East West University

Data Structures (CSE207)

Title: Food Ordering System

(Project Report)

Submitted by –

Name	ID
Md. Naeemur Rahman	2022-1-60-167
Md. Sadat Ahmed Rafi	2022-1-60-261
Md.Alamin Hossen	2022-1-60-043

Section: 07

Semester: Summer 2023

Submitted to –

Amit Mandal

Lecturer

Department of Computer Science and Engineering

Date of Submission: 16 September 2023

USED DATA STRUCTURE

Linked List
Recursion
Binary Search Tree
HashMap
Heap

KEY POINTS

Objective: This project aims to create a simple yet functional food ordering system in C programming language.

User Authentication: The system provides user authentication functionality where users (customers) can register, login, and update their shipping addresses. The system owner has a special account. Users can register an account, and their data is stored in a hashmap. The system owner's account is registered automatically.

Data Storage: User accounts, product details, and orders are stored in various data structures like hashmaps, binary search trees (BSTs), and files.

Product Management: Products are managed using a Binary Search Tree (BST). Admins can add, remove, and view products.

Ordering: Users can place orders by selecting a product and providing their details. Order information is stored in a file.

Price Filtering: Users can filter products by price, viewing them from highest to lowest or lowest to highest using heap algorithm.

Heap Data Structures: The project demonstrates the use of max-heap and min-heap data structures to sort and display products by price.

File Handling: Products, orders, and user data are read from and written to text files for persistence.

Interactive Command Line Interface: The project has an interactive command-line interface for user interaction, including menus and prompts.

Error Handling: The system checks for errors, such as existing usernames during registration and file opening failures.

Modularity: The code is organized into functions and structures for better maintainability and readability.

Security: User passwords are stored securely and verified during authentication.

HASHMAP

Below functions are essential parts of the project's user authentication and administration system. They maintain data security while enabling user registration, login, and account management. User passwords are securely managed, and user account information is kept in a HashMap employing chaining to handle collisions. The system searches for distinct usernames during registration and creates a new user account if one is detected. Users enter their username and password to authenticate, and if the two are matched, they are given access to user-specific features. The foundation of user engagement within the food ordering system is formed by error management, which gives users feedback while preserving the confidentiality and integrity of their data.

```
struct HashMap {
    struct HashMapEntry* table[HASH_MAP_SIZE];
};
struct UserAccount{
    char username[50];
    char password[20];
    char address[100];
};
struct HashMapEntry {
    char username[50];
    struct UserAccount* account;
    struct HashMapEntry* next;
};
struct UserHashMap{
    struct HashMapEntry* table[HASH_MAP_SIZE];
};

void initUserHashMap(struct UserHashMap* hashmap){
    for(int i = 0; i<HASH_MAP_SIZE;i++){
        hashmap->table[i] = NULL;
    }
}

int chartoAscii(char* str){
    int hash = 0;//sum of ascii values
    int i = 0;
    while(str[i]!= '\0'){
```

```
        hash+= str[i];
        i++;
    }
    return hash;
}
// Function to search for a user by username in the hashmap
struct UserAccount* searchUser(struct UserHashMap* hashMap, char* username) {
    // Calculate the hash table index number for the username
    int index =  chartoAscii(username)%HASH_MAP_SIZE;
    // Get the pointer to the first entry in the corresponding bucket
    struct HashMapEntry* current = hashMap->table[index];


    while (current != NULL) {
        if (strcmp(current->account->username, username) == 0) {
            // Username found, return the associated user account
            return current->account;
        }
        current = current->next;
    }

    // Username not found in the hashmap
    return NULL;
}


void insertUser(struct UserHashMap* hashMap, char* username, struct UserAccount*
newUser) {
    // Calculate the hash code for the username

    int index =  chartoAscii(username)%HASH_MAP_SIZE;//hashmap index number using

    // Create a new entry for the user
    struct HashMapEntry* newEntry = (struct HashMapEntry*)malloc(sizeof(struct
HashMapEntry));
    strcpy(newEntry->username, username);
    newEntry->account = newUser;
    newEntry->next = NULL;

    // Check if the bucket is empty (no collisions)
    if (hashMap->table[index] == NULL) {
        // Set the new entry as the first entry in the bucket
        hashMap->table[index] = newEntry;
    } else {
        // Chaining  Handle collision by adding to the linked list
        struct HashMapEntry* current = hashMap->table[index];
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newEntry;
    }
}

void registerUser(struct UserHashMap* hashmap,char* username, char* password, char*
address ){
    if(searchUser(hashmap, username)!= NULL){
```

```
        printf("Username %s is already taken. Please try another name\n", username);
        return;
    }
    else{

    struct UserAccount* newUser = (struct UserAccount*)malloc(sizeof(struct
UserAccount));
    strcpy(newUser->username, username);
    strcpy(newUser->password, password);
    strcpy(newUser->address,address);

    insertUser(hashmap,username,newUser);
}
//printf("User '%s' registered successfully.\n", username);

}

// Function to authenticate a user
struct UserAccount* authenticateUser(struct UserHashMap* hashMap, char* username,
char* password) {
    struct UserAccount* user = searchUser(hashMap, username);

    if (user != NULL && strcmp(user->password, password) == 0) {//search in user
account
        printf("User '%s' authenticated successfully.\n", username);
        return user;
    }
    printf("Authentication failed for user '%s'.\n", username);
    return NULL;
}
```

