to generate fake data, such as names, addresses, and dates of birth. It must be installed to run the script (`pip install Faker`).

**3. Classes:**

- Student: Represents a student with attributes including ID, name, date of birth, and address.

- Address: Represents an address with attributes for street, city, state abbreviation, and ZIP code.

**4. Functions:**

- generate_students(num_students): Generates a list of fake student objects with unique IDs, names, dates of birth, and addresses. It utilizes the Faker library to create realistic-looking data.

- write_to_txt(students, filename): Writes the generated student data to a text file in comma-separated values format. Each line in the file represents a student record with fields separated by commas.

**5. Main Functionality:**

- The main() function orchestrates the data generation and file writing process.

- It defines the number of students to generate (`num_students`) and the output file name (`filename`).

- The script checks if the output file already exists. If it does, the script terminates to prevent overwriting existing data.

- Otherwise, it generates fake student data using the generate_students() function and writes it to a text file using the write_to_txt() function.

- Upon successful completion, it prints a message indicating the location of the generated file.

**6. Usage:**

- To use the script, ensure that the Faker library is installed.

- Save the script to a Python file (e.g., generate_student_data.py).

- Run the script using a Python interpreter (`python generate_student_data.py`).

- After execution, the script will create a text file named "students.txt" in the same directory containing the generated student data.

**7. Note:**

- The script uses Faker to generate random data, ensuring that each student record appears realistic.

- It employs unique IDs to prevent duplicates in the generated dataset.

- The generated data is stored in a text file

- Users can customize the number of students generated by modifying the `num_students` variable in the main() function.

- Additional customization of the generated data can be achieved by adjusting Faker's configuration options.

# Student System Management Program Documentation (Linked List version)

## Overview

This program implements a Student System Management system using a singly linked list data structure. It allows users to perform operations such as adding, deleting, searching, and updating student records. The program utilizes object-oriented programming principles with C++.

## Files and Components

**housekeeping.h:** This header file declares a function housekeeping() used for displaying a menu to the user.

**LinkedList.h**: This header file declares the LinkedList class which represents the singly linked list data structure. It contains functions for inserting, searching, deleting, updating, and saving records.

**Node.h:** This header file declares the Node class which represents the individual nodes of the linked list. Each node contains information about a student record.

**Node.cpp:** This file defines the member functions of the Node class declared in Node.h.

**Filereader.h:** This header file declares a function readandInsert() responsible for reading student records from a file and inserting them into the linked list.

**Filereader.cpp:** This file defines the readandInsert() function declared in Filereader.h. It reads records from a file, parses them, and inserts them into the linked list.

**source.cpp:** This file contains the main function where the program execution begins. It interacts with users, displays menu options, and calls appropriate functions based on user input.

## Functions

LinkedList Class

**Insert_Record:** Inserts a new student record into the linked list.

**Search_Record:** Searches for a student record by ID and displays its details.

**Search_Record_fname:** Searches for a student record by first name and displays details of all matching records.

**Delete_Record:** Deletes a student record by ID from the linked list.

**Update_record:** Updates the details of a student record by ID.

**SaveToFile:** Saves the student records from the linked list to a file.

readandInsert Function

**readandInsert:** Reads student records from a file and inserts them into the linked list.

## Other Functions

**housekeeping:** Displays a menu to the user.

**print_execution_time:** Measures and prints the execution time of a specific operation.

**Execution Flow**

The program begins execution in the main() function.

It reads student records from a file using readandInsert() and inserts them into the linked list.

It displays a menu to the user and waits for user input.

Based on the user's choice, it calls appropriate functions to perform operations like adding, deleting, searching, or updating student records.

The program continues to display the menu until the user chooses to exit.

## Usage

Compile the program using a C++ compiler.

Ensure that the input file "students.txt" containing student records is present in the same directory as the executable.

Run the program and follow the on-screen instructions to perform various operations on student records.

