Student Record

This C++ program manages student information in a text file ("students.txt"). It allows adding, deleting, and displaying student information using a menu-driven interface. Let's go through the code line by line:

1. `#include` directives:
   - The program includes `iostream` for input and output.
   - The program includes `fstream` for file handling.
   - The program includes `string` for using string data type.

2. Namespace declaration:
   - `using namespace std;` allows the program to use standard library objects without the `std::` prefix.

3. `struct` declaration:
   - A `struct` named `Student` is declared to represent a student's details.
   - The `Student` struct contains four fields: `rollNumber` (int), `name` (string), `division` (string), and `address` (string).

4. `addStudent` function:
   - This function takes a `Student` object as a parameter and appends the student's details to a text file (`students.txt`).
   - The file is opened in append mode (`ios::app`), and the student's details are written as a line in the file.
   - If the file cannot be opened for writing, an error message is displayed.
   - After writing the student's details, the file is closed, and a success message is displayed.

5. `deleteStudent` function:
   - This function takes an `int` parameter (`rollNumber`) to identify the student to be deleted.
   - The function reads each line from the `students.txt` file and writes it to a temporary file (`temp.txt`) unless the roll number matches the one to be deleted.
   - If a matching roll number is found, it sets a `found` flag to `true` and does not write the student's details to the temporary file.
   - After processing all lines, the original file (`students.txt`) is deleted, and the temporary file is renamed to `students.txt`.
   - If the student is found and deleted, a success message is displayed; otherwise, a "Student not found" message is displayed.

6. `displayStudent` function:
   - This function takes an `int` parameter (`rollNumber`) to identify the student to be displayed.
   - The function reads each line from the `students.txt` file, extracting student details.
   - If a student's roll number matches the provided roll number, the student's details are displayed, and a `found` flag is set to `true`.
   - If no matching student is found, a "Student not found" message is displayed.

7. `main` function:
   - The main function contains a loop that continuously displays a menu with options: Add Student, Delete Student, Display Student, and Exit.
   - Based on the user's choice, the corresponding function (`addStudent`, `deleteStudent`, or `displayStudent`) is called.

- The loop exits when the user selects the "Exit" option.

In summary, the program uses text files to manage student information, allowing the user to add, delete, and display student records through a simple command-line menu.

Telephone

This C++ program implements a hash table to manage a simple telephone book database. The hash table is used to efficiently store, search, update, and delete records based on a person's name. Let's go through the code line by line:

1. `#include` directives:
   - `#include <iostream>`: Required for input and output operations.

2. Namespace declaration:
   - `using namespace std;` allows the program to use standard library objects without the `std::` prefix.

3. `class node` declaration:
   - The `node` class stores information for each record in the database.
   - It contains private members: `name` (string), `telephone` (string), and `key` (int) to represent a person's details.
   - The class has a default constructor that initializes `key` to 0.
   - A `friend` class relationship is declared with the `hashing` class to allow access to `node`'s private members.

4. `ascii_generator` function:
   - This function calculates the ASCII sum of the input string (`s`) and returns it modulo 100.
   - The ASCII sum serves as a key for hashing.

5. `class hashing` declaration:
   - This class represents the hash table and contains the data array, `data`, of type `node`, and a `size` constant set to 100.
   - The class includes various methods for creating, searching, updating, and deleting records, as well as displaying the hash table.

6. `hashing::hashing()` constructor:
   - The constructor initializes `k` to 0.

7. `create_record` function:

- This function creates a record in the hash table.
- The ASCII sum (`k`) of the name (`n`) is calculated using `ascii_generator` and the index is computed using `k % size`.
- The function then checks for an empty slot (where `key` is 0) in the data array, and stores the record.
- If there is a collision, it uses linear probing (incrementing the index) to find an empty slot.

8. `search_record` function:
- This function searches for a record in the hash table based on the provided name.
- The key is calculated using `ascii_generator` and the index is computed.
- It then searches the hash table using linear probing until the record is found or the search is unsuccessful.

