ATM SYSTEM

Submitted by

KARTHIKEYA DEVARLA [RA2211003010192]
SIDDARTH JAVVADI[RA2211003010154]
GOWTHAM REDDY KOMMEPALLI[RA2211003010169]

Under the Guidance of

Dr.M.REVATHI

Assistant Professor, Department of Computing Technologies

In partial satisfaction of the requirements for the degree of

BACHELORS OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

NOVEMBER 2023

SRM INSTITUTION OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR-603203

BONAFIDE CERTIFICATE

Certified that this Course Project Report titled **"ATM SYSTEM"** is the bonafide work done by KARTHIKEYA **DEVARLA[RA2211003010192]**,**SIDDARTH JAVVADI[RA2211003010154]** and **GOWTHAM REDDY KOMMEPALLI[RA2211003010169]** who carried out under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

SIGNATURE
Faculty In-Charge
Dr. M.REVATHI
Assistant Professor
Department of Computing Technologies
SRM Institute of Science and Technology

HEAD OF THE DEPARTMENT
Dr. M. Pushpalatha
Professor and Head
Department of Computing Technologies
SRM Institute of Science and Technology

ABSTRACT

An ATM (Automated Teller Machine) system can be implemented using various data structures and algorithms to manage the functionality and data associated with it. Here's an overview of how data structures play a role in an ATM system:

Queue: A queue data structure can be used to manage the order of customers waiting to use the ATM. When a customer arrives at the ATM, they join the queue. This ensures that customers are served on a first-come, first-served basis.

Hash Tables: Hash tables can be used for fast retrieval of customer account information. Each customer's account information can be stored as key-value pairs in the hash table, with the account number or card number as the key. This allows quick access to customer data during transactions.

Linked Lists: Linked lists can be used to manage transaction histories. Each customer's transaction history can be stored as a linked list where each node represents a transaction. This allows for easy appending of new transactions and retrieval of transaction history.

Stacks: Stacks can be used to manage the state of the ATM during transactions. For example, when a customer initiates a transaction, the ATM can push the current state onto a stack. If the transaction is canceled or fails, the ATM can pop the previous state from the stack to return to the previous state.

Tree Structures: Tree structures like binary search trees (BSTs) can be used for organizing and searching customer account data efficiently. This is particularly useful when searching for specific account information based on account numbers.

Arrays: Arrays can be used to store the available denominations of cash in the ATM. This allows the ATM to dispense cash during withdrawals by selecting the appropriate denominations based on the requested amount.

Graphs: While not as common, graphs could be used in scenarios where complex ATM network topologies need to be represented. For instance, if multiple ATMs are connected in a network, a graph structure could be used to model the relationships between them and optimize cash replenishment routes.

Priority Queues: Priority queues can be used to manage transactions, giving priority to certain transactions, such as urgent transactions or transactions from premium customers.

Algorithms such as searching, sorting, and encryption are also crucial in implementing an ATM system. Searching algorithms help find customer account information efficiently, sorting algorithms can be used to maintain transaction histories in chronological order, and encryption is essential for securing sensitive data like PINs and transaction information.

DATA STRUCTURES USED

QUEUE:

A queue is a fundamental data structure in computer science and is widely used to manage and process data in a linear, first-in, first-out (FIFO) manner. In a queue, the element that has been in the data structure the longest is the first to be processed, and the element that has been in the data structure the shortest is the last to be processed. Queues are commonly used in various applications and algorithms. Here are some key characteristics and operations associated with queues:

Key Characteristics of a Queue:

FIFO (First-In, First-Out): The element that is added to the queue first is the one that gets removed first. This ensures that elements are processed in the order they were added.

Linear Data Structure: A queue is typically represented as a linear data structure, where elements are added at one end (the "rear" or "enqueue" end) and removed from the other end (the "front" or "dequeue" end).

Limited Access: Queues offer limited access to their elements. Typically, you can only access and manipulate the front and rear elements.

Operations on a Queue:

Enqueue (Insertion): This operation adds an element to the rear of the queue. It is also sometimes referred to as "push."

Dequeue (Removal): This operation removes and returns the element from the front of the queue. It is also sometimes referred to as "pop."