## Note

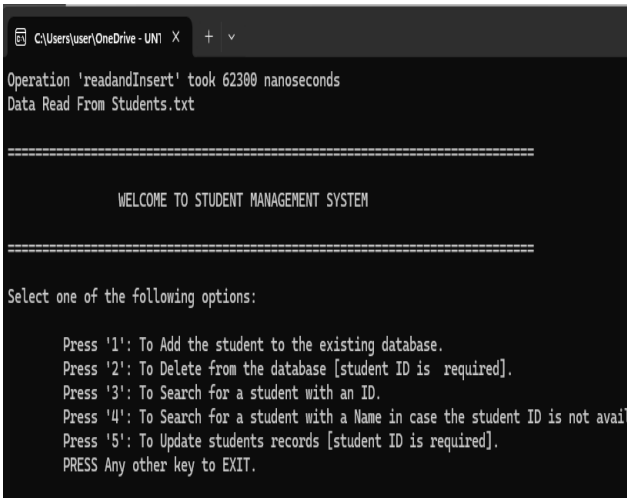
Ensure that the input file "students.txt" follows the format: ID, First Name, Last Name, Date of Birth, Street, City, State, Zip Code.

The program assumes valid input and may not handle invalid input gracefully.

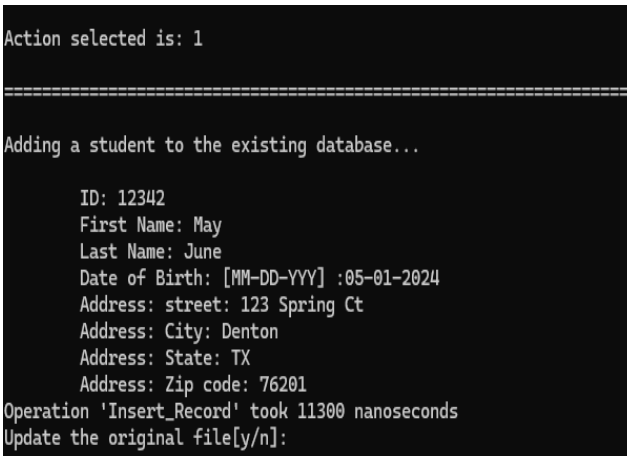
## Some of the screenshots of the all the functionality:

Here are some of the screenshots from the Linked list Student management System using linked list.