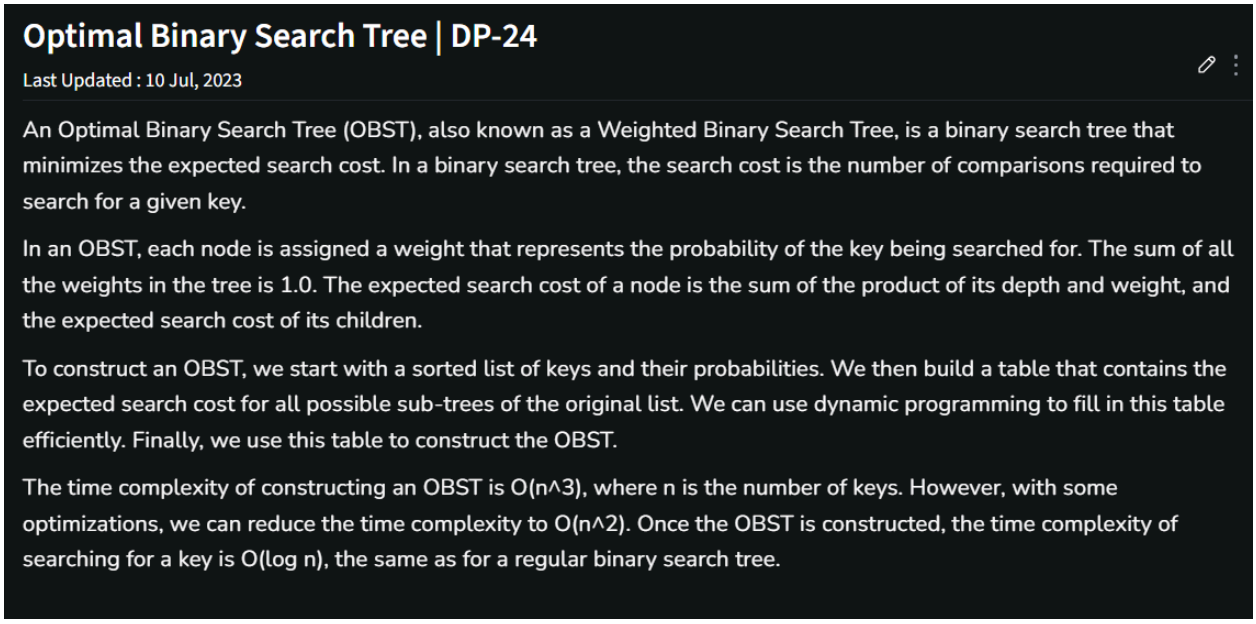
9. `delete_record` function:
- This function deletes a record from the hash table based on the provided name.
- The key is calculated using `ascii_generator` and the index is computed.
- If a record with the same key and name is found, it sets the node's `key` to 0 and its name and telephone to empty strings.

10. `update_record` function:
- This function updates a record's telephone number in the hash table based on the provided name.
- The key is calculated using `ascii_generator` and the index is computed.
- If a record with the same key and name is found, it prompts the user for a new telephone number and updates the record.

11. `display_record` function:
- This function displays the contents of the hash table.
- It prints the name and telephone of each node in the hash table that has a non-zero key.

12. `main` function:
- The `main` function starts a menu-driven program loop that allows the user to perform different operations on the hash table.
- It allows the user to create, search, update, and delete records, as well as display all records.
- The loop ends when the user selects the "Exit" option.

OBT



Optimal Binary Search Tree | DP-24

Last Updated : 10 Jul, 2023

An Optimal Binary Search Tree (OBST), also known as a Weighted Binary Search Tree, is a binary search tree that minimizes the expected search cost. In a binary search tree, the search cost is the number of comparisons required to search for a given key.

In an OBST, each node is assigned a weight that represents the probability of the key being searched for. The sum of all the weights in the tree is 1.0. The expected search cost of a node is the sum of the product of its depth and weight, and the expected search cost of its children.