Peek (Front): This operation allows you to view the element at the front of the queue without removing it.

IsEmpty: This operation checks if the queue is empty, indicating that no elements are present.

Size (Length): This operation returns the number of elements in the queue.

Common Use Cases for Queues:

Task Scheduling: Queues are used to manage tasks or processes that need to be executed in a specific order, such as a print queue or a job scheduling system.

Breadth-First Search (BFS): Queues are employed in graph algorithms like BFS to explore nodes level by level.

Synchronization: Queues are useful for managing concurrent access to shared resources, like in multithreading and parallel computing.

Buffering: In data processing and I/O operations, queues can be used to buffer data to handle variations in input and output speeds.

Simulations: Queues are often used in simulations to represent waiting lines, such as customers waiting in line at a service counter.

Background Processing: Queues are used in systems that perform background tasks or asynchronous processing, like message queues in distributed systems.

DYNAMIC MEMORY ALLOCATION:-

Dynamic memory allocation in data structures refers to the process of allocating and managing memory for data structures at runtime. Unlike static memory allocation, where memory is allocated at compile time and the size is fixed, dynamic memory allocation allows data structures to grow or shrink as needed during program execution. This flexibility is crucial for managing data structures efficiently, especially when dealing with data of varying sizes or when you're unsure of the exact memory requirements in advance. Dynamic memory allocation is commonly used in various programming languages, including C, C++, and others. Here's an overview of dynamic memory allocation in data structures.

1. Dynamic Memory Allocation Functions:

In languages like C and C++, dynamic memory allocation is typically achieved through functions like `malloc`, `calloc`, `realloc`, and `free`. Here's a brief description of these functions:

`malloc`: Stands for "memory allocation." It is used to allocate a specified number of bytes of memory and returns a pointer to the first byte of the block.

`calloc`: Stands for "contiguous allocation." It is used to allocate memory for an array of elements, initializing them to zero.

`realloc`: Stands for "reallocate." It is used to change the size of a previously allocated memory block.

`free`: Used to deallocate or release memory that was previously allocated using `malloc`, `calloc`, or `realloc`.

2. Data Structures Utilizing Dynamic Memory Allocation:

Dynamic memory allocation is commonly used in various data structures, including:

Linked Lists: Linked lists can grow dynamically by allocating memory for new nodes as data is inserted. When elements are removed, memory can be deallocated.

Dynamically Sized Arrays: Arrays can be allocated dynamically to accommodate a variable number of elements. When the array needs to grow, a larger block of memory is allocated, and the elements are copied to the new memory location.

Trees: Binary search trees, AVL trees, and other tree-based structures allocate memory dynamically for nodes as new data is inserted.

Hash Tables: Hash tables can dynamically allocate memory to handle collisions and resizing when the load factor exceeds a certain threshold.

Graphs: Graph nodes and edges can be allocated dynamically to represent complex network structures.

3. Benefits and Considerations:

Dynamic memory allocation provides several advantages:

Flexibility: Data structures can adapt to changing data sizes during runtime.

Efficient Memory Usage: Memory is allocated only when needed, reducing memory wastage.