*Menu page*

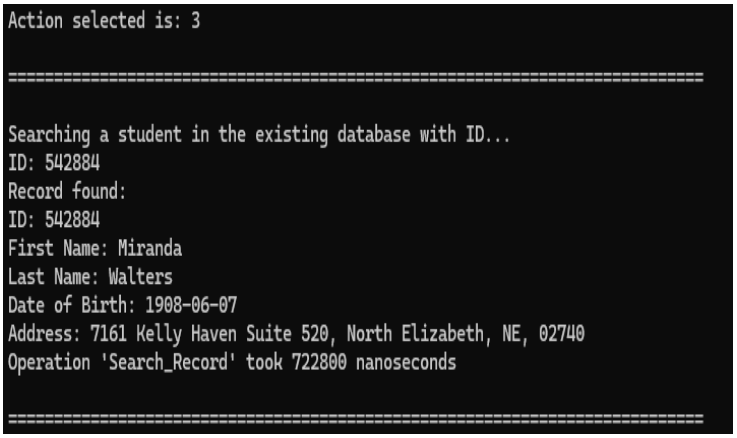



*Option1: Insert*

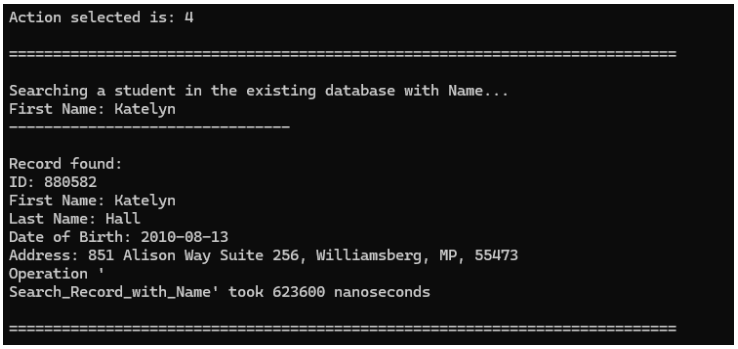



*Option2: Delete*

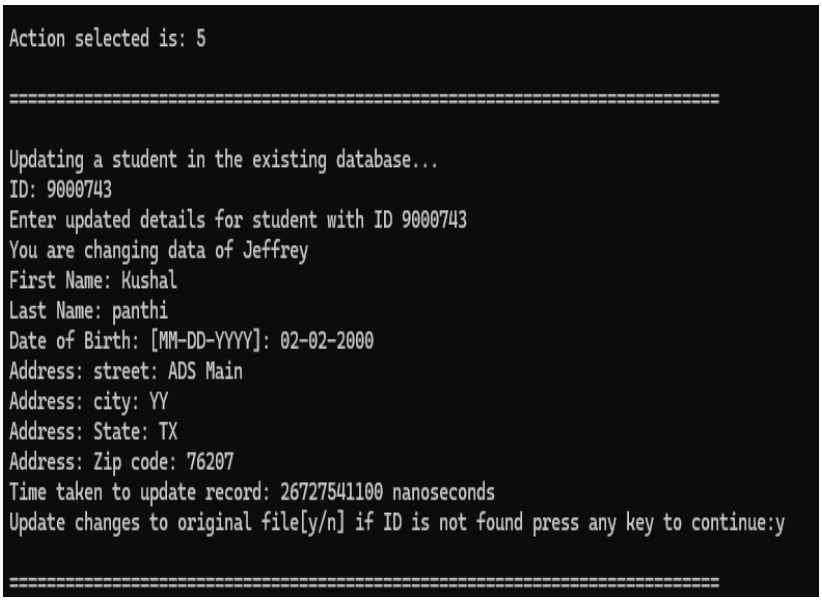

*Option3: Search*




*Option4: Search with name*



*Option5: Updating:*



*In our program Inserting is done at the end.*

| Time in Nanoseconds for linked list | | | | |
|---|---|---|---|---|
| **Actions** | 100 data sets | 10000 data sets | 100000 data sets | Suggestive Conclusion |
| **File_reading** | 360000 | 708547000 | 130377449600 | O(n) |
| **New_Insert at end** | 12700 | 1099700 | 95272100 | O(n) |
| **Updating_1st_position** | 11400194100 | 12472483400 | 12123987200 | O(1) |
| **Updating _last_position** | 11539630400 | 16368036000 | 32200129800 | O(n) |
| **Searching_with_id_random** | 2093500 | 2531000 | 4100900 | O(n) |
| **last_postion_Searching_ID** | 963300 | 1807200 | 4861500 | O(n) |
| **first_postion_Searching_ID** | 2124900 | 2663600 | 2077500 | O(1) |
| **Searching_with_name_random** | 2620300 | 48114800 | 831926700 | O(n) |
| **last_postion_Searching_name** | 914700 | 5806800 | 605537100 | O(n) |
| **Deletion_beginning** | 390300 | 364700 | 384960 | O(1) |
| **Deletion_ending** | 331100 | 878900 | 4638400 | O(n) |
| **Deletion_random** | 379600 | 796900 | 3045300 | O(n) |

***- although the time are given the time is for given instance at given time only (CPU usage, Memory allocation and other environment factors were not considered and may differ from one system to another)***

*This data matches with the time complexity published by*
*https://www.geeksforgeeks.org/time-and-space-complexity-of-linked-list/*

| Operation | Time Complexity (Singly Linked List) |
|---|---|
| Accessing by Index | O(n) |
| Insertion at Beginning | O(1) |
| Insertion at End | O(n) |
| Insertion at Given Position | O(n) |
| Deletion at Beginning | O(1) |
| Deletion at End | O(n) |
| Deletion at Given Position | O(n) |
| Searching | O(n) |

Here:

**Time Complexity of Searching (Finding an Element):**

Best Case: O(1) - If the element is found at the head of the list.