To construct an OBST, we start with a sorted list of keys and their probabilities. We then build a table that contains the expected search cost for all possible sub-trees of the original list. We can use dynamic programming to fill in this table efficiently. Finally, we use this table to construct the OBST.

The time complexity of constructing an OBST is O(n^3), where n is the number of keys. However, with some optimizations, we can reduce the time complexity to O(n^2). Once the OBST is constructed, the time complexity of searching for a key is O(log n), the same as for a regular binary search tree.

This C++ program calculates the cost of an optimal binary search tree (BST) given a set of keys and their frequencies. An optimal BST minimizes the total search time (or cost) for a set of keys with known frequencies. The program follows a dynamic programming approach to calculate the cost of the optimal BST. Let's go through the code line by line:

1. `#include` directives:
   - `#include <iostream>`: Required for input and output operations.
   - `#include <limits.h>`: Required for accessing the constant `INT_MAX` representing the maximum integer value.

2. Namespace declaration:
   - `using namespace std;` allows the program to use standard library objects without the `std::` prefix.

3. `sum` function:
   - This function calculates the sum of frequencies (`freq`) from index `i` to `j`.
   - It initializes a variable `s` to 0 and then iterates from index `i` to `j`, accumulating the frequencies in `s`.
   - The function returns the calculated sum.

4. `optCost` function:
   - This function calculates the cost of the optimal binary search tree (BST) using a dynamic programming approach.

- The function takes three parameters: an array of keys (`keys`), an array of frequencies (`freq`), and the number of keys (`n`).
- It initializes a 2D array (`cost`) to store the cost of building optimal BSTs for different ranges of keys.
- The function starts by initializing the diagonal of the `cost` array with the frequencies (base cases).
- It then uses a loop to iterate through different lengths of subarrays of keys, from 2 to `n`.
- For each length, it iterates through the possible starting indices of the subarray (`i`) and calculates the cost of constructing an optimal BST for the subarray from `i` to `j`.
- The cost is calculated for each root (`r`) within the subarray, considering the cost of left and right subtrees and the sum of frequencies in the range.
- The minimum cost is stored in the `cost` array.
- Finally, the function returns the optimal cost for the entire array (from `0` to `n-1`).

5. `main` function:
- The main function reads the number of keys (`n`) from the user.
- It initializes arrays for keys and frequencies and takes input from the user for each key and frequency.
- The function calls the `optCost` function with the keys, frequencies, and the number of keys as arguments.
- It prints the cost of the optimal BST calculated by the `optCost` function.
- The program ends by returning `0` to indicate successful execution.

In summary, the program calculates the cost of constructing an optimal binary search tree using a dynamic programming approach and provides the user with an interactive interface to input the keys and their frequencies.

Employee

This C++ program manages a list of employees stored in a binary file (`employee.dat`). It provides a menu-driven interface to allow users to add, display, and delete employees based on their employee ID (`empID`). The program uses the `Employee` struct to store employee details such as ID, name, designation, and salary. Let's go through the code line by line:

1. `#include` directives:
   - `#include <iostream>`: For input and output operations.
   - `#include <fstream>`: For file handling operations.
   - `#include <cstring>`: For working with C-style strings.

2. Namespace declaration:
   - `using namespace std;` allows the program to use standard library objects without the `std::` prefix.

3. `struct Employee`:
   - The `Employee` struct defines the structure for storing employee details.
   - It contains four fields: `empID` (integer), `name` (C-style string of size 50), `designation` (C-style string of size 50), and `salary` (double).

4. `addEmployee` function:
   - This function adds a new employee record to the `employee.dat` binary file.
   - It opens the file in binary append mode (`ios::binary | ios::app`) to add new records to the end of the file.
   - If the file fails to open, an error message is displayed.
   - An `Employee` struct (`emp`) is created, and the user inputs the employee ID, name, designation, and salary.
   - The function uses `reinterpret_cast` to cast the address of `emp` to a `char*` pointer for writing the binary data to the file.
   - Once the employee record is written to the file, the file is closed, and a success message is displayed.