BINARY SEARCH TREE

These processes work together to construct a Binary Search Tree (BST) for the project's management of product information. 'createNode' generates a new BST node, whereas 'insert' inserts a product into the BST depending on its name. The'searchProduct' method returns a product by name, and the 'deleteNode' function deletes a node from the BST while handling a variety of scenarios, including nodes with one or two children. When deleting a node, the 'inOrderPredecessor' function helps locate the predecessor node. These actions allow for the effective storage, retrieval, and removal of products from the project,

optimizing the essential system components of product management and inventory control.

```
struct Product {
    char name[100];
    double price;
    int id;
};

struct TreeNode {
    char key[100];
    struct Product product;
    struct TreeNode* left;
    struct TreeNode* right;
};

// Function to create a new TreeNode
struct TreeNode* createNode(const char* key, struct Product product) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    if (newNode != NULL) {
        strcpy(newNode->key, key);
        newNode->product = product;
        newNode->left = newNode->right = NULL;
    }
    return newNode;
}
// Function to insert a product into the BST
struct TreeNode* insert(struct TreeNode* root, const char* key, struct Product
product) {
    if (root == NULL) {
        return createNode(key, product);
    }

    int compareResult = strcmp(key, root->key);

    if (compareResult < 0) {
        root->left = insert(root->left, key, product);
    } else if (compareResult > 0) {
        root->right = insert(root->right, key, product);
    }

    return root;
}

// Function to search for a product in the BST
struct TreeNode* searchProduct(struct TreeNode* root, const char* key) {
    if (root == NULL) {
        return NULL;
    }

    int compareResult = strcmp(key, root->key);
```

```
    if (compareResult < 0) {
        return searchProduct(root->left, key);
    } else if (compareResult > 0) {
        return searchProduct(root->right, key);
    } else {
        return root;
    }
}
```

```
struct TreeNode* inOrderPredecessor(struct TreeNode* node) {
    struct TreeNode* current = node->left;
    while (current && current->right) {
        current = current->right;
    }
    return current;
}

// Function to delete a node with a given key from the BST
struct TreeNode* deleteNode(struct TreeNode* root, const char* key) {
    if (root == NULL) {
        return root;
    }

    // smaller means its in left subtree
    if (strcmp(key, root->key) < 0) {
        root->left = deleteNode(root->left, key);
    }
    // bigger means its in right subtree
    else if (strcmp(key, root->key) > 0) {
        root->right = deleteNode(root->right, key);
    }
    // key in same as the root's key, then this is the node to be deleted
    else {
        // Node with only one child or no child
        if (root->left == NULL) {
            struct TreeNode* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct TreeNode* temp = root->left;
            free(root);
            return temp;
        }

        // Node with two children: Get the in-order predecessor
        struct TreeNode* temp = inOrderPredecessor(root);

        // Copy the in-order predecessor's content to this node
        strcpy(root->key, temp->key);
        root->product = temp->product;

        // Delete the in-order predecessor
        root->left = deleteNode(root->left, temp->key);
    }
```

```
    return root;
}
```

HEAP

These procedures control the MaxHeap and MinHeap heap types to provide effective price-based product retrieval. The commands "createMaxHeap" and "createMinHeap" start these heaps away. 'insertMax' and 'insertMin' insert products while keeping heap attributes for easy retrieval of top and lowest prices. 'maxHeapify' and'minHeapify' ensure heap structure after insertions. For user experience and product suggestions, these processes offer price-optimized product selection.