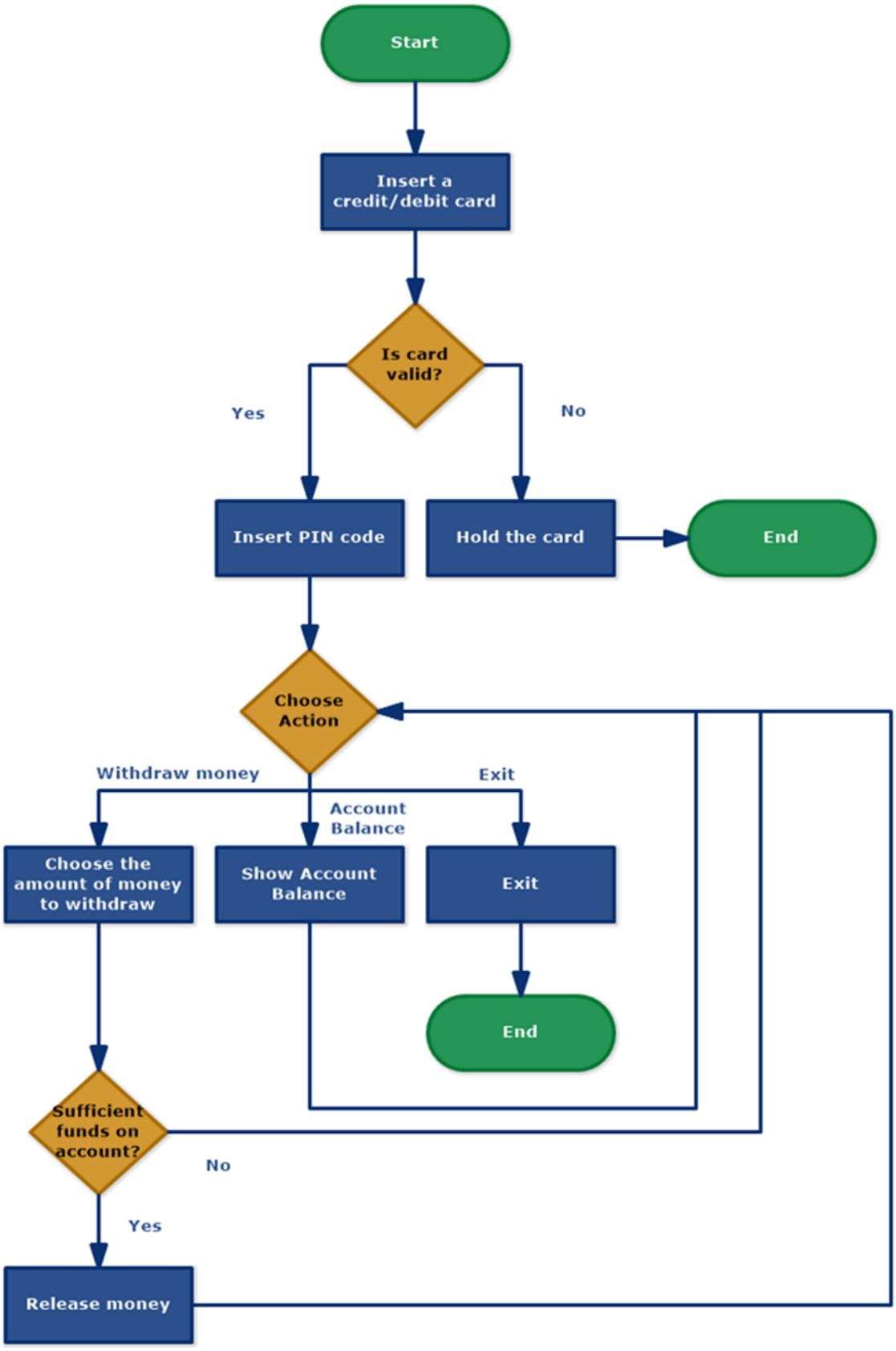
Avoiding Stack Limitations: Dynamic memory allocation is often used for large data structures that can't be accommodated on the stack.

However, it also comes with responsibilities:

