### Discussion of Terms

#### 1. Circular Lists

A circular list is a type of linked list where the last node points back to the first node, forming a circle. This structure allows for continuous traversal of the list without needing to reset to the head. Circular lists can be singly or doubly linked.

**Example:** In a circular singly linked list with nodes A, B, and C, the next pointer of C points back to A. This allows you to traverse the list starting from any node and eventually return to that node.

#### 2. Skip Lists

Skip lists are a probabilistic data structure that allows for fast search, insertion, and deletion operations. They consist of multiple layers of linked lists, where each higher layer acts as an express lane for the lower layers. This structure allows for logarithmic time complexity for search operations.

**Example:** A skip list might have a base layer with all elements and additional layers that skip over some elements, allowing for faster traversal. For instance, if you have a list of numbers, the first layer might contain all numbers, while the second layer might contain every second number.

#### 3. Recursive Definitions

A recursive definition is a definition that refers to itself in its own definition. This is commonly used in mathematics and computer science to define sequences or structures.

**Example:** The factorial function can be defined recursively as:

* ( n! = n \times (n-1)! ) for ( n > 0 )
* ( 0! = 1 )

#### 4. Tail Recursion vs Non-Tail Recursion

* **Tail Recursion**: A function is tail recursive if the recursive call is the last operation in the function. This allows for optimization by the compiler to reuse the current function's stack frame.

**Example:**

int tail\_recursive\_factorial(int n, int accumulator = 1)

{

if (n == 0) return accumulator;

return tail\_recursive\_factorial(n - 1, n \* accumulator);

}

**Non-Tail Recursion**: A function is non-tail recursive if there are operations after the recursive call, which means the current stack frame cannot be reused.

**Example:**

int non\_tail\_recursive\_factorial(int n) {

if (n == 0) return 1;

return n \* non\_tail\_recursive\_factorial(n - 1);

}

#### 5. Indirect Recursion

Indirect recursion occurs when a function calls another function, which in turn calls the first function. This can create a cycle of calls between two or more functions.

**Example:**

void functionA() {

// some code

functionB();

}

void functionB() {

// some code

functionA();

}

#### 6. Nested Recursion

Nested recursion occurs when a recursive function makes a call to itself within another recursive call. This can lead to complex recursive structures.

**Example:**

int nested\_recursion(int n) {

if (n > 100) return n - 10;

return nested\_recursion(nested\_recursion(n + 11));

}

#### 7. Excessive Recursion

Excessive recursion refers to a situation where a recursive function makes too many calls, leading to stack overflow or performance issues. This often happens with poorly designed recursive algorithms.

**Example:** A naive Fibonacci function can lead to excessive recursion:

int excessive\_fibonacci(int n) {

if (n <= 1) return n;

return excessive\_fibonacci(n - 1) + excessive\_fibonacci(n - 2);

}

### Complete C++ Implementation

#include <iostream>

#include <cstring>

using namespace std;

struct abyssinia\_bank {

int account\_num;

char account\_holder[20];

char account\_branch[20];

double balance;

struct abyssinia\_bank \*next;

};

struct abyssinia\_bank \*start\_ptr = nullptr;

void accept\_node() {

abyssinia\_bank \*new\_node = new abyssinia\_bank;

cout << "Enter account number: ";

cin >> new\_node->account\_num;

cout << "Enter account holder name: ";

cin >> new\_node->account\_holder;

cout << "Enter account branch: ";

cin >> new\_node->account\_branch;

cout << "Enter balance: ";

cin >> new\_node->balance;

new\_node->next = nullptr;

// Insert in sorted order based on account\_num

if (start\_ptr == nullptr || start\_ptr->account\_num > new\_node->account\_num) {

new\_node->next = start\_ptr;

start\_ptr = new\_node;

} else {

abyssinia\_bank \*current = start\_ptr;

while (current->next != nullptr && current->next->account\_num < new\_node->account\_num) {

current = current->next;

}

new\_node->next = current->next;

current->next = new\_node;

}

}

void display\_all\_nodes() {

abyssinia\_bank \*current = start\_ptr;

if (current == nullptr) {

cout << "No records found." << endl;

return;

}

while (current != nullptr) {

cout << "Account Number: " << current->account\_num << endl;

cout << "Account Holder: " << current->account\_holder << endl;

cout << "Account Branch: " << current->account\_branch << endl;

cout << "Balance: " << current->balance << endl;

cout << "------------------------" << endl;

current = current->next;

}

}

void display\_node\_by\_id(int account\_num) {

abyssinia\_bank \*current = start\_ptr;

while (current != nullptr) {

if (current->account\_num == account\_num) {

cout << "Account Number: " << current->account\_num << endl;

cout << "Account Holder: " << current->account\_holder << endl;

cout << "Account Branch: " << current->account\_branch << endl;

cout << "Balance: " << current->balance << endl;

return;

}

current = current->next;

}

cout << "Account not found." << endl;

}

int main() {

string username, password;

const string correct\_username = "BOA";

const string correct\_password = "2017";

int attempts = 0;

const int max\_attempts = 2;

// User authentication

cout << " # BANK OF ABYSSINIA #" << endl;

cout << endl;

while (attempts < max\_attempts) {

cout << "ENTER USER NAME: ";

cin >> username;

cout << "ENTER PASSWORD: ";

cin >> password;

if (username == correct\_username && password == correct\_password) {

cout << "LOGIN IS SUCCESSFUL" << endl;

cout << " | Welcome to APOLLO DIGITAL BANK OF ABYSSINIA |" << endl;

cout << endl;

int choice;

do {

cout << "1. Accept Node" << endl;

cout << "2. Display All Nodes" << endl;

cout << "3. Display Node by ID" << endl;

cout << "4. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

accept\_node();

break;

case 2:

display\_all\_nodes();

break;

case 3: {

int account\_num;

cout << "Enter account number to search: ";

cin >> account\_num;

display\_node\_by\_id(account\_num);

break;

}

case 4:

cout << "Exiting..." << endl;

break;

default:

cout << "Invalid choice. Please try again." << endl;

}

} while (choice != 4);

return 0; // Exit after successful login and menu interaction

} else {

attempts++;

cout << "Invalid username or password. Please try again." << endl;

if (attempts < max\_attempts) {

cout << "You have " << (max\_attempts - attempts) << " attempt(s) left." << endl;

}

}

}

cout << "You have exceeded the maximum number of login attempts. Exiting..." << endl;

return 0;

}

### Explanation of the Code

1. **Struct Definition**: The **abyssinia\_bank** struct defines the structure of each node in the linked list, containing account details and a pointer to the next node.
2. **Global Pointer**: **start\_ptr** is a global pointer that points to the head of the linked list.
3. **accept\_node Function**: This function creates a new node, accepts user input for the account details, and inserts the node into the linked list in sorted order based on **account\_num**.
4. **display\_all\_nodes Function**: This function traverses the linked list and displays all account details.
5. **display\_node\_by\_id Function**: This function searches for a node by **account\_num** and displays its details if found.
6. **main Function**: This function provides a simple menu for the user to interact with the program, allowing them to accept new nodes, display all nodes, or search for a node by account number.