MAX HEAP

```
struct MaxHeap {
    struct Product* array;
    int size;
    int capacity;
};

// Function to create a new max-heap
struct MaxHeap* createMaxHeap(int capacity) {
    struct MaxHeap* heap = (struct MaxHeap*)malloc(sizeof(struct MaxHeap));
    heap->capacity = capacity;
    heap->size = 0;
    heap->array = (struct Product*)malloc(sizeof(struct Product) * capacity);
    return heap;
}

// Function to swap two products in the heap
void swap(struct Product* a, struct Product* b) {
    struct Product temp = *a;
    *a = *b;
    *b = temp;
}


void maxHeapify(struct MaxHeap* heap, int index) {
    int largest = index;
    int left = 2 * index + 1;
    int right = 2 * index + 2;
```

```
    if (left < heap->size && heap->array[left].price > heap->array[largest].price) {
        largest = left;
    }

    if (right < heap->size && heap->array[right].price > heap->array[largest].price)
{
        largest = right;
    }

    if (largest != index) {
        swap(&heap->array[index], &heap->array[largest]);
        maxHeapify(heap, largest);
    }
}

// Function to insert a new product into the max-heap
void insertMax(struct MaxHeap* heap, struct Product product) {
    if (heap->size == heap->capacity) {
        printf("Heap is full. Cannot insert.\n");
        return;
    }

    int index = heap->size;
    heap->array[index] = product;
    heap->size++;

    while (index > 0 && heap->array[index].price > heap->array[(index - 1) /
2].price) {
        swap(&heap->array[index], &heap->array[(index - 1) / 2]);
        index = (index - 1) / 2;
    }
}
```

MIN HEAP

```
struct MinHeap {
    struct Product* array;
    int size;
    int capacity;
};


struct MinHeap* createMinHeap(int capacity) {
    struct MinHeap* heap = (struct MinHeap*)malloc(sizeof(struct MinHeap));
    heap->capacity = capacity;
    heap->size = 0;
    heap->array = (struct Product*)malloc(sizeof(struct Product) * capacity);
    return heap;
}
```

```c
void minHeapify(struct MinHeap* heap, int index) {
    int smallest = index;
    int left = 2 * index + 1;
    int right = 2 * index + 2;

    if (left < heap->size && heap->array[left].price < heap->array[smallest].price) {
        smallest = left;
    }

    if (right < heap->size && heap->array[right].price < heap->array[smallest].price)
{
        smallest = right;
    }

    if (smallest != index) {
        swap(&heap->array[index], &heap->array[smallest]);
        minHeapify(heap, smallest);
    }
}


void insertMin(struct MinHeap* heap, struct Product product) {
    if (heap->size == heap->capacity) {
        printf("Heap is full. Cannot insert.\n");
        return;
    }

    int index = heap->size;
    heap->array[index] = product;
    heap->size++;

    while (index > 0 && heap->array[index].price < heap->array[(index - 1) /
2].price) {
        swap(&heap->array[index], &heap->array[(index - 1) / 2]);
        index = (index - 1) / 2;
    }
}
```

DATA STORING (FILE)

```
int chartoAscii(char* str){
void saveProductsToFile(struct TreeNode* root, const char* filename) {
    FILE* file = fopen(filename, "w"); // Open the file for writing (text mode)
    if (file == NULL) {
        printf("Error opening file for writing.\n");
        return;
    }

    // Write each product node to the file
    saveProductToFile(root, file);

    fclose(file);
}
```