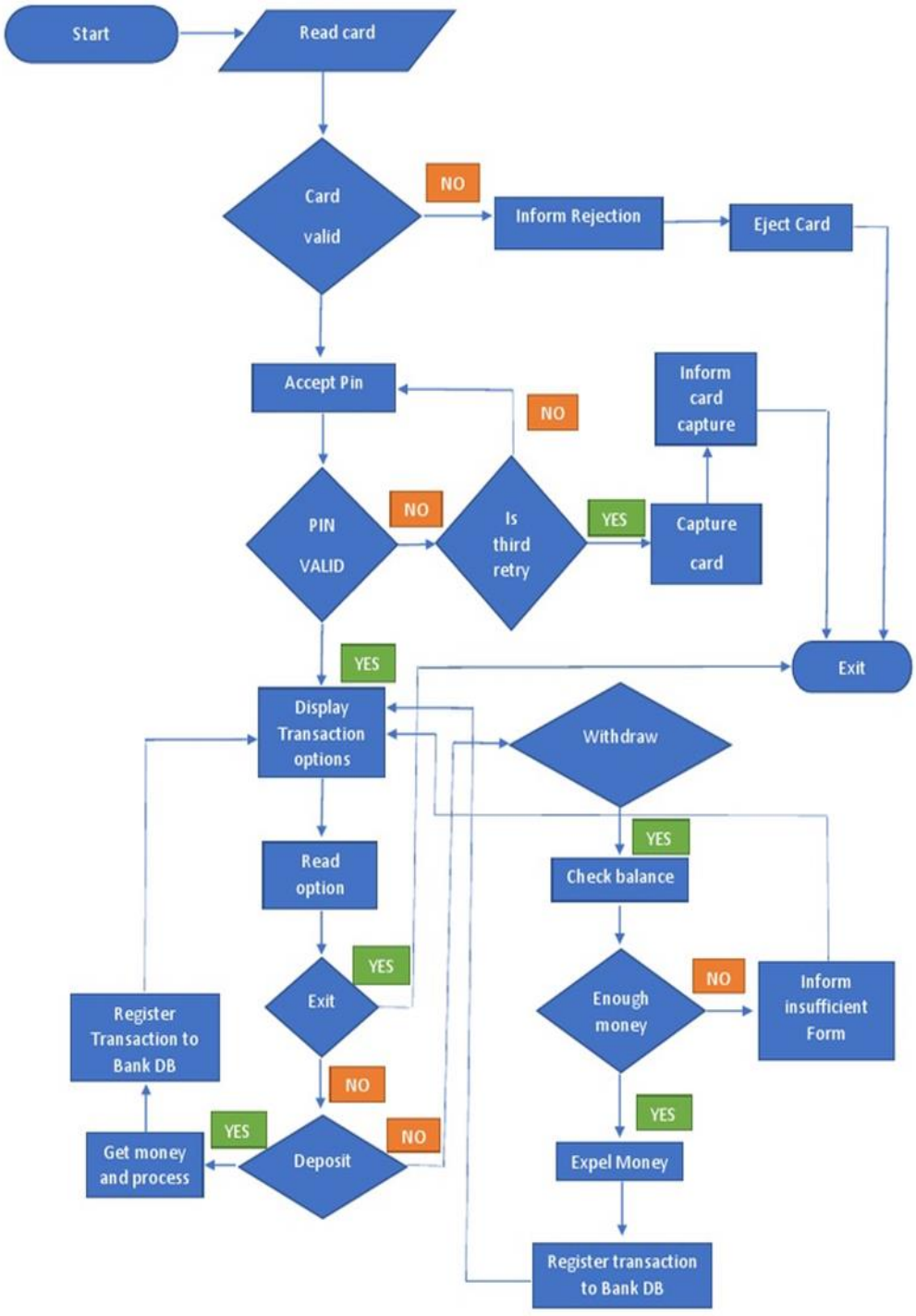
Proper Deallocation: Memory allocated dynamically should be freed when no longer needed to prevent memory leaks.

Fragmentation: Over time, dynamic memory allocation may lead to memory fragmentation, which can impact system performance.

QUEUE FLOW CHART:



DYNAMIC MEMORY ALLOCATION FLOW CHART:



QUEUE CODE:-

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Define a structure for an account
struct Account {
    int accountNumber;
    int pin;
    double balance;
};

// Define an array to store accounts
struct Account *accounts = NULL;
int numAccounts = 0;

// Function to create a new account
void createAccount(int accountNumber, int pin, double initialBalance) {
    struct Account *newAccount = (struct Account *)malloc(sizeof(struct
Account));

    if (newAccount == NULL) {
        printf("Memory allocation failed. Cannot create a new account.\n");
        return;
    }

    newAccount->accountNumber = accountNumber;
    newAccount->pin = pin;
    newAccount->balance = initialBalance;

    accounts = (struct Account *)realloc(accounts, (numAccounts + 1) *
sizeof(struct Account));

    if (accounts == NULL) {
        printf("Memory reallocation failed. Cannot create a new account.\n");
        free(newAccount);
        return;
    }

    accounts[numAccounts] = *newAccount;
    free(newAccount);
    numAccounts++;
    printf("Account created with account number %d and initial balance
```