Worst Case: O(n) - If the element is at the end of the list or not present.

Average Case: O(n) - Similar to the worst case, as each element may need to be checked on average.

**Time Complexity of Insertion (Adding an Element):**

Best Case: O(1) - If inserting at the beginning of the list.

Worst Case: O(n) - If inserting at the end or in the middle of the list, requiring traversal.

Average Case: O(n) - Similar to the worst case, as traversal may be needed on average.

**Time Complexity of Deletion (Removing an Element):**

Best Case: O(1) - If deleting the first element.

Worst Case: O(n) - If deleting the last element or one in the middle, requiring traversal.

Average Case: O(n) - Similar to the worst case, as traversal may be needed on average.

**Some errors:**

While searching by name, multiple people had the same name, potentially causing issues with the time complexity of searching.

Determining the time complexity in updating is challenging due to user input delays. We assumed similar input delays in all three cases.

# Student System Management Program Documentation (Binary Search Tree)

- Some tweaks in faker, we decided to only opt for name that encompasses student full name, all the general code of faker was same from the faker used for linked list.
- Instead of first name, last name only Name in place

This C++ code implements a Binary Search Tree (BST) with AVL balancing. Here's a breakdown of the key components and functionalities:

## Struct Student:

Defines the structure of a student node with various attributes such as ID, name, date of birth, address, etc. Each student node has pointers to left and right children.

## Functions:

**createStudent:** Creates a new student node with the given attributes.

height: Computes the height of a node in the AVL tree.

**balanceFactor:** Computes the balance factor of a node.

**rightRotate** and **leftRotate:** Perform right and left rotations to balance the AVL tree.

**insertStudent:** Inserts a new student into the AVL tree while maintaining balance.

**minValueStudent:** Finds the node with the minimum ID in the AVL tree.

**deleteStudentByID:** Deletes a student node by ID from the AVL tree while maintaining balance.

**searchStudentByID** and **searchStudentByName**: Searches for a student by ID or name in the AVL tree.

**updateStudent:** Updates the details of a student.

**readStudentsFromFile** and **writeStudentToFile:** Read student data from a file into the AVL tree and write student data from the AVL tree back to a file, respectively.

## Main Functionality:

**The main function:**

Reads student data from a file named "students.txt" into the AVL tree.

Provides a menu-driven interface for various operations such as adding, deleting, searching, and updating student records.

The timing functionality measures the time taken for specific operations like insertion, deletion, and search.

**User Interface:**

The housekeeping function displays a welcome message and presents options for the user to choose from.

**Screen shots:**

**note the student data were 100 while these screenshots were taken so time is very small*

**Menu:**

```
Time taken to read the input file: 234500 nanoseconds

================================================================================

                WELCOME TO STUDENT MANAGEMENT SYSTEM(BST)

================================================================================

Select one of the following options:

        Press '1': To Add a student to the existing database.
        Press '2': To Delete a student from the database [student ID is required].
        Press '3': To Search for a student with an ID.
        Press '4': To Search for a student with a Name in case the student ID is not available.
        Press '5': To Update student records [student ID is required].
        PRESS Any other key to EXIT.

Action selected is:
```

**OPTION 1: INSERT**

```
Action selected is: 1
Enter student ID: 123444
Enter student full name: Kushal
Enter student date of birth: 02-02-2002
Enter student street: ASD Main
Enter student city: Denton
Enter student state: TX
Enter student zip: 76207
Time taken for deletion: 2800 nanoseconds

================================================================================

                WELCOME TO STUDENT MANAGEMENT SYSTEM(BST)
```