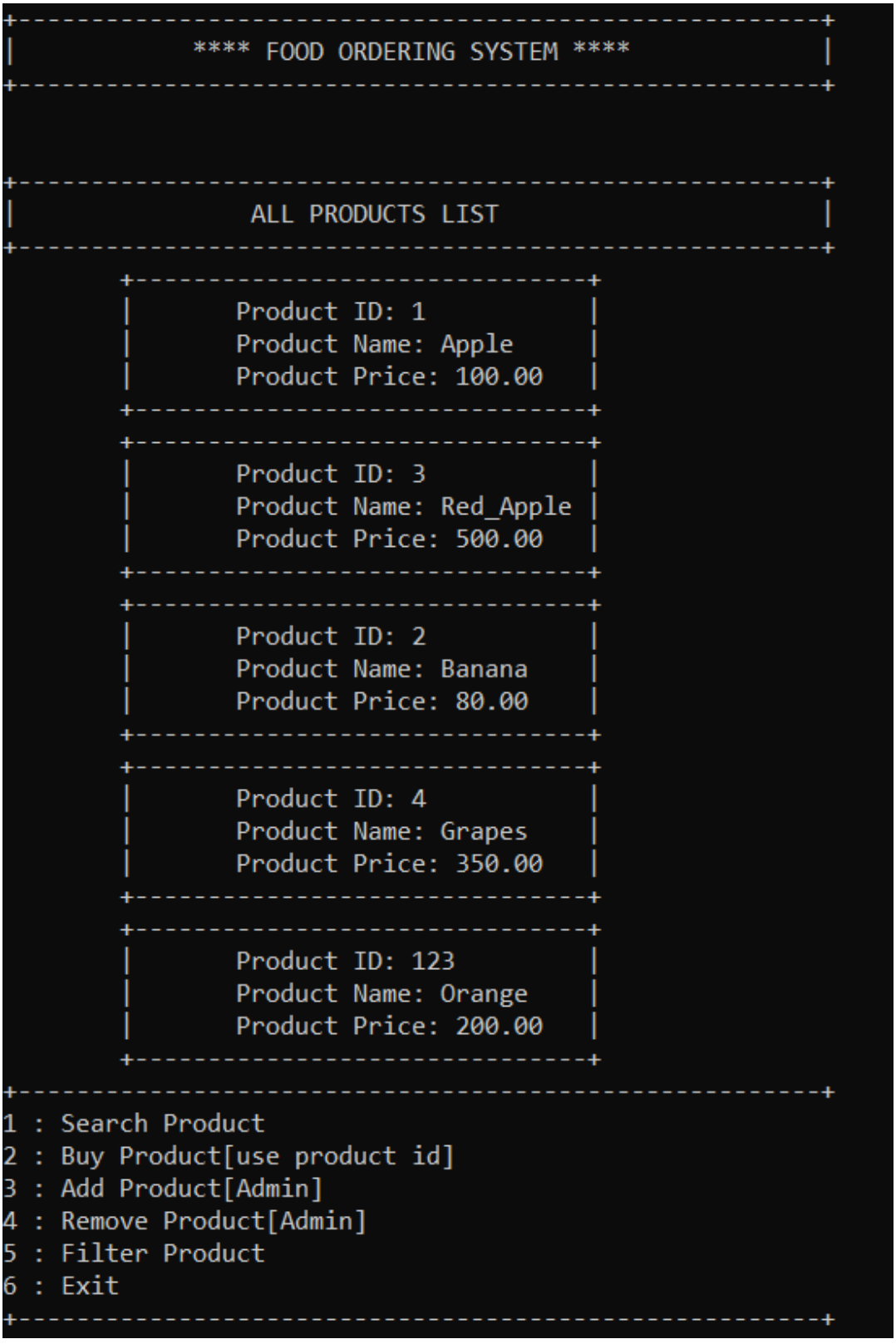
- This function was used for saving the product list into the file named products.txt

SYSTEM MENU

```
  printf("+-------------------------------------------------------+\n");
    printf("1 : Search Product \n");
    printf("2 : Buy Product[use product id]\n");//customer
    printf("3 : Add Product[Admin] \n");//admin
    printf("4 : Remove Product[Admin]\n");//admin
    printf("5 : Filter Product \n");
    printf("6 : Exit\n");
        printf("+-------------------------------------------------------+\n");
    printf("\nChoose Option :  ");
```

- This is the system menu.

SYSTEM OUTPUT



```
Choose Option :  1

Enter the product name to search for: Apple
Product Found:
        +--------------------------------+
        |       Product ID: 1            |
        |       Product Name: Apple      |
        |       Product Price: 100.00    |
        +--------------------------------+
```

```
Choose Option :  5

+-------------------------------------------------------+
1 : Show highest to lowest price
2 : Show lowest to highest price
3 : Back to main menu
+-------------------------------------------------------+

Enter your choice: 1
Products from highest to lowest price:
        +--------------------------------+
        |       Product ID: 3            |
        |       Product Name: Red_Apple  |
        |       Product Price: 500.00    |
        +--------------------------------+
        +--------------------------------+
        |       Product ID: 4            |
        |       Product Name: Grapes     |
        |       Product Price: 350.00    |
        +--------------------------------+
        +--------------------------------+
        |       Product ID: 123          |
        |       Product Name: Orange     |
        |       Product Price: 200.00    |
        +--------------------------------+
        +--------------------------------+
        |       Product ID: 1            |
        |       Product Name: Apple      |
        |       Product Price: 100.00    |
        +--------------------------------+
        +--------------------------------+
        |       Product ID: 2            |
        |       Product Name: Banana     |
        |       Product Price: 80.00     |
        +--------------------------------+
```

Improvement Opportunities

To enhance the project, features like order history, product categories, and better error handling could be added.

Conclusion:

This Food Ordering System provides essential functionalities for users to register, authenticate, order products, and manage the product list. It demonstrates data storage, retrieval, and manipulation techniques, making it a useful foundation for further development.