```
%.2f.\n", accountNumber, initialBalance);
}

// Function to find an account by account number
int findAccountIndex(int accountNumber) {
   for (int i = 0; i < numAccounts; i++) {
      if (accounts[i].accountNumber == accountNumber) {
         return i;
      }
   }
   return -1;
}

// Function to authenticate a user
bool authenticate(int accountIndex, int pin) {
   if (accountIndex >= 0 && accountIndex < numAccounts) {
      if (accounts[accountIndex].pin == pin) {
         return true;
      }
   }
   return false;
}

// Function to check the account balance
void checkBalance(int accountIndex) {
   printf("Account balance: %.2f\n", accounts[accountIndex].balance);
}

// Function to deposit money
void deposit(int accountIndex, double amount) {
   accounts[accountIndex].balance += amount;
   printf("Deposit of %.2f successful. New balance: %.2f\n", amount,
accounts[accountIndex].balance);
}

// Function to withdraw money
void withdraw(int accountIndex, double amount) {
   if (amount <= accounts[accountIndex].balance) {
      accounts[accountIndex].balance -= amount;
      printf("Withdrawal of %.2f successful. New balance: %.2f\n", amount,
accounts[accountIndex].balance);
   } else {
      printf("Insufficient funds.\n");
   }
```

```
}

int main() {
    // Sample account creation
    createAccount(12345, 1111, 1000.0);
    createAccount(67890, 2222, 500.0);

    int accountNumber, pin;
    int accountIndex = -1;
    bool authenticated = false;

    printf("Welcome to the ATM\n");
    printf("Enter your account number: ");
    scanf("%d", &accountNumber);

    accountIndex = findAccountIndex(accountNumber);

    if (accountIndex != -1) {
        printf("Enter your PIN: ");
        scanf("%d", &pin);

        authenticated = authenticate(accountIndex, pin);

        if (authenticated) {
            printf("Authentication successful\n");
            int choice;
            double amount;

            printf("ATM Menu:\n");
            printf("1. Check Balance\n");
            printf("2. Deposit\n");
            printf("3. Withdraw\n");
            printf("Enter your choice: ");
            scanf("%d", &choice);

            switch (choice) {
                case 1:
                    checkBalance(accountIndex);
                    break;
                case 2:
                    printf("Enter the deposit amount: ");
                    scanf("%lf", &amount);
                    deposit(accountIndex, amount);
                    break;
```

```
            case 3:
                printf("Enter the withdrawal amount: ");
                scanf("%lf", &amount);
                withdraw(accountIndex, amount);
                break;
            default:
                printf("Invalid choice.\n");
                break;
        }
    } else {
        printf("Incorrect PIN. Access denied.\n");
    }
} else {
    printf("Account not found. Access denied.\n");
}

// Cleanup memory
free(accounts);