5. `displayEmployee` function:
   - This function displays the details of an employee given their `empID`.
   - The function opens the `employee.dat` file in binary read mode (`ios::binary`).
   - If the file fails to open, an error message is displayed.
   - An `Employee` struct (`emp`) is read from the file one record at a time.

- If the `empID` of the current record matches the input `empID`, the function displays the employee's details and sets a `found` flag to true.
- If the end of the file is reached without finding a matching `empID`, a "not found" message is displayed.
- The file is then closed.

6. `deleteEmployee` function:
- This function deletes an employee record given their `empID`.
- It opens the `employee.dat` file in binary read mode and a temporary file (`temp.dat`) in binary write mode.
- If either file fails to open, an error message is displayed.
- The function reads records from the `employee.dat` file one by one.
- If the current record's `empID` matches the input `empID`, the record is not written to the temporary file, indicating it is deleted.
- Otherwise, the record is written to the temporary file.
- After processing all records, the files are closed.
- The original `employee.dat` file is removed, and the temporary file is renamed to `employee.dat`.
- If the `empID` was not found, a "not found" message is displayed.

7. `main` function:
- The main function starts a loop to repeatedly display a menu and allow the user to select one of the four options: add an employee, display an employee, delete an employee, or exit the program.
- Depending on the user's choice, the corresponding function (`addEmployee`, `displayEmployee`, or `deleteEmployee`) is called.
- If the user chooses to exit the program, the loop ends, and a goodbye message is displayed.

In summary, this program uses a binary file (`employee.dat`) to store, add, display, and delete employee records based on their employee ID. The user interacts with the program through a menu-driven interface to perform these operations.

AVL Tree Data Structure
An AVL tree defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.

The difference between the heights of the left subtree and the right subtree for any node is known as the balance factor of the node.

The AVL tree is named after its inventors, Georgy Adelson-Velsky and Evgenii Landis, who published it in their 1962 paper “An algorithm for the organization of information”.

Example of AVL Trees:

AVL tree
AVL tree

The above tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.

FLIGHT

This C++ code checks if a graph represented as an adjacency matrix is connected or not. Here is an explanation of the code line by line:

1. `#include <iostream>`: This header file is included to use standard input and output functions such as `cin` and `cout`.

2. `#include<queue>`: This header file is included to use the queue data structure.

3. `#define n 10`: This line defines a constant named `n` with a value of 10. This constant is used to define the size of the adjacency matrix.

4. `int adj_m[n][n];`: This line declares an adjacency matrix `adj_m` with size `n x n` (10x10). The matrix is used to represent the graph. Each entry `adj_m[i][j]` holds the fuel required to travel from city `i` to city `j`.

5. `void traverse(int u, int x, bool visited[])`: This function performs a depth-first search (DFS) starting from vertex `u`. It uses a `visited` array to keep track of which vertices have been visited during the traversal.

6. `visited[u] = true;`: This line marks vertex `u` as visited in the `visited` array.

7. `for(int v = 0; v < x; v++)`: This loop iterates through all the vertices in the graph.

8. `if(adj_m[u][v] != 0)`: This condition checks if there is a connection (fuel required) between vertices `u` and `v` in the adjacency matrix.

9. `if(!visited[v])`: This condition checks if vertex `v` has not been visited yet.

10. `traverse(v, x, visited);`: If the above condition is true, the function calls itself recursively to traverse vertex `v`.

11. `bool isConnected(int x)`: This function checks if the graph is connected or not by iterating through each vertex and calling the `traverse` function.

12. `bool *vis = new bool[n]`: This line dynamically allocates a boolean array of size `n` to keep track of visited vertices.

13. `for(int u = 0; u < x; u++)`: This loop iterates through each vertex in the graph.

14. `for(int i = 0; i < x; i++)`: This loop initializes the `vis` array to `false` for each iteration.

15. `traverse(u, x, vis);`: This line calls the `traverse` function for each vertex `u`.