**Option 3: Search by ID**

```
Action selected is: 3
Enter student ID to search: 4305533
Searching for student with ID 4305533:
ID: 4305533
Name: Jill Rose
DOB: 1960-11-29
Address: 9741 Timothy Crest Apt. 923, South Thomasbury, GU, 72204
Time taken for search: 1200 nanoseconds

================================================================================

                WELCOME TO STUDENT MANAGEMENT SYSTEM(BST)

================================================================================

Select one of the following options:

        Press '1': To Add a student to the existing database.
        Press '2': To Delete a student from the database [student ID is required].
        Press '3': To Search for a student with an ID.
        Press '4': To Search for a student with a Name in case the student ID is not available.
        Press '5': To Update student records [student ID is required].
        PRESS Any other key to EXIT.

Action selected is:
3205375,Steven Reyes,1957-03-23,247 Hines Squares Apt. 947,Richardberg,MI,47380
3292200,Jason Gomez,1958-01-31,8395 David Harbor Apt. 380,North Ambershire,SC,93336
4305533,Jill Rose,1960-11-29,9741 Timothy Crest Apt. 923,South Thomasbury,GU,72204
```

**Option4: Search by name**

```
Action selected is: 4
Enter student name to search: Jose Thomas
Time taken for search: 5700 nanoseconds
Searching for student with name Jose Thomas:
ID: 5442786
Name: Jose Thomas
DOB: 1951-04-04
Address: 86985 Jasmine Fords, Stacybury, LA, 95109

================================================================================

                WELCOME TO STUDENT MANAGEMENT SYSTEM(BST)

================================================================================

Select one of the following options:

        Press '1': To Add a student to the existing database.
        Press '2': To Delete a student from the database [student ID is required].
        Press '3': To Search for a student with an ID.
        Press '4': To Search for a student with a Name in case the student ID is not available.
        Press '5': To Update student records [student ID is required].
        PRESS Any other key to EXIT.

Action selected is:
```

```
5722769,Thomas Collins,1967-08-01,236 Potter Islands Suite 263,Matthewfurt,KY,04508
5442786,Jose Thomas,1951-04-04,86985 Jasmine Fords,Stacybury,LA,95109
```

## Option 5: Updating

```
Select one of the following options:

        Press '1': To Add a student to the existing database.
        Press '2': To Delete a student from the database [student ID is required].
        Press '3': To Search for a student with an ID.
        Press '4': To Search for a student with a Name in case the student ID is not available.
        Press '5': To Update student records [student ID is required].
        PRESS Any other key to EXIT.

Action selected is: 5
Enter student ID to update: 5227181
Current student details:
ID: 5227181
Name: Victor Moody
DOB: 1992-09-19
Address: 156 Miller Rapid, East Annborough, IL, 61787
Enter updated student details:
Name: John
DOB: 02-04-1999
Street: 1st main stree
City: Seatlle
State: WA
ZIP: 8023
Student details updated successfully.
Time taken for update: 39404416800 nanoseconds
```

Time datasheet:

| Actions | Time in nanoseconds |
|---|---|
| File reading and inserting | 153973500 |
| New insert | 9700 |
| Delete | 5500 |
| Random_Search by ID | 551300 |
| Best case (search the 1st position) | 415400 |
| Worst case (search the last posion) | 522500 |
| Random_Search by Name | 4188800 |
| Name:Best case (search the 1st position) | 551000 |
| Name: Worst case (search the last posion) | 7633900 |
| Updating_random | 4683995300 |
| Updating_first position | 2528294400 |
| Updating_last position | 4912364700 |
| Deleting_random | 5700 |
| Deleting_1st position | 6000 |
| Deleting last position | 5300 |

*- although the time are given the time is for given instance at given time only (CPU usage, Memory allocation and other environment factors were not considered and may differ from one system to another*

**File Reading and Inserting:**

Reading a file and inserting each student record into the AVL tree requires iterating through each record once, resulting in a time complexity of O(n), where n is the number of student records in the file.