return 0;
}
```

DYNAMIC MEMORY ALLOCATION CODE:-

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<time.h>

/*
  node structure for implementing a queue to store transaction history
*/
typedef struct node {
    char statement[50];
    struct node* link;
} node;

/*
  ATM function prototypes
*/
void pinGeneration(void);
int checkPin(void);
void showBalance(int *);
void depositMoney(node **, int *);
void withdrawMoney(node **, int *);
void saveHistory(node **, char *);
void removeHistory(node **);
void showHistory(node **);

int main(void) {
    int choice1, choice2;
    int pinValid = 0, balance = 0;

    node *head = NULL;

    while (1) {
        printf("\n\nATM System\n=====================\n");
        printf("1. Generate PIN\n2. Use ATM\n3. Exit\n");
        printf("\nYour choice: ");
        scanf("%d", &choice1);

        switch (choice1) {
            case 1: pinGeneration();
                    exit(EXIT_SUCCESS);

            case 2: pinValid = checkPin();
```

```
            if (pinValid) {
               printf("\nValid PIN\n");
            } else {
               printf("\nInvalid PIN. Please generate a PIN if you don't have
one.\n");
               exit(EXIT_FAILURE);
            }

            /*
                                    On valid PIN entry by user, the ATM Menu is
presented to the user
                                 */
            while(pinValid) {
               printf("\nATM System Menu\n==============\n\n");
               printf("1. Check Balance\n2. Deposit\n3. Withdraw\n4. View
transaction history\n5. Quit\n\n");

               printf("Enter choice: ");
               scanf("%d", &choice2);

               switch(choice2) {
                  case 1: showBalance(&balance);
                        break;
                  case 2: depositMoney(&head, &balance);
                        break;
                  case 3: withdrawMoney(&head, &balance);
                        break;
                  case 4: showHistory(&head);
                        break;
                  case 5: printf("\nThank you for using the ATM\n");
                        exit(EXIT_SUCCESS);
                  default: printf("\nInvalid option entered!\n");
                         break;
               }
            }

         case 3: exit(EXIT_SUCCESS);

         default: printf("\nInvalid choice...Try again...\n");
                 break;
      }
   }
   return 0;
```

```
}

/*
  This function will search for the PIN entered by the user
  in the file where the PIN numbers are stored
  If PIN is found return 1
  Otherwise, return 0
*/
int checkPin(void) {
   FILE *fp;
   // buffer to read PIN and store from file
   char pin[8];
   // buffer to read PIN and store from user
   char keyPin[8];
   int pinValid = 0;

   printf("\n\nEnter the PIN: \n");
   scanf("%s", keyPin);

   fp = fopen("pin.txt", "r");
   if (NULL == fp) {
      printf("\nFile cannot be opened\n");
      exit(EXIT_FAILURE);
   }

   /*
     Search for the PIN entered by user in file pin.txt
   */
   while (fgets(pin, sizeof(pin), fp) != NULL) {
      if (strstr(pin, keyPin)) {
         pinValid = 1;
      }
   }
   fclose(fp);

   return pinValid;
}

/*
  This function will generate a 4-digit random number that
  is considered as PIN
*/
void pinGeneration(void) {
   FILE *fp;
```

```
    /*
          Generate a random 4 digit number
        */
    srand(time(NULL));
    int generatedPin = 1000+rand()%9000;

    printf("\nPIN generated successfully\n");
    printf("\nYour generated PIN: %d\n", generatedPin);
    printf("\nRe-run the program and use ATM with this PIN\n\n");

    fp = fopen("pin.txt", "a");
    if (NULL == fp) {
       printf("\nCannot open file!");
       exit(EXIT_FAILURE);
    }

    /*
      Write PIN to the file
    */
    fprintf(fp, "%d\n", generatedPin);
    fclose(fp);
}

/*
  This function will display the current balance amount
*/
void showBalance(int *balance) {
    printf("\nYour current balance is Rs.%d\n", *balance);
}

/*
  This function will add the money deposited to the balance
*/
void depositMoney(node **head, int *balance) {
    int depositAmount;
    /*
      buffer to store
    */
    char depositStmt[50];

    printf("\nEnter amount to deposit: ");
    scanf("%d", &depositAmount);
```

```
   if (depositAmount > 0) {
      *balance += depositAmount;
      printf("\nRs.%d deposited\n", depositAmount);

                /*
                  saving formatted string in depositStmt character array
                */
      snprintf(depositStmt, sizeof(depositStmt), "Rs.%d deposited\n",
depositAmount);
      saveHistory(head, depositStmt);
   } else {
      printf("\nInvalid amount entered\n.");
   }
}

/*
  This function will deduct the money withdrawn from the balance
*/
void withdrawMoney(node **head, int *balance) {
   int withdrawAmount;
   char withdrawStmt[50];

   printf("\nEnter amount to withdraw: ");
   scanf("%d", &withdrawAmount);

   if (withdrawAmount > 0) {
      if (withdrawAmount > *balance) {
         printf("\nCannot withdraw. Balance Rs.%d\n", *balance);
      } else {
         *balance -= withdrawAmount;
         printf("\nRs.%d withdrawn\n", withdrawAmount);

         /*
                        saving formatted string in withdrawStmt character array
                      */
         snprintf(withdrawStmt, sizeof(withdrawStmt), "Rs.%d withdrawn\n",
withdrawAmount);
         saveHistory(head, withdrawStmt);
      }
   } else {
      printf("\nInvalid amount entered\n.");
   }
}
```

```
/*
  This function will save a transaction statement
*/
void saveHistory(node **head, char *str) {
   static int count = 0;
   node *temp;
   temp = (node *)malloc(sizeof(node));

   strcpy(temp->statement, str);
   temp->link = NULL;

   if (NULL == *head) {
      *head = temp;
      count++;
   } else {
      if (10 == count) {
         removeHistory(head);
         count--;
      }
      node *p;
      p = *head;
      while (NULL != p->link) {
         p = p->link;
      }
      p->link = temp;
      count++;
   }
}

/*
  This function is used to remove the oldest transaction when
  10 transactions are made
*/
void removeHistory(node **head) {
   node *temp;
   temp = *head;
   *head = (*head)->link;
   temp->link = NULL;
   free(temp);
}

/*
  This function will display the transaction history
*/
```

```
void showHistory(node **head) {
   node *temp;
   temp = *head;

   if (NULL == temp) {
      printf("\nNo transaction history...\n");
   } else {
      printf("\nTransaction History\n-------------------\n\n");
      while (NULL != temp) {
         printf("%s\n", temp->statement);
         temp = temp->link;
      }
   }
}
```