16. `if(!vis[i])`: This condition checks if any vertex has not been visited after calling `traverse`. If so, the graph is not connected.

17. `return false;`: If a vertex is not visited, the function returns `false` indicating the graph is not connected.

18. `return true;`: If all vertices have been visited, the function returns `true` indicating the graph is connected.

19. `using namespace std;`: This line declares the use of the standard namespace, allowing the use of standard library classes and functions without the `std::` prefix.

20. `int main()`: This is the main function where program execution begins.

21. `int x;`: This line declares an integer variable `x` to hold the number of cities.

22. `cout << "Enter number of cities :: ";`: This line prints a prompt asking for the number of cities.

23. `cin >> x;`: This line reads the user input and stores it in the variable `x`.

24. `char arr[x][20];`: This line declares a 2D character array `arr` to store the names of cities.

25. `for (int i = 0; i < x; i++)`: This loop iterates through each city.

26. `cout << "Enter " << i + 1 << " city name :: ";`: This line prompts the user to enter the name of each city.

27. `cin >> arr[i];`: This line reads the name of each city and stores it in `arr`.

28. `for(int i = 0; i < x; i++)`: This loop iterates through each city.

29. `for(int j = 0; j < x; j++)`: This loop iterates through each pair of cities.

30. `cout << "\nIs city " << arr[i] << " connected to city " << arr[j] << endl << "Enter y if yes :: ";`: This line prompts the user to enter whether the current pair of cities (`i` and `j`) are connected.

31. `char ch;`: This line declares a character variable `ch` to store the user's response.

32. `cin >> ch;`: This line reads the user's response and stores it in `ch`.

33. `if(ch == 'y' || ch == 'Y')`: This condition checks if the user's response indicates a connection between the cities.

34. `int a;`: This line declares an integer variable `a` to store the fuel required to travel from city `i` to city `j`.

35. `cout << "Enter fuel required to go from " << arr[i] << " to " << arr[j] << "\nEnter :: ";`: This line prompts the user to enter the amount of fuel required to travel between the cities.

36. `cin >> a;`: This line reads the fuel required and stores it in `a`.

37. `adj_m[i][j] = a;`: This line stores the fuel required in the adjacency matrix.

38. `adj_m[i][j] = 0;`: This line sets the adjacency matrix value to zero if the cities are not connected.

39. `cout << "\n-------------------------------\n";`: This line prints a separator line to improve readability.

40. `cout << "ADJ MAT\n";`: This line prints a heading for the adjacency matrix.

41. `for(int i = 0; i < x; i++)`: This loop iterates through each row of the adjacency matrix.

42. `for(int j = 0; j < x; j++)`: This loop iterates through each column of the adjacency matrix.

43. `cout << adj_m[i][j] << "\t";`: This line prints each element of the adjacency matrix, separated by tabs.

44. `cout << "\n";`: This line prints a newline character at the end of each row of the adjacency matrix.

45. `cout << "\n-------------------------------\n";`: This line prints another separator line.

46. `if(isConnected(x))`: This line calls the `isConnected` function with `x` (number of cities) as an argument. If the function returns `true`, it indicates the graph is connected.

47. `cout << "The Graph is connected.";`: If the graph is connected, this line prints the message "The Graph is connected."

48. `else`: If the graph is not connected, the program will execute the following block of code.

49. `cout << "The Graph is not connected.";`: If the graph is not connected, this line prints the message "The Graph is not connected."

50. `return 0;`: This line indicates successful completion of the program and returns zero.

Difference Between BFS and DFS:

Parameters	BFS	DFS
Stands for	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
Data Structure	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
Definition	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
Conceptual Difference	BFS builds the tree level by level.	DFS builds the tree sub-tree by sub-tree.
Approach used	It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
Suitable for	BFS is more suitable for searching vertices closer to the given source.	DFS is more suitable when there are solutions away from source.
Applications	BFS is used in various applications such as bipartite graphs, shortest paths, etc.	DFS is used in various applications such as acyclic graphs and finding strongly connected components etc.