**New Insert:**

Inserting a new student into the AVL tree involves traversing the tree to find the correct position for insertion. Since AVL trees are balanced, the time complexity for insertion is O(log n), where n is the number of nodes in the tree.

**Delete:**

Deleting a student by ID from the AVL tree also requires traversing the tree to find the node to delete. The time complexity for deletion in AVL trees is O(log n), where n is the number of nodes in the tree.

**Random Search by ID:**

Searching for a student by ID involves traversing the AVL tree from the root to the node with the specified ID. In the average case, the time complexity for search is O(log n), where n is the number of nodes in the tree.

**Best Case (Search the 1st Position):**

The best-case scenario for searching by ID occurs when the desired ID is located at the root of the AVL tree. In this case, the time complexity is O(1).

**Worst Case (Search the Last Position):**

The worst-case scenario for searching by ID occurs when the desired ID is located at the deepest level of the AVL tree. In this case, the time complexity is O(log n), where n is the number of nodes in the tree.

**Random Search by Name:**

Searching for a student by name involves traversing the AVL tree based on the alphabetical order of names. In the average case, the time complexity for search is O(log n), where n is the number of nodes in the tree.

**Name: Best Case (Search the 1st Position):**

The best-case scenario for searching by name occurs when the desired name is located at the root of the AVL tree. In this case, the time complexity is O(1).

Name: Worst Case (Search the Last Position):

The worst-case scenario for searching by name occurs when the desired name is located at the deepest level of the AVL tree. In this case, the time complexity is O(log n), where n is the number of nodes in the tree.

**Updating Random:**

Updating a student's record involves searching for the student by ID and then modifying the attributes. Since searching by ID has a time complexity of O(log n) and updating the record itself is O(1), the overall time complexity is O(log n).

**Updating First Position:**

If the first student's record is updated, the time complexity is O(log n) as it involves searching by ID in the AVL tree.

**Updating Last Position:**

Similarly, updating the last student's record also has a time complexity of O(log n) as it requires searching by ID in the AVL tree.

**Deleting Random:**

Deleting a random student's record involves searching for the student by ID and then deleting it. Since searching by ID has a time complexity of O(log n) and deletion itself is O(log n), the overall time complexity is O(log n).

**Deleting 1st Position:**

Deleting the first student's record also has a time complexity of O(log n) as it requires searching by ID in the AVL tree.

**Deleting Last Position:**

Deleting the last student's record also has a time complexity of O(log n) as it involves searching by ID in the AVL tree.

Notable changes:
As the code was divided among the team members, we used to two separate faker code but have similar foundation to code. Some generic output and layout changes in user interfaces. Most of the things were kept similar.

**Conclusion:**

When considering the choice between a singly linked list and a balanced binary search tree like AVL for a student management system, several factors come into play. AVL trees offer superior search efficiency with logarithmic time complexity for operations based on student ID or name, compared to the linear time complexity of singly linked lists. Additionally, AVL trees maintain balance during insertion and deletion, resulting in efficient performance for these operations, whereas linked lists may require traversal for maintaining sorted order, impacting their efficiency. Despite AVL trees requiring more memory overhead and having a more complex implementation due to balancing requirements, they offer inherent sorting based on IDs or names, which can be advantageous for certain queries. On the other hand, singly linked lists are simpler to implement and maintain, making them suitable for applications with sequential access or frequent insertions and deletions at the beginning or end of the list. Overall, AVL trees are generally preferred for student management systems requiring efficient searching, insertion, and deletion based on IDs or names, while singly linked lists may suffice for simpler applications with less stringent performance requirements.