QUEUE OUTPUT:-

```
Account created with account number 12345 and initial balance 1000.00.
Account created with account number 67890 and initial balance 500.00.
Welcome to the ATM
Enter your account number: 12345
Enter your PIN: 1111
Authentication successful
ATM Menu:
1. Check Balance
2. Deposit
3. Withdraw
Enter your choice: 1
Account balance: 1000.00
```

```
ATM Menu:
1. Check Balance
2. Deposit
3. Withdraw
Enter your choice: 3
Enter the withdrawal amount: 500
Withdrawal of 500.00 successful. New balance: 500.00
```

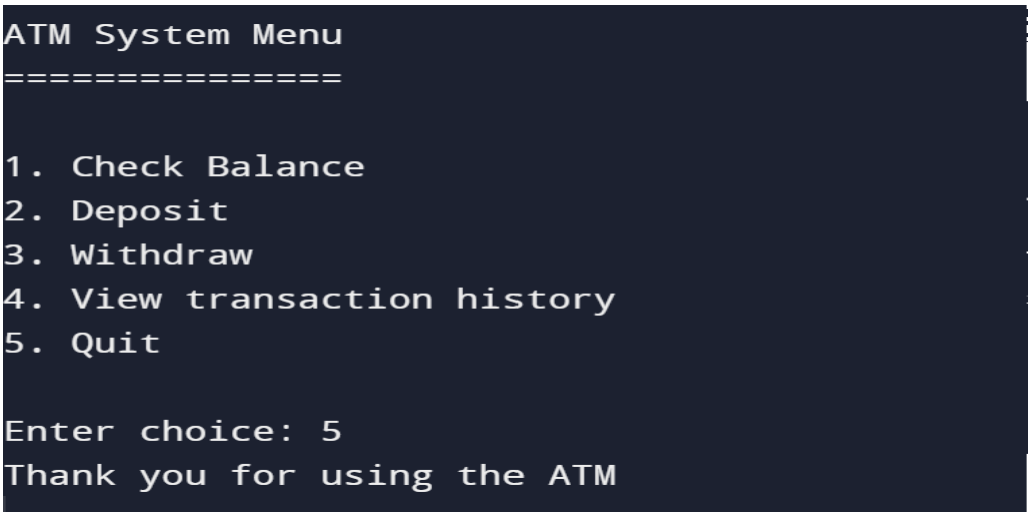
```
ATM Menu:
1. Check Balance
2. Deposit
3. Withdraw
Enter your choice: 3
Enter the withdrawal amount: 500
Withdrawal of 500.00 successful. New balance: 500.00
```

DYNAMIC MEMORY ALLOCATION OUTPUT:-

```
ATM System
======================
1. Generate PIN
2. Use ATM
3. Exit

Your choice: 1
PIN generated successfully

Your generated PIN: 2434

Re-run the program and use ATM with this PIN
```

```
ATM System
======================
1. Generate PIN
2. Use ATM
3. Exit

Your choice: 2
Enter the PIN:
2434
Valid PIN

ATM System Menu
===============

1. Check Balance
2. Deposit
3. Withdraw
4. View transaction history
5. Quit

Enter choice:
```

```
Enter choice: 2
Enter amount to deposit: 10000
Rs.10000 deposited

ATM System Menu
===============

1. Check Balance
2. Deposit
3. Withdraw
4. View transaction history
5. Quit

Enter choice: 3
Enter amount to withdraw: 2000
Rs.2000 withdrawn
ATM System Menu
===============

1. Check Balance
2. Deposit
3. Withdraw
4. View transaction history
5. Quit

Enter choice: 4
Transaction History
-------------------

Rs.10000 deposited

Rs.2000 withdrawn
```



TIME COMPLEXITY:-
The time complexity of using a queue and dynamic memory allocation in an ATM machine depends on the specific operations you're considering.

Queue:
Enqueue (adding a customer request to the queue) is typically O(1) as it involves appending an element to the end of the queue.
Dequeue (processing a request from the front of the queue) is also typically O(1).

Dynamic Memory Allocation:
Allocating memory dynamically (e.g., for storing customer data) can have a time complexity of O(1) for small allocations in many cases. However, it can become O(n) if the memory allocator needs to search for a suitable block of memory. Deallocating memory can be O(1) or O(n), depending on the specific memory management system.

In an ATM machine context, if you're mainly concerned with managing customer requests and transactions, a queue is typically more efficient as it provides constant-time complexity for enqueue and dequeue operations. Dynamic memory allocation may introduce unnecessary overhead unless you have specific requirements for managing customer data that necessitate dynamic memory.

Keep in mind that the actual performance also depends on the efficiency of the underlying data structures and algorithms used in the implementation, as well as the scale and complexity of the ATM system.

CONCLUSION:-

In conclusion, the design of an ATM (Automated Teller Machine) system that leverages data structures, particularly queues, and dynamic memory allocation, plays a critical role in ensuring the efficiency, security, and scalability of the system. Here's a summary of the key points:

Queue in ATM System: Queues are a fundamental data structure in an ATM system for managing customer requests. The FIFO (First-In, First-Out) nature of queues ensures that customer transactions are processed in the order they are initiated. Customers join a queue when they arrive at the ATM, and the ATM processes their transactions based on this order.

Queue in Transaction Processing: In the context of an ATM system, queues are used to maintain the order of transactions. When a customer initiates a transaction, it is added to the queue. The ATM dequeues transactions in the order they were added, ensuring fair and consistent service to customers.

Dynamic Memory Allocation: Dynamic memory allocation is essential for an ATM system as it allows the system to adapt to varying workloads and customer demands. It enables the allocation of memory for customer data, transaction histories, and other dynamic components based on the system's runtime requirements.

Dynamic Memory Allocation for Transaction Histories: Dynamic memory allocation is particularly useful for managing transaction histories, which can grow over time. Memory is allocated as needed to store the transaction data, and as transactions are completed, memory can be deallocated to prevent memory wastage.

Security and Data Integrity: Proper dynamic memory allocation ensures that customer data, PINs, and transaction details are handled securely. Careful memory management is crucial to prevent data leakage and unauthorized access to sensitive information.

Scalability: Dynamic memory allocation enables the ATM system to scale as the number of customers and transactions increase. The system can adapt by allocating additional memory resources to accommodate the growing demand.

Efficiency: Dynamic memory allocation, when used efficiently, helps reduce memory fragmentation and optimizes resource usage. This ensures that the ATM system runs smoothly and without unnecessary memory overhead.

Incorporating queue data structures and dynamic memory allocation into an ATM system is a powerful approach to address the dynamic nature of banking operations. This combination enhances the system's responsiveness, scalability, and security, providing customers with a seamless and reliable banking experience. However, it's crucial to implement these concepts with a strong emphasis on data security and efficient memory management to ensure the system's reliability and integrity.